Programming in C/C++

Data Structures (`structs`) – Take Home Lecture

(if we have time can START in class)

Kristina Shroyer

IMPORTANT NOTES

- I have assigned this as a take home lecture
  - It IS fair game for the objective exam
  - Understanding this WILL help you with the main topic on the programming final which is objects (we'll do that next)

- This is the only Chapter in the book where I recommend you **use only the lecture notes and not the book**
  - We are going to learn structures as abstract data types
    - The book goes into some other ways to use C++ structures that can confuse you if this is your first time learning this material
    - If you have a lot of programming experience feel free to use both sources (notes and book) for information

Objectives

You should be able to describe:

- Structures
- Arrays of Structures
- Structures as Function Arguments
  - We only will look at a couple of cases of this for now, we'll look at more specifics when we do pointers after Spring Break

Structures (structs)

- **Concept:** C++ allows you to group a set of variables into a single item known as a structure
  - So for you've written two types of programs
    - Programs that keep data in individual variables (built in types and class types)
    - Programs that use arrays:
      - Arrays make it possible to access list of data of ***same data type*** using single variable name
- Sometimes a relationship exists between two or more separate variables (of ***different*** data types)
  - Need different mechanism to store information of varying data types in one **data structure (specialized data type – without the operations)**
    - Mailing list data (name, address, city, state, zip code)
    - Parts inventory data (name of part, number in inventory)
  - Think of a structure as a ***data type (without operations, we'll add those when we do object oriented programming)*** that contains more than one variable and the variables can be different data types
    - Remember a data type is a set of data and the operations that act on that set of data
      - A struct is really just half a data type because it defines a new set of data
        - The operations are going to only happen to the struct components or members as we will see

Structures

- **Example:** Variables holding Payroll Data
  - All of these variables listed are related because they can hold data about the same employee
  - The variable definition statements alone however do not make it clear these variables are related to each other
  - C++ gives you the ability to create a relationship between variables such as these by packaging them together into a **structure…**this structure allows you to create a ***data type*** with more than one variable in it

Variable Definition	Data Held
`int empNumber`	Employee Number
`string name`	Employee Name
`float hours`	Hours worked
`float payRate`	Hourly pay rate
`float grossPay`	Gross Pay

Structures

- **Structure:** Data structure that stores different types of data under single name

- Creating and using a structure requires two steps:
  - Declaration
  - Assigning values

Structure Declarations

- Before a structure can be used it must be declared
- **Structure declaration:** List the data types, data names and arrangements of data items
- Format for structure declaration:

```
struct tag
{
  variable1 declaration;
  variable2 declaration;
  //as many declaration
};
```

- What are we really doing here?
  - **Creating our own unique data type**
  - ***IMPORTANT:***
    - **We are NOT setting aside any memory here, we are merely creating a new data type based on a customized set of data**

Structure Declarations

- Format for structure declaration:

```
struct tag
{
  variable1 declaration;
  variable2 declaration;
  //as many variable declarations as you want
};  //note the required semicolon
```

- The **tag** is the name of the structure (the name of our new customized data type). It is used like a data type name (so it is similar to **`int`**, **`double`**, **`float`** etc)

- The variable declarations that appear inside the braces declare the **members** of the structure

Structure Declarations

- Payroll Data Example

```
struct Payroll
{
    int empNumber;
    string name;
    float hours;
    float payRate;
    float grossPay;
};
```

  - This declaration declares a structure named `Payroll` with 5 members
  - So Payroll is a **data type** that consists of 5 parts
  - You will notice I started the tags of the structures (`Payroll`) with a capital letter.
    - I used this convention to differentiate these names (names of data types) from the names of variables
  - **Notice a semi colon is required after the closing brace of the structure declaration**

Structure Declarations

- Payroll Data Example

```
struct Payroll
{
    int empNumber;
    string name;
    float hours;
    float payRate;
    float grossPay;
};
```

- So what is this `struct Payroll`? What does it represent?
  - It's a declaration of a **data type** that was defined by you the user
- **It's important to note that the Payroll structure declaration above does NOT define a variable**
  - It simply tells the compiler what our Payroll structure is made of
  - **It is a MODEL for what a Payroll structure (data type of sorts) is**
  - **In essence it creates a new data type called Payroll**

Defining Variables of type `struct`

- So if we only have a declaration for our **`Payroll`** **data type**, how do we declare variables of this type so we can use them in our programs?
- Once you have created/defined your data type by declaring your **`struct`** you can then define variables of this data type with simple definition statements just as you would with any other data type
- To define a variable of the Payroll data type we just created:

```
Payroll deptHead;
```

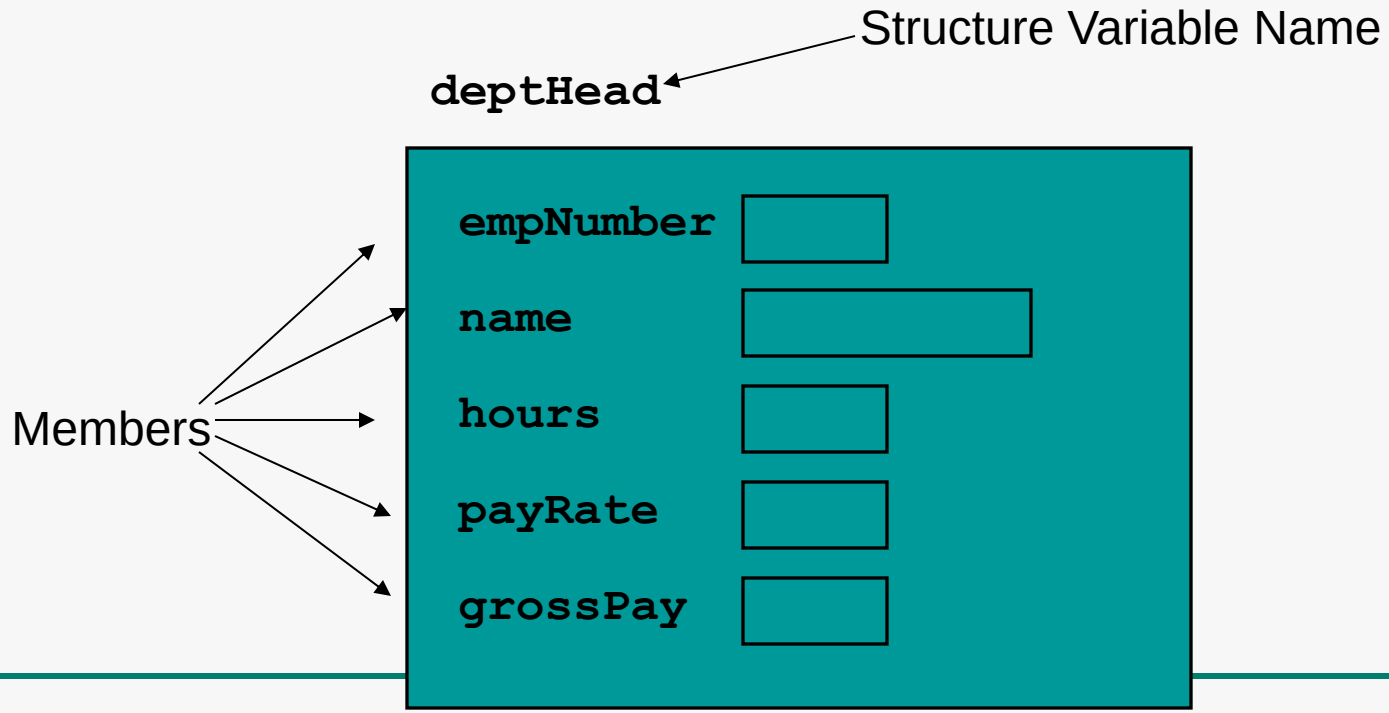
- The data type of department head is a **`Payroll`** structure (department head is of type **`Payroll`**)
  - The structure tag payroll is placed before the variable name (**`deptHead`**) in the same manner the word **`int`** or **`float`** would be to define variables of those types

Defining Variables of type `struct`

- The deptHead variable is of type Payroll:

```
Payroll deptHead;
```

- Remember that structure variables are actually made up of other variables called **members**
- Because **`deptHead`** is a Payroll structure it contains copies of the five members listed in the Payroll structure declaration
- **There is NOW space in memory to hold a value in each of the Payroll structures members**
  - Before the variable declaration, all we had was a definition of a data type…the variable declaration sets aside the memory we need to use our **`struct`** data type

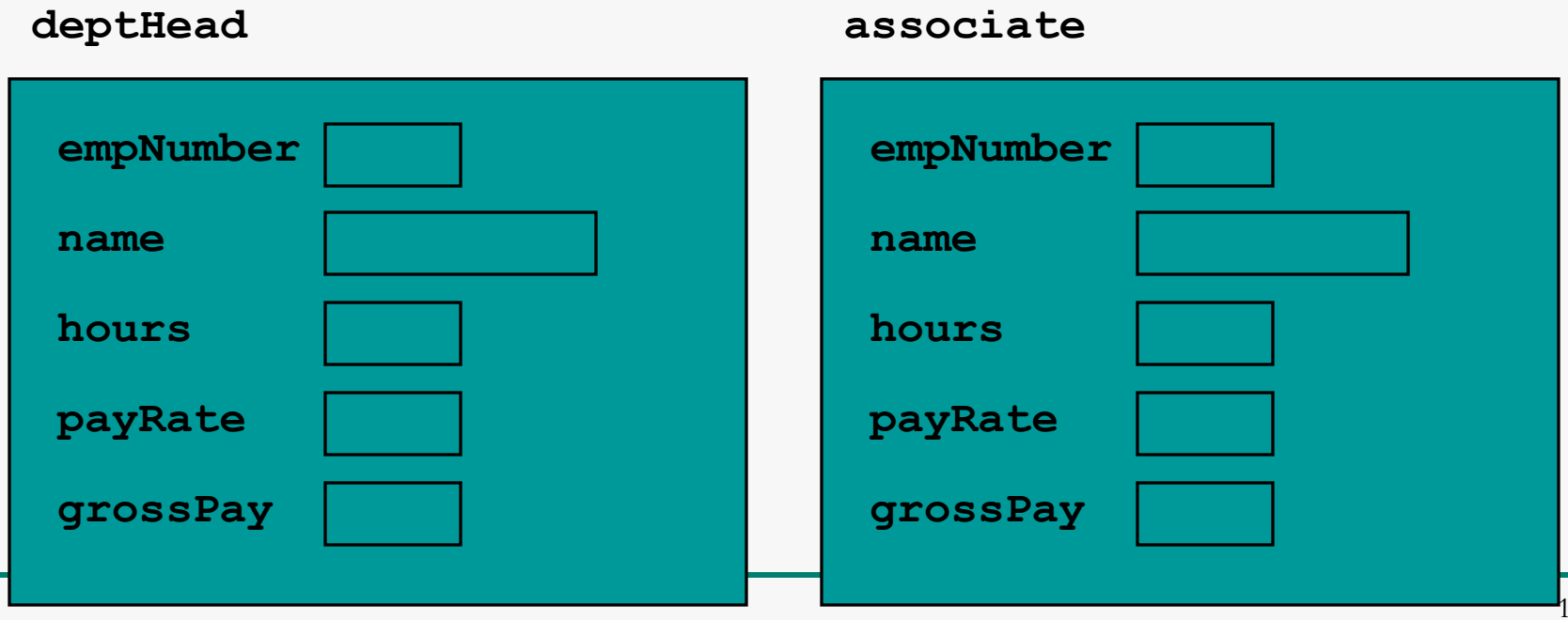


Defining Multiple Variables of type `struct`

- Just as its possible to define multiple `int` or `float` or other types of variables it's also possible to define multiple variables of a structure type
- Variable definitions of type Payroll:

```
Payroll deptHead, associate;
```

- Each of these variables is a separate **instance** of the Payroll structure with its own memory allocated to hold its members



Defining Multiple Variables of type `struct`

- Example

```
struct Payroll
{
    int empNumber;
    string name;
    float hours;
    float payRate;
    float grossPay;
};

Payroll deptHead, associate;
```

  - Even though these structure variables have distinct names each contains members of the same name
    - However these members belong to those variables and have their own space
    - The members of each unique variable of the `struct` data type are accessed separately

Format of Structure Declaration

- Format of structure definition is flexible: The following is valid

```
struct Birthday{int month; int day; int year;};
```

- Although the above formatting is valid, this format is not as readable as the following two formats.
  - So it's better to use one of these formats
  - For this class use one of these two formats:

```
struct Birthday {
   int month;
   int day;
   int year;
};

struct Birthday
{
   int month;
   int day;
   int year;
};
```

Declaring `structs` and Defining Variables of Type `struct`

- Summary:
  - There are two steps to implementing a structure in a program

1. Create the structure declaration. This establishes the tag (or name) of the structure (data types) and a list of items that are the structure's members (this does NOT create any variables)

2. Define variables which are instances of the structure and then use these variables in the program to hold data

Accessing Structure Members

- **Concept:** The **dot operator** allows you to access the individual members of a structure in a program

- **Populating the structure:** Assigning data values to individual data items (members) of a structure type variable
  - Individual data structure members are accessed by giving the structure name and individual data item name separated by period (a dot)
  - General Format to access a member (data item) of a `struct` type
    - `structName.memberName`
    - The period is called the **member access operator (dot operator)**

Accessing Structure Members

- **Example:**

```
struct Payroll
{
   int empNumber;
   string name;
   float hours;
   float payRate;
   float grossPay;
};

Payroll deptHead, associate;

deptHead.empNumber = 475;
associate.empNumber = 823;
```

- In this example the number 475 is assigned to the **`empNumber`** member of the **`deptHead`** variable of type Payroll
- **The dot operator connects the name of the member with the name of the structure variable which it belongs to**
- The second statement assigns a value to the **`empNumber`** member of the associate variable of type **`struct`**.

Accessing Structure Members

- **With the dot operator (.) you can use structure member variables just like regular variables**
- The code below displays the contents of deptHead's members

```
struct Payroll
{
    int empNumber;
    string name;
    float hours;
    float payRate;
    float grossPay;
};

Payroll deptHead, associate;

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

Accessing Structure Members – Example (Example 1 - payrollStructure1.cpp)

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

struct Payroll //declaration of the payroll struct
{
   int empNumber;
   string name;
   float hours;
   float payRate;
   float grossPay;
};  //don't forget the semi colon

int main()
{
   Payroll employee;  //variable of type Payroll

   cout << "Enter the employee's number: ";  //get the employee data
   cin >> employee.empNumber;
   cout << "Enter the employee's name: ";
   cin.ignore();  //to ignore the \n character left in the buffer from cin
   getline(cin, employee.name);  //had to use getline to allow first and last name
   cout << "How many hours did the employee work? ";
   cin >> employee.hours;
   cout << "What was the employee's pay rate? ";
   cin >> employee.payRate;

   employee.grossPay = employee.hours * employee.payRate;  //calc gross  pay
   cout << "The gross pay for employee #" << employee.empNumber << " named "
          << employee.name << " is $" << fixed << setprecision(2) << employee.grossPay << endl;

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

Displaying and Comparing Structure Variables

- In the previous example each **member** of the `employee` structure variable was displayed separately
  - This was necessary because a structure variable cannot be displayed by simply passing the whole variable to `cout`
    - **ILLEGAL:**
      - `cout << employee << endl;`

- In the same regard, it is possible to compare the contents of two individual structure members, but you cannot compare entire structure variables
    - **ILLEGAL:**
      - `if(employee1 > employee2)`
    - **LEGAL**
      - `if(employee1.hours > associate2.hours)`

- Individual structure members that hold a single data item such as a `float`, `int`, or a `string` can be used like any regular variable once you access them with the dot operator
  - The first example showed this with `cin`, `cout`, and addition
  - This second example shows **individual structure variables** being passed to functions

Example (Example 2 - CircleStructure.cpp)

```
/*This program uses a structure to hold geometric data about a circle*/
#include <iostream>
#include <iomanip>
#include <cmath>  //needed to use the pow function
using namespace std;

struct Circle  //declares the circle structure
{
   double radius;
   double diameter;
   double area;
 };

int main()
{
   const double PI = 3.14159;

   Circle circ1, circ2;  //define 2 circle structure variables

   //get the Circle diameters
   cout << "Enter the diameter of circle 1: ";
   cin >> circ1.diameter;
   cout << "Enter the diameter of circle 2: ";
   cin >> circ2.diameter;

   //Perform calculations
   circ1.radius = circ1.diameter/2;
   circ1.area = PI * pow(circ1.radius, 2.0);
   circ2.radius = circ2.diameter/2;
   circ2.area = PI * pow(circ2.radius, 2.0);

   //output results
   cout << "\nThe radius and area of the circles are: \n";
   cout << "Circle 1 -- Radius: " << setw(6) << circ1.radius
        << "   Area:  " << setw(6) << circ1.area << endl;
   cout << "Circle 2 -- Radius: " << setw(6) << circ2.radius
        << "   Area:  " << setw(6) << circ2.area << endl;

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

Initializing a Structure

- The members of a structure variable may be initialized with starting values when the structure variable is defined

- There are two ways a **structure variable** can be initialized when it is defined:
  - Using an Initialization List
  - Using a Constructor

Initializing a Structure

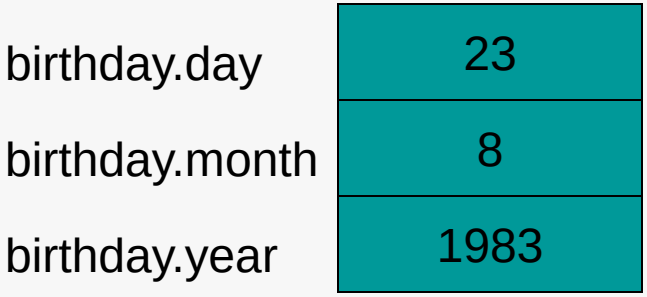
- The simplest way to initialize the members of a structure variable is to use an **initialization list**
  - An **initialization list** is a list of values used to initialize a set of memory locations
- Example:

```
struct Date
{
     int day;
     int month;
     int year;
};
```

  - A date variable can now be defined and initialized by following the variable name with the assignment operator and an initialization list

```
Date birthday = {23, 8, 1983};
```

    - This statement defines birthday to be a variable that is a Date structure
    - The values following the definition are assigned to its values in order



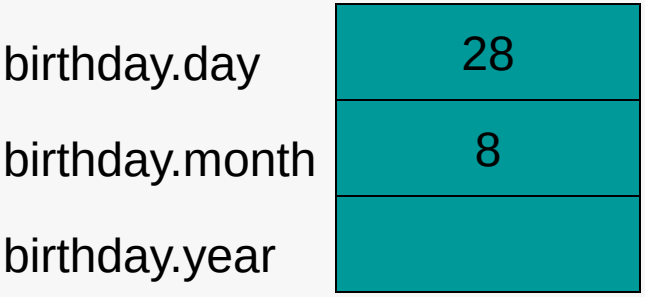
Initializing a Structure

- **Partial Initialization:** It is also possible to initialize just some of the members of a structure variable.
- Example: If we know the birthday to be stored is August 28 but we don't know the year

```
Date birthday = {28, 8};
```

  - Only the day and month members are initialized

- **If you leave a structure member uninitialized you must leave all the members that follow it uninitialized as well**
  - C++ does not provide a way to skip members using an initialization list
    - **ILLEGAL:** `Date birthday = {28, ,1988};`



Initializing a Structure

- You cannot initialize a structure member in the declaration of the structure
- Why?
  - A structure is NOT a variable
  - A structure only creates a new data type
  - Only variables set aside storage in memory
  - Only variables can be initialized
- **ILLEGAL:**

```
struct Date
{
    int day = 23;    //ILLEGAL
    int month = 8;   //ILLEGAL
    int year = 1983; //ILLEGAL
 };
```

- A structure declaration only declares what a structure "looks like"
  - **The member variables are created and stored in memory ONLY when a structure variable is created with a definition statement**
    - Until this occurs there is no place to store an initial value
    - This is called creating an instance of a struct

Initializing a Structure

- **Example3 – payrollStruct2.cpp :** Modify Payroll Program with partial initialization list
  - Imagine the base pay for an employee is 15.50 per hour and all employees work at least 5 hours per week
  - Note you can change these variables after they are initialized with assignment statements as the user finds out the employee info they are missing

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

struct Payroll //declaration of the payroll struct
{
  int empNumber;
  string name;
  float hours;
  float payRate;
  float grossPay;
};  //don't forget the semi colon

int main()
{

   Payroll employee = {0, "", 5.0, 15.50};  //variable of type Payroll - ADD PARITIAL INIT
        LIST

   employee.grossPay = employee.hours * employee.payRate;  //calc gross  pay
   cout << "The gross pay for employee #" << employee.empNumber << " named "
        << employee.name << " is $" << fixed
        << setprecision(2) << employee.grossPay << endl;

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

Initializing a Structure - Constructors

- Using an initialization list to initialize a structure has two drawbacks
  - It does not allow you to leave some members uninitialized AND to initialize other members that follow
  - It does not work on some compilers if the structure includes objects, such as strings

- In these cases you can initialize a structure by adding a **constructor** to the structure declaration
  - A **constructor** is a special function (included inside the struct definition) that can be used to construct, or set up and initialize, a structure

Initializing a Structure - Constructors

- A **constructor** looks like a regular function except:
  1. Its name is the same as the structure tag
  2. It has no return type (remember with regular functions a return type must be specified even if that return type is void)
- **Example:**

```
struct Employee
{
    string name;
    int vacationDays;
    int vacDaysUsed;

  Employee()   //constructor  (notice it is inside the struct declaration)
  {
     name = "";
     vacationDays = 10;
     vacDaysUsed = 0;
  }
};
```

- This is a `struct` declaration that includes a constructor
- The purpose of this constructor is to initialize default values for a variable of type Employee at the time the variable is declared
  - IF new employees are allowed 10 vacation days none of which have been taken so far when an employee variable is created this constructor saves the time of having to enter this identical information each time a new employee variable is created.
  - The unique name for the employee can be added later and the vacationDays and vacDays used member variables can be updated as needed

Initializing a Structure - Constructors

- A constructor is not called like a regular function
  - Instead it is automatically executed whenever a variable of that structure type is created

- In the Employee example say we declare the following variables of type **`Employee`**:
  - **`Employee emp1, emp2;`**
  - This statement creates two variables that are **`Employee`** structures.
    - The constructor in the Employee **`struct`** declaration is automatically called when these variables are created
    - So this statement initializes the members of **`emp1`** and **`emp2`**.
      - After this statement:
        - the **`name`** members of both **`emp1`** and **`emp2`** are the empty string
        - the **`vacationDays`** members of both **`emp1`** and **`emp2`** are both 10
        - the **`vacDaysUsed`** members of both **`emp1`** and **`emp2`** are both 0

Initializing a Structure - Constructors

- Sometimes you may not want to have common initialization values for each variable created of a particular structure type
  - You might want to initialize the variables of a structure with unique values when they are created
    - So maybe emp1 initially has 5 vacation days because they are a part time employee but emp2 is a full time employee and should initially have 10 vacation days
  - This is accomplished by passing information into the constructor
    - **Remember a constructor is a special type of function.**
      - It can have parameters that receive arguments (input data) like a regular function does

Initializing a Structure - Constructors

- **Example:** Consider a program that uses a structure called `PopInfo` to hold population info about cities in the state of California
  - Each city in the state has a unique name and population so we would like to initialize the variables of type `PopInfo` when they are created

```
struct PopInfo
{
   string cityName;  //city name
   long population;  //city population

   PopInfo(string n, long p)   //constructor with 2 parameters
   {
      cityName = n;
      population = p;
   }
};
```

  - Using this structure with the constructor that accepts arguments, `PopInfo` type variables can be created an initialized as follows:

```
PopInfo city1("Fremont", 50000);
PopInfo city2("Lancaster", 30302);
```

  - Each variable will be initialized with the data it passes to the constructor when it is created.
  - Remember that constructors are automatically called when the structs are created

Initializing a Structure - Constructors

- **Example 4 – popInfo.cpp**: CA city population program

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

struct PopInfo  //declare struct to hold city name and population info
{
   string cityName;
   long population;

   PopInfo(string n, long p)  //constructor with 2 parameters
   {
      cityName = n;
      population = p;
   }
};

int main()
{

   PopInfo city1("Fremont", 50000);
   PopInfo city2("Lancaster", 30302);
   //PopInfo city3;     //uncomment this to see the error if you don't pass values
                        //to the constructor when you create your structs

   //print data
   cout << city1.cityName << " has a population of " << city1.population << endl;
   cout << city2.cityName << " has a population of " << city2.population << endl;

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

Initializing a Structure - Constructors

- **Example:** Consider a program that uses a structure called PopInfo to hold population info about cities in the state of California
- What happens when you try to define a PopInfo variable without passing values into the constructor
  - Program Error
- **Solution:**
  - Use Default Values in the constructor that can be used in case a variables is created without passing any arguments to the constructor

```
struct PopInfo
{
   string cityName;
   long population;

   PopInfo(string n = "Unnamed", long p = 0)
   {
      cityName = n;
      population = p;
   }
};
```

Initializing a Structure - Constructors

- **Example 5 – popInfo2.cpp** :Default Values in Constructor

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

int main()
{
   struct PopInfo  //declare struct to hold city name and population info
   {
      string cityName;
      long population;

      PopInfo(string n = "Unnamed", long p = 0)  //constructor with 2 parameters &
                                                  //DEFAULT VALUES
      {
         cityName = n;
         population = p;
      }
   };

   PopInfo city1("Fremont", 50000);
   PopInfo city2("Lancaster", 30302);
   PopInfo city3;     //the default constructor values are used

   //print data
   cout << city1.cityName << " has a population of " << city1.population << endl;
   cout << city2.cityName << " has a population of " << city2.population << endl;
   cout << city3.cityName << " has a population of " << city3.population << endl;

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

Initializing a Structure - Constructors

- Notice when we call create **`structs`** without using any arguments there are no parenthesis following the variable declaration
  - **For those of you familiar with classes this is different**
  - **`ILLEGAL:`**
    - `PopInfo city3();`
  - **`LEGAL:`**
    - `PopInfo city3;`
- We will learn more about constructors when we start object oriented programming in Chapter 10

Initializing a Structure – Overloading Constructors

- In our functions lectures we discussed overloaded functions
- Since constructors are special types of functions, instead of using default parameters in our constructor we can use overloaded constructors to allow a user create a new **`PopInfo struct`** instance in different situations.
  - There may be situations where the user knows the population of the city and where the user does not know the name of the city.
  - There may be situations where the user does not know the population or the name of the city but wants to generate a variable of the PopInfo type and fill in the member information later.
  - There may be situations where the user knows the name of the city and where the user does not know the population of of the city.

Initializing a Structure – Overloading Constructors (Example 6 - PopInfoOverloaded.cpp)

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

struct PopInfo  //declare struct to hold city name and population info
{
  string cityName;
  long population;

   PopInfo(string n, long p = 0) //constructor with 2 parameters and one default value
  {
     cityName = n;
     population = p;
  }

  PopInfo(long p)  //construcutor witih one parameter for the population
  {
    cityName = "Unnamed";
    population = p;
  }

  PopInfo()  //constructor with no parameters
  {
    cityName = "Unnamed";
    population = 0;
  }
};

int main()
{
   PopInfo city1("Fremont", 50000); //creates three variables of the PopInfo Data Type
   PopInfo city2("Lancaster", 30302);
   PopInfo city3;     //the default constructor values are used

   cout << city1.cityName << " has a population of " << city1.population << endl;  //display the data
   cout << city2.cityName << " has a population of " << city2.population << endl;
   cout << city3.cityName << " has a population of " << city3.population << endl;

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

Nested Structures

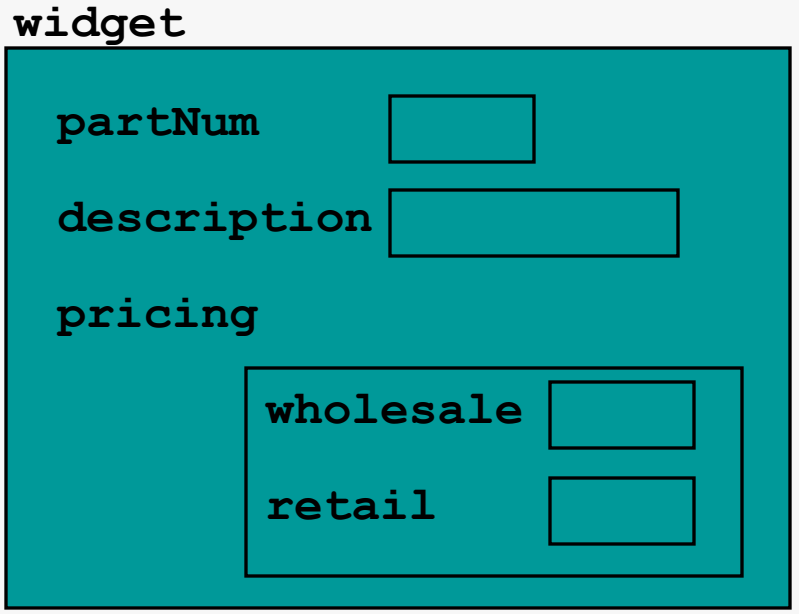
- **Concept:** It is possible for a structure variable to be a member of another structure variable

```
struct Costs
{
   float wholesale;
   float retail;
};

struct Item
{
   string partNum;
   string description;
   Costs pricing;
};
```

- The Costs structure has two members (wholesale and retail)
- The Item structure has three members, the partNum and description are string objects
- The pricing member is a nested Costs structure
  - Pricing is a variable of the struct type Costs
- The following code creates a variable widget of struct type Item

```
Item widget;
```



Nested Structures

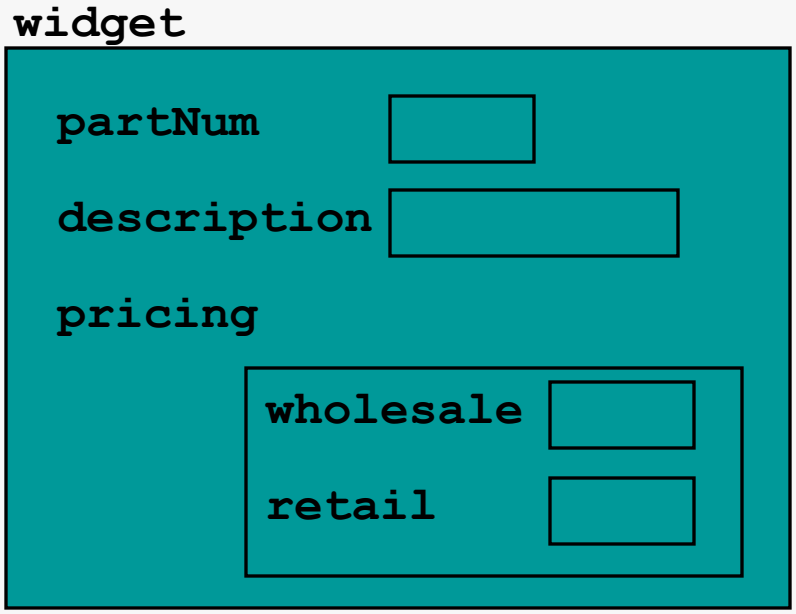
```
struct Costs
{
   float wholesale;
   float retail;
};

struct Item
{
   string partNum;
   string description;
   Costs pricing;
};
Item widget;
```

- The members in widget would be accessed as follows:

```
widget.partNum = "123";
widget.description = "iron";
widget.pricing.wholesale = 100.00;
Widget.pricing.retail = 150.00;
```

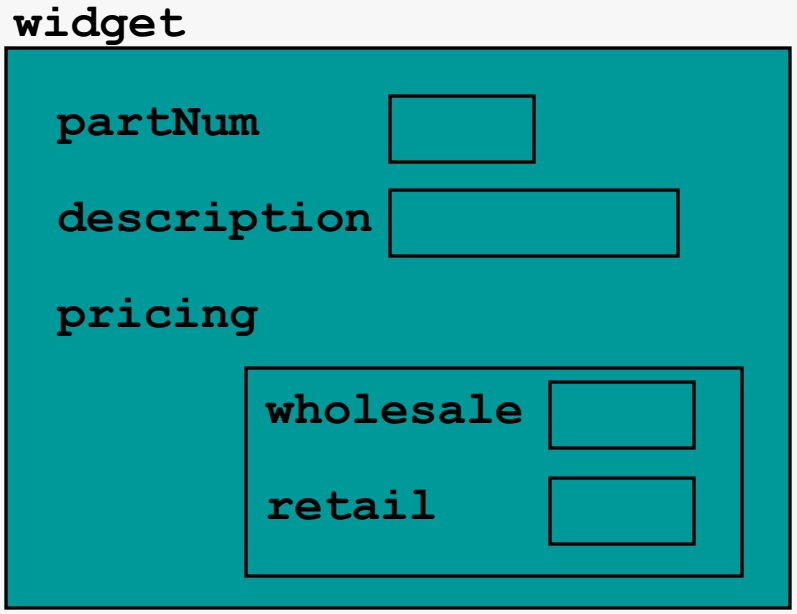
- Notice that wholesale and retail are NOT members of widget, pricing is the member of widget
- So to access retail, widget's pricing variable must first be accessed and then, since Costs is a structure, the wholesale and retail methods can be accessed



Nested Structures

```
struct Costs
{
   float wholesale;
   float retail;
};

struct Item
{
   string partNum;
   string description;
   Costs pricing;
};
Item widget;
```



- The following statements would be ILLEGAL:

```
cout << widget.retail;//illegal
cout << widget.wholesale //illegal
```

- When thinking about whether to use a nested structure or not, think about how the various members of the structure are related
  - **A structure bundles items that logically belong together**

Example 7 – pet.cpp

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

struct AnnualCostInfo
{
   float food;
   float medical;
   float license;
   float misc;
};

struct PetInfo
{
   string name;
   string type;
   int age;
   AnnualCostInfo cost;
};

int main()
{


   PetInfo pet;   //define a structure variable
   pet.name = "Sassy";
   pet.type = "cat";
   pet.age = 5;
   pet.cost.food = 250.00;
   pet.cost.medical = 150.00;
   pet.cost.license = 7.00;
   pet.cost.misc = 50.00;

   cout << setiosflags(ios::fixed) << setprecision(2)
        << "Annual costs for my " << pet.age << " year old "<< pet.type << " " << pet.name << " are $"
        << (pet.cost.food + pet.cost.medical + pet.cost. license + pet.cost.misc) << endl;

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

Arrays of Structures

- Concept: Elements of Arrays can be structures
- Arrays of structures are defined like any other array
- Assume the following structure declaration exists in a program:

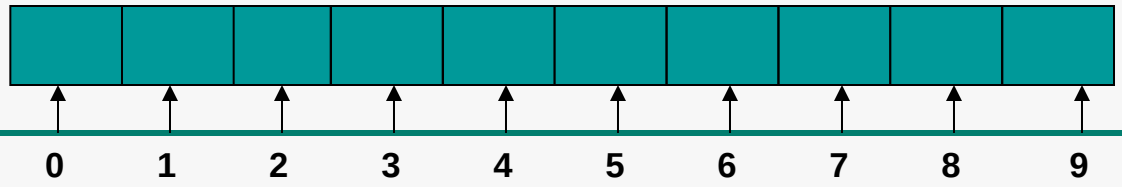
```
struct BookInfo
{
   string title;
   string author;
   string publisher;
   float price;
};
```

- The following statement defines an array **bookList** with has 10 elements
  - Each element is a **BookInfo** structure

```
BookInfo bookList[10];
```

  - Each element in the array contains enough storage to hold one **BookInfo** data type instance

bookList array of type **BookInfo**



Arrays of Structures

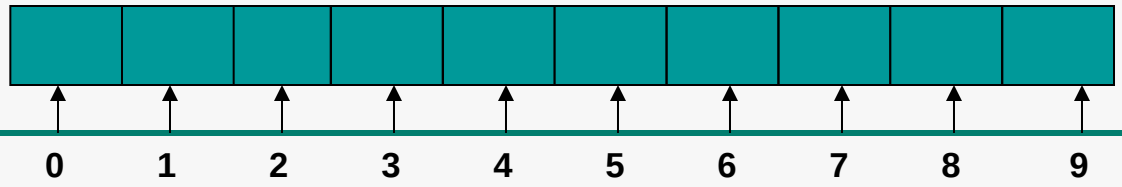
```
struct BookInfo
{
   string title;
   string author;
   string publisher;
   float price;
};
BookInfo bookList[10];
```

- Each element in the array may be accessed through the array subscripts
  - `bookList[0]` is the first structure in the array
  - `bookList[1]` is the second structure in the array

- So how would we assign information to the first booklist object?

```
bookList[0].title = "The Outsiders";
bookList[0].author = "S. E. Hinton";
bookList[0].publisher = "Pearson";
bookList[0].price = 17.50;
```

`bookList` array of type `BookInfo`



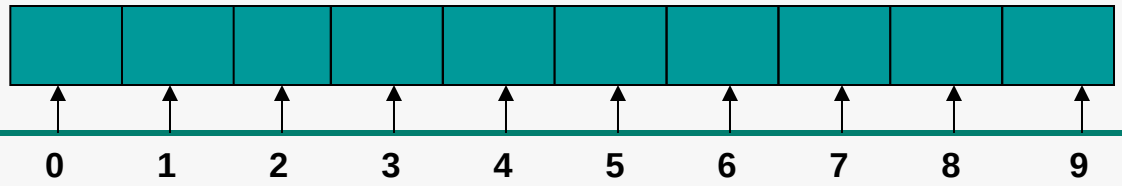
Arrays of Structures

```
struct BookInfo
{
   string title;
   string author;
   string publisher;
   float price;
};
BookInfo bookList[10];
```

- Let's say each element in the bookList had been initialized with some data. The following for loop would print out the information stored in each element

```
for(int index = 0; index < 10; index ++)
{
   cout << bookList[index].title << endl;
   cout << bookList[index].author << endl;
   cout << bookList[index].publisher << endl;
   cout << bookList[index].price << endl;
}
```

`bookList` array of type `BookInfo`



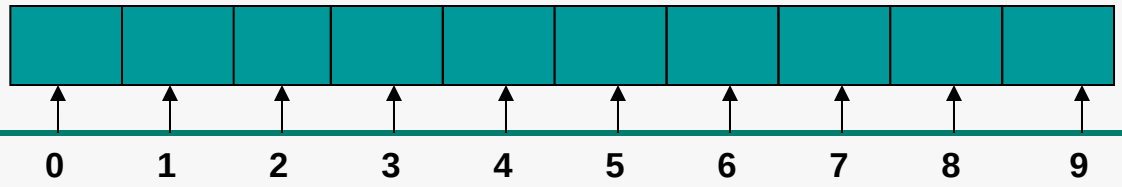
Arrays of Structures

```
struct BookInfo
{
   string title;
   string author;
   string publisher;
   float price;
};
BookInfo bookList[10];
```

- Notice that the title, author, and publisher members of BookInfo are string objects.
  - Since string objects are stored as character arrays you could also access their individual elements

```
bookList[0].title[0];
//accesses the first character of the title of the first
//bookList array element
```

bookList array of type **BookInfo**



Example 8 - payInfoArray

```
#include <iostream>
#include <string>
#include <iomanip>
using namespace std;

struct PayInfo
{
  string empID;
  int hours;
  float payRate;
  float grossPay;
};

int main()
{


   const int numWorkers = 3;

   int index;  //loop counter
   PayInfo workers[numWorkers];  //array of 3 PayInfo structures

   //get Payroll Data
   for(index = 0; index < numWorkers; index++){
      cout << "Enter the employee ID number: ";
      cin >> workers[index].empID;
      cout << "Enter the employee hours: ";
      cin >> workers[index].hours;
      cout << "Enter the employee pay rate ";
      cin >> workers[index].payRate;

      //calculate and input gross pay
      workers[index].grossPay = workers[index].payRate * workers[index].hours;
   }
    cout << '\n';

   //display each employee's gross pay
   for(index = 0; index < numWorkers; index++){
      cout << "Employee #" << workers[index].empID
           << " has gross pay of $" << setiosflags(ios::fixed)
           << setprecision(2) << workers[index].grossPay << endl;
   }
   system("PAUSE");
   return 0;
}
```

Structure as Function Arguments

- **Using Individual Structure Members as Function Arguments**
  - Like other variables of their same type individual members of a structure variable may be used as function arguments of that type

Structure Members as Function Arguments –Ex 9

```
#include <iostream>    //EXAMPLE:  (Example 8 - MembersFuncArg.cpp)
using namespace std;

double multiply(double, double);  //function prototype

struct Rectangle  //creates the struct data type rectangle
{
   double length;
   double width;
   double area;
};

int main()
{
   Rectangle box;   //define a variable of type Rectangle
   box.length = 4.3;
   box.width = 5.4;

   //this function call passes the length and width members of box to the multiply
   //function and stores the return value from the function into the area member of box
   box.area = multiply(box.length, box.width);

   cout << "The area of the box is " << box.area << endl;

   system("PAUSE");
   return 0;

}

//Define a function that has two double parameters
double multiply(double x, double y)
{
   return x * y;
}
```

Entire Structures as Function Arguments

- **Using an Entire Structure as a Function Argument**
  - Sometimes its more convenient to pass an entire structure into a function instead of the individual members
  - The following function uses a Rectangle structure variable as its parameter

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

  - The following function call would pass the entire box variable from the previous program into r

```
showRect(box);
```

Entire Structures as Function Arguments

- **Using an Entire Structure as a Function Argument**

```
showRect(box);

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

Important: ***instances of primitive data types, instances of class data types like strings or anything from Ch 10, and structs in C++ are PASSED BY VALUE unless we indicate otherwise with the & (or later on the pointer)***

- ***This is NOT like Java***

- ***The only type automatically passed by reference is an array***

- Once the function is called **`r.length`** contains a ***copy*** of **`box.length`**, **`r.width`** contains a copy of **`box.width`** and **`r.area`** contains a copy of **`box.area`**
- Structure variables like all variables (except arrays) are passed to functions **by value** as shown here unless we specify otherwise
- The Revised Rectangle program using this function is shown next

Entire Structures as Function Args – Pass by Value Ex

```
#include <iostream>   //EXAMPLE:  (Example 10 -EntireStructFuncArg.cpp)
using namespace std;

//creates the struct data type rectangle
struct Rectangle
{
   double length;
   double width;
   double area;
};

void showRect(Rectangle); //function prototype

int main()
{
   Rectangle box; //define a variable of type Rectangle
   box.length = 4.3;
   box.width = 5.4;
   box.area = box.length * box.width;

   showRect(box);

   system("PAUSE");
   return 0;
}

//Define a function that uses an entire Rectangle structure variable as its parameter
void showRect(Rectangle r)
{
   cout << "The length of box is " << r.length << endl;
   cout << "The width of box is  " << r.width << endl;
   cout << "The area of box is   " << r.area << endl;
}
```

Anything I do to parameter r in the **`showRect`** function will NOT change the Rectangle argument I pass in from main – let's try it

Entire Structures as Function Args – Pass by Reference

- ***Pass by value*** works out ok for a structure if the function does not need to have access (be able to modify) to the members of the original structure argument
  - If a function needs to have access to the members of the original structure argument the structure should be ***passed by reference***
  - Remember passing by reference passes the address of the argument (in this case a structure variable) giving the function access to the actual variable
  - Also think of all the space wasted copying a structure with a large number of members in it

Example 11 – Entire Struct Passed by Reference

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

struct InvItem //creates the struct data type Inv Item
{
   int partNum;
   string description;
   int onHand;  //number of units on hand
   double price;
};
void getItem(InvItem&); //function prototype, passes InvItem by ref to populate struct with data
void showItem(InvItem); //function prototype, passes InvItem by value to print out its data

int main()
{
  InvItem firstPart;
  getItem(firstPart); //function call to pass by reference function
  showItem(firstPart); //function call to pass by value function
  system("PAUSE");
  return 0;
}

void getItem(InvItem& piece)
{
   cout << "Enter the Part Number: ";
   cin >> piece.partNum;
   cout << "Enter the Part Description: ";
   cin.ignore();
   getline(cin, piece.description);
   cout << "Enter the quantity on hand: ";
   cin >> piece.onHand;
   cout << "Enter price per part: ";
   cin >> piece.price;
}

void showItem(InvItem piece)
{
   cout << fixed << setprecision(2) << endl;
   cout << "Part Number:          " << piece.partNum << endl;
   cout << "Part Description:     " << piece.description << endl;
   cout << "Quantity on Hand:     " << piece.onHand << endl;
   cout << "Price Per Part:       " << piece.price << endl;
}
```

Constant Reference Parameters

- Disadvantages of Passing a Structure by Value
  - If a structure has a lot of members passing a structure by value can slow down a program's execution time
    - This is because when an argument is passed by value a copy of it is created
      - For a structure argument passed by value each of its members must passed so a copy of each member must be created

- When a argument is Passed by Reference a copy of it is not created
  - Only a reference that points to the original argument is passed
  - The disadvantage of pass by reference however is that the function has access to the original argument (all of a structure's members in this case) and can alter the argument's value (all member values)

- **Solution: Pass the argument by a *constant reference***

Constant Reference Parameters

- **Solution:** Pass the argument by a ***constant reference***
  - When an argument is passed by ***constant reference*** **this means that only a reference to the original variable is passed to the function BUT the argument cannot be changed by the function**
  - To declare that a function parameter will receive a constant reference the keyword **`const`** must be placed in the parameter list of the function prototype and the function header
  - Revised **`InvItem`** example for **`showItem`** function

```
void showItem(const InvItem&)  //function prototype
void showItem(const InvItem &piece) //function header
```

  - Show Revised **`InvItem`** example

Constant Reference Parameters – (Example 12 - InvItem-PassByRef.cpp)

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

struct InvItem  //creates the struct data type Inv Item
{
   int partNum;
   string description;
   int onHand;  //number of units on hand
   double price;
};
void getItem(InvItem&); //function prototype, passes InvItem by ref to populate struct with data
void showItem(const InvItem&); //function prototype, passes InvItem by constant reference to print out its data

int main()
{
  InvItem firstPart;
  getItem(firstPart);//function call to pass by reference function
  showItem(firstPart);//function call to pass by value function
  system("PAUSE");
  return 0;
}

void getItem(InvItem& piece)
{
   cout << "Enter the Part Number: ";
   cin >> piece.partNum;
   cout << "Enter the Part Description: ";
   cin.ignore();
   getline(cin, piece.description);
   cout << "Enter the quantity on hand: ";
   cin >> piece.onHand;
   cout << "Enter price per part: ";
   cin >> piece.price;
}

void showItem(const InvItem& piece)//pass arg to parameter by const reference
{
   cout << fixed << setprecision(2) << endl;
   cout << "Part Number:          " << piece.partNum << endl;
   cout << "Part Description:     " << piece.description << endl;
   cout << "Quantity on Hand:     " << piece.onHand << endl;
   cout << "Price Per Part:       " << piece.price << endl;
}
```

Returning a Structure from a Function

- **Concept:** A function may return a structure
  - Just as functions can return data types such as int, long, and double they can also return structure data types

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

//a function such as the following could be written to
//return a variable of the circle data type
Circle getData()
{
   Circle temp;
   temp.radius = 10.0;
   temp.diameter = 20.0;
   temp.area = 314.159;
   return temp;
}

//in the calling program, there would have to be a declaration of a
//circle variable, then the results of the getData function could be
//passed to that variable

  Circle piePlate;
  piePlate = getData();
```

Returning a Structure from a Function – Example 13

```
#include <iostream>  //EXAMPLE 12:   (returnCircStruct.cpp)
#include <iomanip>
using namespace std;

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

Circle getData(); //function prototype, returns a circle struct

int main()
{
  Circle piePlate;
  piePlate = getData();

  cout << "PIE PLATE MEMBERS" << endl;
  cout << "Radius: " << piePlate.radius << endl;
  cout << "Diameter: " << piePlate.diameter << endl;
  cout << "Area:  " << piePlate.area << endl;

  system("PAUSE");
  return 0;
}

//returns a variable of the circle data type with information in the members
Circle getData()
{
   Circle temp;
   temp.radius = 10.0;
   temp.diameter = 20.0;
   temp.area = 314.159;
   return temp;
}
```