STRUCTS,
I/O STREAMS, AND
INTRO TO CLASSES

STRUCTS

- User defined structure types (structs) are the steppingstone to understanding classes
- An **object** is a variable that has member functions
- A **class** is a data type whose variables are objects
- A **struct** is an object without any member functions.

STRUCTS

- A **struct** is an object without any member functions
- structs are used for diverse data
  - a collection of values of different types
  - the collection is treated as a single item
  - **Member names** – (fields) identifiers inside the struct.

```
struct CDAccount {
    double balance;
    double interestRate;
    int term; // number of months
};
```

STRUCTS

```
struct CDAccount {
   double balance;
   double interestRate;
   int term; // number of months
};

//declaring struct objects
CDAccount myAccount, yourAccount;
```

Structure values – collection of member values, one member value for each member.

The **dot operator** is used to specify a member variable of a structure variable

```
myAccount.balance //this is a double
```

STRUCTS

```
#include <iostream>

using namespace std;

struct CDAccount {
      double balance;
      double interestRate;
      int term; // number of months
};

void getData(CDAccount& theAccount);
//postcondition: the user has entered the values

int main()
{
      //declaring struct objects
      CDAccount myAccount, yourAccount;
      cout << "\nmy Account input: ";
      getData(myAccount);
      return 0;
}

//get the data from the user by reference
void getData(CDAccount& theAccount)
{
      cout << "\nEnter the account initial balance: ";
      cin >> theAccount.balance;

      cout << "\nEnter the interest rate: ";
      cin >> theAccount.interestRate;

      cout << "\nEnter the term (must be 12 or fewer months): ";
      cin >> theAccount.term;
}
```

STRUCTS AND FUNCTIONS

A function can be call by reference or call by value of a structure type:

```
void GetData(CDAccount& theAccount);
//postcondition: the user has entered the values

double CalculateNewBalance(CDAccount theAccount);
//postcondition: returns the new CD account balance at the end
of the term
```

A function can return a structure type:

```
CDAccount GetData2();
//postcondition: the user has entered the values
```

HIERARCHICAL STRUCTS

Sometimes it makes sense to have structures whose members are smaller structures

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

struct PersonInfo {
    string firstname;
    double height; // in inches
    double weight; // in pounds
    Date birthday; //Date struct
};
```

HIERARCHICAL STRUCTS

Sometimes it makes sense to have structures whose members are smaller structures

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

struct PersonInfo {
      string firstname;
      double height; // in inches
      double weight; // in pounds
      Date birthday; //Date struct
};
```

```
//declaring struct object
PersonInfo person1;

cout << "\nEnter your first name ";
cin >> person1.firstname;//person1 first name

cout << "\nWhat year were you born: ";
cin >> person1.birthday.year; //person 1 birth year
```

When initializing values in a struct it is done in the order of the member variables.

```
PersonInfo person2 = { "Jim", 54.3, 65.2, 10, 15, 2010 };
cout << "\nThe month person 2 was born was " << person2.birthday.month << endl;
```

INTRODUCTION TO CLASSES

Introduction

- C++ offers Object-Oriented Programming (OOP) through classes and objects
- Classes provide encapsulation, abstraction, inheritance, and polymorphism

CLASSES

A class is a data type whose variables are objects.

Object – a variable that has member functions as well as the ability to hold data values.

The object is actually the value of the variable.

- In a struct by default all member variables are Public
- In a class by default all member variables are Private

•**Private members** are only accessible inside the class.

•**Public members** are accessible outside the class

FEATURES OF CLASSES

Encapsulation is the practice of bundling data into a single unit with methods that operate on that data. Data hiding using private and public access specifiers.

Abstraction: Exposing only essential details.

Inheritance: Reusing properties and behaviors of another class. a new class (child) can inherit properties and behaviors of an existing (parent) class

Polymorphism is the ability of a single variable or function to handle different data types or objects. Allowing different implementations through function overloading and overriding.

CLASSES

- Member function - is defined the same way as any other function except that the class name the scope resolution operator `(::)` are given in the function header
- Member functions are implemented outside the class using `::` (scope resolution operator)
- Helps separate interface from implementation.

```
void DayOfYear::Output()

returnType ClassName::FunctionName(formal parameter list)
{
    Function body statements
}
```

CLASSES

```cpp
//class example DayOfYear
#include <iostream>
#include <string>
using namespace std;

class DayOfYear
{
public:
    DayOfYear(); //default constructor
    DayOfYear(int, int); //explicit-value constructor
    void input(); //member function
    void output()const; //member function

private:
    int month;
    int day;
};
```

CLASSES

```cpp
//class example 1 DayOfYear
#include <iostream>
#include <string>
using namespace std;

class DayOfYear
{
public:
	DayOfYear(); //default constructor
	DayOfYear(int, int); //explicit-value constructor
	void input();
	void output()const;

private:
	int month;
	int day;
};
```

```cpp
//member function definitions
//constructor initializes member variables to 0
DayOfYear::DayOfYear() : month(0), day(0)
{
    //body is intentionally left blank
}
//overloaded constructor initializes the member variables with input
//also called an explicit-value constructor
DayOfYear::DayOfYear(int newMonth, int newDay)
{
    month = newMonth;
    day = newDay;
}
```

CLASSES

```
//class example 1 DayOfYear
#include <iostream>
#include <string>
using namespace std;

class DayOfYear
{
public:
	DayOfYear(); //default constructor
	DayOfYear(int, int); //explicit-value constructor
	void input();
	void output()const;

private:
	int month;
	int day;
};
```

```
//gets the date from the user
void DayOfYear::input()
{
    //dot operator is not needed because it is a member function
    cout << "\nEnter the month as a number: ";
    cin >> month;

    cout << "\nEnter the day of the month: ";
    cin >> day;
}
```

CLASSES

```
//class example 1 DayOfYear
#include <iostream>
#include <string>
using namespace std;

class DayOfYear
{
public:
	DayOfYear(); //default constructor
	DayOfYear(int, int); //explicit-value constructor
	void input();
	void output()const;

private:
	int month;
	int day;
};
```

```
//prints the month and day
void DayOfYear::output()const
{
    //dot operator is not needed because it is a member function
    cout << "\nmonth is " << month
        << ", day is " << day << endl;
}
```

CLASSES

```
//class example 1 DayOfYear
#include <iostream>
#include <string>
using namespace std;

class DayOfYear
{
public:
	DayOfYear(); //default constructor
	DayOfYear(int, int); //explicit-value constructor
	void input();
	void output()const;

private:
	int month;
	int day;
};
```

```
//main function driver
int main()
{
	DayOfYear today, birthday;
	cout << "\nEnter todays date: ";
	today.input();
	cout << "\nEnter your birthday: ";
	birthday.input();
	cout << "\nTodays date is: ";
	today.output();
	cout << "\nYour birthday is: ";
	birthday.output();

	return 0;
}
```

CLASSES

Private and public Members
With an ideal class definition, you should be able to change the details of how the class is implemented and the only things that you should need to change in any program that uses the class are the definitions of the member functions.

Member variables should only be accessed through the member functions.
Private members cannot be directly accessed in the program except within the member functions.

The variables that follow the label private will be **private member variables** and the functions will be **private member functions.**

Programmers find that it usually makes their code easier to understand and easier to update if they make all member variables private.

Member functions set and get are used to access the member variables:

STREAMS

- A **class** is a data type whose variables are objects
- A **struct** is an object without any member functions
- **Object** – a special kind of variable that has its own special-purpose functions
- **Stream** – A flow of data.

STREAMS

- **Object** – a special kind of variable that has its own special-purpose functions
- **Stream** – A flow of data.

```
#include <iostream> //name of the header
using namespace std; //for cin and cout
```

cin is an input stream connected to the keyboard
cout is an output stream connected to the screen

You can define other streams that come and go from files

```
#include <fstream> //name of the header file for file i/o
using namespace std; //for ifstream and ofstream
```

#include <fstream>

- **object** – A variable that has functions as well as data associated with it
- **class** – a type whose variables are objects
- **member function** – a function that is associated with an object

```
//function call
calling_object.member_function_name(argument list);
```

```
using namespace std; //for ifstream and ofstream
ifstream inStream, in; //two input stream objects
ofstream outStream, out; //two output stream objects

//connecting to the input file:
inStream.open("inputData.txt");
in.open("inFile.txt")

//connecting to an output file:
outStream.open("outputData.txt");
out.open("outFile.txt");
```

CLASSES AND OBJECTS

```
//function call
calling_object.member_function_name(argument list);
```

```
ofstream outStream;
char filename[16];

//user should enter filename with .txt extension
cout << "Enter the filename: ";
cin >> filename;
//connecting to an output file:
//use app if you want to append data to the file
outStream.open(filename, ios::app);

//magic formula
//setf - set flags
outStream.setf(ios::fixed); //fixed – usual way, i.e not e-notation
outStream.setf(ios::showpoint); //show the decimal point
outStream.precision(2);
```