
# Lecture 4

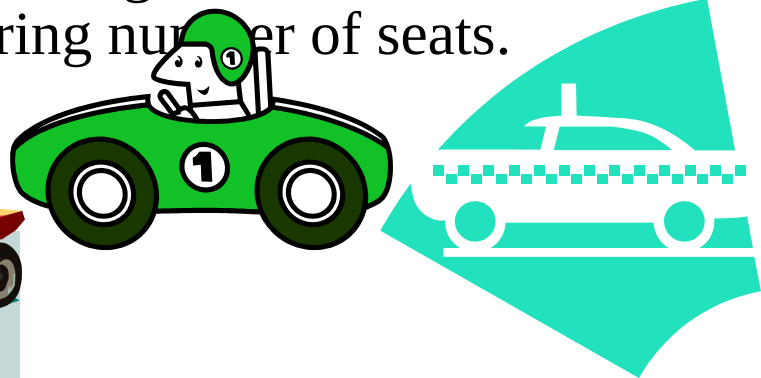
## CSE202: OBJECT ORIENTED PROGRAMMING

# Outline

- Structure
- Union
- Enumeration and Classes

# Why Use Structures?

- Quite often we deal with entities that are collection of dissimilar data types.
- For example, suppose you want to store data about a **car**. You might want to store its name (a string), its price (a float) and number of seats in it (an int).
- If data about say 3 such cars is to be stored, then we can follow two approaches:
  - Construct individual arrays, one for storing names, another for storing prices and still another for storing number of seats.
  - Use a structure variable.



# Introduction

- Structures are **user defined data types**
- Structures
  - Structure is a group of data items of different data types held together in a single unit.
  - Collections of related variables under **one name**
    - Can contain variables of different data types
  - Commonly used to define records to be stored in files.

# Structure

- There are three aspects of working with structures:
  - Defining a structure type
  - Declaring variables and constants of newly created type
  - Using and performing operations on the objects of structure type

# Structure Definition

**Syntax**

```
struct  sname {
        type var1;
        type var2;
        type var3;
                .
                .
        type varN;
};
```

**struct** is a keyword to define a structure.
**sname** is the name given to the structure/structure tag.
**type** is a built-in data type.
**var1,var2,var3,…..,varN** are elements of structure being defined.
**;** semicolon at the end.

# Structure Definitions

- Example:

```
struct car{
    char n[20];
    int seats;
    float price
 };
```

  - `struct` **keyword** introduces the definition for structure `car`
  - `car` is the structure name or tag and is used to declare variables of the structure type
  - `car` contains three members of type `char, float, int`
    - These members are `name, price` and `seats`.

- No variable has been associated with this structure
- No memory is set aside for this structure.

# Structure Definitions

- `struct` information
  - A structure definition does not reserve space in memory .
    - Instead creates a new data type used to define structure variables
- Defining **variables** of structure type
  - Defined like other variables:

```
Car myCar1,mycar2…
```

  - Can use a comma separated list along with structure definition:

```
struct car{
      char n[20];
      int seats;
      float price;
  } myCar;
```

At this point, the memory is set aside for the structure variable myCar.

# MCQ

Structure is a data type  in which

A  elements must have same data types

B  elements may have different data types

C  elements must have different data types

D  none of these

# Solution

Structure is a data type in which

A elements must have same data types

B **elements may have different data types**

C elements must have different data types

D none of these

# MCQ

A structure is a collection of

A homogenous e elements

B heterogenous elements

C homogenous elements and heterogenous elements

# Solution

A structure is a collection of

A homogenous e elements

B **heterogenous elements**

C homogenous elements and heterogenous elements

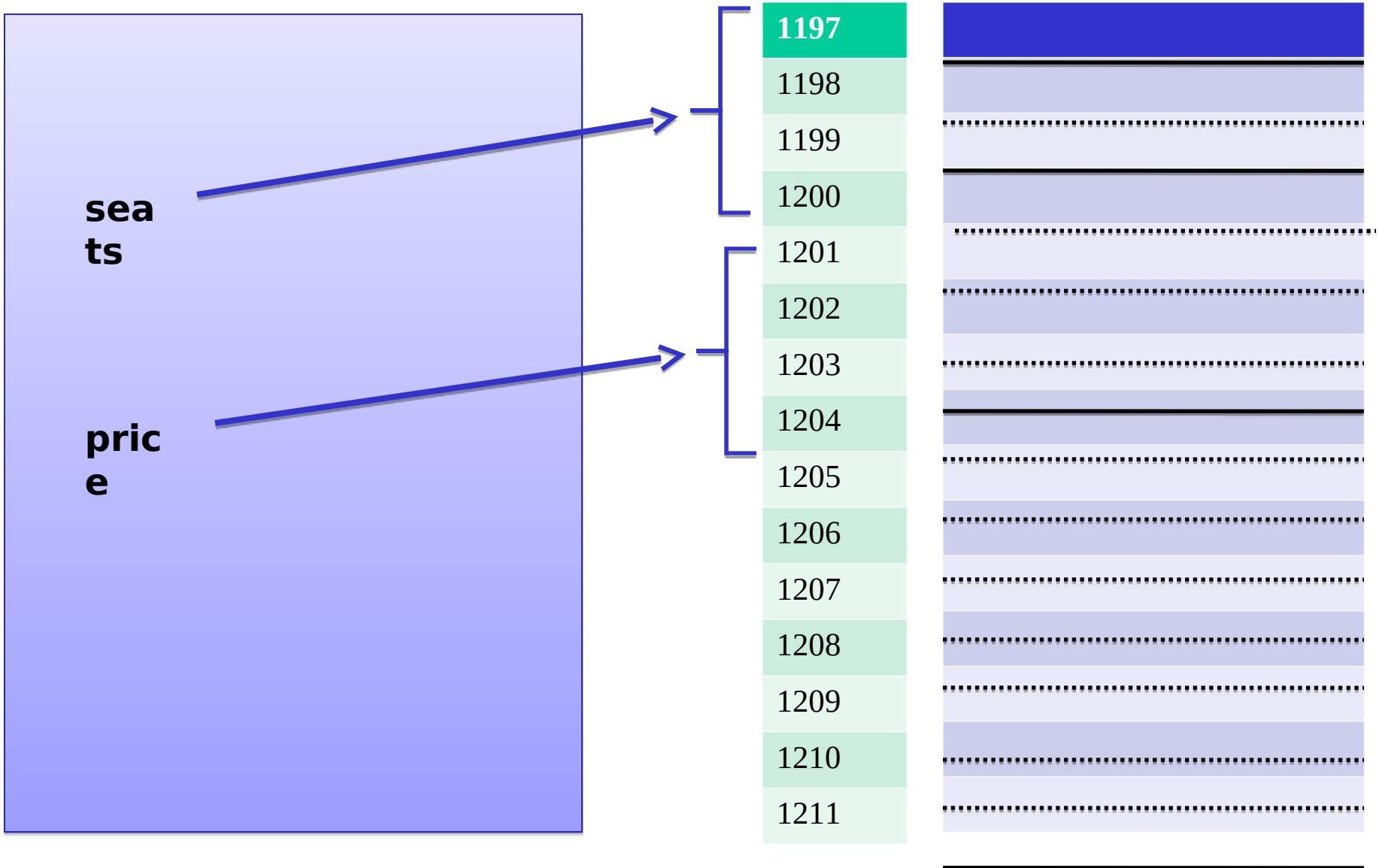
# How the members are stored in memory

Consider the declarations to understand how the members of the **structure variables are stored in memory**

```
struct car{
    int seats;
   float price;
}myCar,
```

**Note:** **all members are stored in contiguous memory location in order in which they are declared.**

# How the members of the structure variables are stored in memory



# Program example of structure

```
#include <iostream>
using namespace std;
struct employee
{
   char name[50];
   int age;
   float salary;
};

int main()
{
   employee e1;
   cout << "Enter Full name: ";
   cin.getline(e1.name, 50);
   cout << "Enter age: ";
   cin >> e1.age;
   cout << "Enter salary: ";
   cin >> e1.salary;
   cout << "\nDisplaying Information." << endl;
   cout << "Name: " << e1.name << endl;
   cout <<"Age: " << e1.age << endl;
   cout << "Salary: " << e1.salary;
   return 0;
}
```

# Union

- Union is similar as structure. The major distinction between them is in terms of storage.
- In structure each member has its own storage location whereas all the members of union uses the same location.
- The union may contain many members of different data type but it can handle only one member at a time union can be declared using the keyword union.

# Example

- A class is a very good example of structure and union in this example students are sitting in contiguous memory allocation as they are treated as a structure individually. And if we are taking the place of teacher then in a class only one teacher can teach. After leaving the first teacher then another teacher can enter.

# Union Declaration

```
union item
{
 int m;
 float x;
 char c;
}code;
```

This declare a variable code of type union item

# Unions

- Valid union operations
  - Assignment to union of same type: =
  - Taking address: &
  - Accessing union members: .
  - Accessing members using pointers: ->

# Program example of Union

```
#include <iostream>

union Data {

    int intValue;

    float floatValue;

    char charValue;

};

int main() {

    Data myData;

myData.intValue = 42;

std::cout << "Integer value: " <<
myData.intValue << std::endl;

myData.floatValue = 3.14f;

    std::cout << "Float value: " <<
myData.floatValue << std::endl;

myData.charValue = 'A';

    std::cout << "Char value: " <<
myData.charValue << std::endl;

return 0;

}
```

# MCQ

Which statement is true in case of memory allocation of members of union

A Memory is allocated for all variables.

B Allocates memory for variable which variable require more memory.

C Allocates memory for variable which variable require less memory.

D none of these

# Solution

Which statement is true in case of memory allocation of members of union

A Memory is allocated for all variables.

B **Allocates memory for variable which variable require more memory.**

C Allocates memory for variable which variable require less memory.

D none of these

# MCQ

which is true in case of union

A require more memory space than Structure

B Declared with Struct Keyword

C require less memory space than Structure

D require more execution time than Structure

# Solution

which is true in case of union

A require more memory space than Structure

B Declared with Struct Keyword

C **require less memory space than Structure**

D require more execution time than Structure

# Difference between Structure and Union

| Structure | Union |
|---|---|
| Struct keyword is used to define a structure. | Union keyword is used to define a union. |
| Members do not share memory in a structure. | Members share the memory space in a union. |
| Any member can be retrieved at any time in a structure. | Only one member can be accessed at a time in a union. |
| Several members of a structure can be initialized at once. | Only the first member can be initialized. |
| Size of the structure is equal to the sum of size of the each member. | Size of the union is equal to the size of the largest member. |
| Altering value of one member will not affect the value of another. | Change in value of one member will affect other member values. |
| Stores different values for all the members. | Stores same value for all the members. |

| | Class | Structure |
|---|---|---|
| Definition | A class in C++ can be defined as a collection of related variables and functions encapsulated in a single structure. | A structure can be referred to as a user defined data type possessing its own operations. |
| Keyword for the declaration | Class | Struct |
| Default access specifier | Private | Public |
| Example | class myclass<br>{<br>private:<br>int data;<br>public:<br>myclass(int data_):<br>data(data_)<br>{}<br>virtual void foo()=0;<br>virtual ~class()<br>{}<br>}; | struct myclass<br>{<br>private:<br>int data;<br>public:<br>myclass(int data_):<br>data(data_)<br>{}<br>virtual void foo()=0;<br>virtual ~class()<br>{}<br>}; |

# MCQ

C structure differs from CPP class in regards that by default all the members of the structure are _____ in nature.

**a.** private

**b.** protected

**c.** public

**d.** None of these

## Solution

C structure differs from CPP class in regards that by default all the members of the structure are _____ in nature.

**a.** private

**b.** protected

**c. public**

**d.** None of these

# Difference between C structures and C++ structures

- **Member functions inside structure**: Structures in C cannot have member functions inside structure but Structures in C++ can have member functions along with data members.
- **Direct Initialization:** We cannot directly initialize structure data members in C but we can do it in C++.

```
#include <iostream>
using namespace std;
struct Record {
        int x = 7;
};
int main()
{
        Record s;
        cout << s.x << endl;
        return 0;
}
```

- **Using struct keyword:** In C, we need to use struct to declare a struct variable. In C++, struct is not necessary. For example, let there be a structure for Record. In C, we must use "struct Record" for Record variables. In C++, we need not use struct and using 'Record' only would work.
- **Static Members:** C structures cannot have static members but is allowed in C++.

```
struct Record {
        static int x;
};
int main()
{
        return 0;
}
```

**Note: This will generate an error in C but no error in C++.**

- **Constructor creation in structure**: Structures in C cannot have constructor inside structure but Structures in C++ can have Constructor creation.
- **sizeof operator:** This operator will generate 0 for an empty structure in C whereas 1 for an empty structure in C++.
- **Data Hiding:** C structures do not allow concept of Data hiding but is permitted in C++ as C++ is an object oriented language whereas C is not.
- **Access Modifiers:** C structures do not have access modifiers as these modifiers are not supported by the language. C++ structures can have this concept as it is inbuilt in the language.

# Difference between Structure in C and Structure in C++

| C Structures | C++ Structures |
|---|---|
| Only data members are allowed, it cannot have member functions. | Can hold both: member functions and data members. |
| Cannot have static members. | Can have static members. |
| Cannot have a constructor inside a structure. | Constructor creation is allowed. |
| Direct Initialization of data members is not possible. | Direct Initialization of data members is possible. |
| Writing the 'struct' keyword is necessary to declare structure-type variables. | Writing the 'struct' keyword is not necessary to declare structure-type variables. |
| Do not have access modifiers. | Supports access modifiers. |
| Only pointers to structs are allowed. | Can have both pointers and references to the struct. |

# Array of structure

The arrays and structures can be combined to form complex data objects. There may be structures contained within an array; also, there may be an array as an element of a structure.

Since an array can contain similar elements, the combination of structures within an array is an array of structures. To declare an array of structures, you must first define a structure and then declare an array variable of that type. For example, to store the addresses of 100 members of the council, you need to create an array.

Now, to declare a 100-element array of structures of type **addr,** we will write:

**Syntax:- addr mem_addr[100];**

# Example of Array of Structure

```
#include<iostream>

using namespace std;

struct emp // structure name (emp)

{

string name; // structure member

};

int main()

{

emp e[5]; // array-of-structure variable (e[5])

int i;

 for(i=0; i<5; i++)

{

cout<<"Enter the name of the employee no."<<i+1<<": ";

cin>>e[i].name;

}

cout<<"\nNames of all employees: \n";

for(i=0; i<5; i++)

{

cout<<e[i].name<<endl; } cout<<endl; return 0;

}
```

# Example of Array of Union

```
#include <iostream>
using namespace std;
// Define a union to hold different data types
union Data {
    int intValue;
    char charValue;
    double doubleValue;
};
        int main() {
    const int SIZE = 5;
    Data dataArray[SIZE]; // Array of unions to store different data types
// Populate the array with data
    dataArray[0].intValue = 42;
    dataArray[1].charValue = 'A';
    dataArray[2].doubleValue = 3.14;
    dataArray[3].intValue = 100;
    dataArray[4].charValue = 'Z';

    // Access and display the data from the array
    cout << "Array of Union:" << endl;
    for (int i = 0; i < SIZE; ++i)
```

```
{
    cout << "Index " << i << ": ";
// Since a union can only hold one value at a time, we need to know the type stored at each index
        if (i % 3 == 0) {
            cout << "Integer Value: " << dataArray[i].intValue << endl;
        }
else if (i % 3 == 1) {
            cout << "Character Value: " << dataArray[i].charValue << endl;
        }
else {
            cout << "Double Value: " << dataArray[i].doubleValue << endl;
        }
    }

    return 0;
}
```

# Enumeration (ENUM)

An enumeration is a user-defined type consisting of a set of named constants called enumerators

- It is a user-defined data type which can be assigned some limited values. These values are defined by the programmer at the time of declaring the enumerated type.
- The values that we specify for the data type must be **legal identifiers**
- The syntax for declaring an enumeration type is:

**enum typeName{value1, value2, ...};**

The syntax to declare an enum is as follows:

```
enum model_name
 {
  value1,
  value2,
  value3,
  . .
};
```

# Declaration of Enumerated Types

- Consider the colors of the rainbow as an enumerated type:

  **enum rainbowColors{ red, orange, yellow, green, blue, indigo, violate };**

- The identifiers between { } are called **enumerators**

- The order of the declaration is significant
  **red < orange < yellow**

- By default, the first enumerator has a **value of 0**
- Each successive enumerator is one larger than the value of the previous one, unless you explicitly specify a value for a particular enumerator

# Declaration of Enumerated Types

- Why are the following illegal declarations?

**enum grades{'A', 'B', 'C', 'D', 'F'};**

**enum places{1st, 2nd, 3rd, 4th};**

They do not have legal identifiers in the list..

- What could you do to make them legal?

**enum grades{A, B, C, D, F};**

**enum places{first, second, third, fourth};**

- As with the declaration of any object
  - Specify the type name
  - Followed by the objects of that type

Given:

**enum daysOfWeek { Sun, Mon, Tue,Wed, Thu, Fri, Sat }**

Then we declare:

**daysOfWeek Today, payDay, dayOff;**

# Assignment with Enumerated Types

- Once an enumerated variable has been declared
  - It may be assigned an enumerated value
  - Assignment statement works as expected

```
payDay = Fri;   // note: no quotes
// Fri is a value, a constant
```

- **Enumerated variables may receive only values of that enumerated type**

# Example 1:

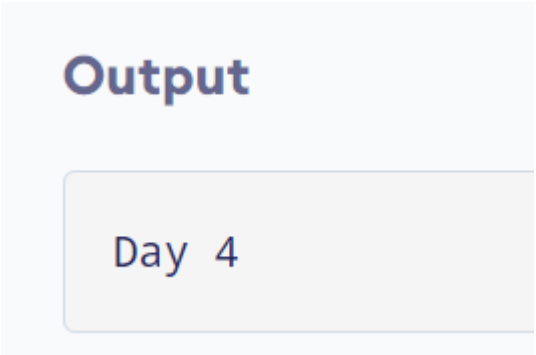
```
main()
{
enum days{sun,mon,tues,wed,thur,fri,sat};
days day1,day2;
day1=sun;
day2=fri;
cout<<day1<<“\t”<<day2;
if(day1>day2)
{
cout<<“day1 comes after day2”;
}
else
{
cout<<“day1 comes before day2”;
}
}
```

# Example 2:

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

enum week { Sunday, Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday };

int main()
{
    week today;
    today = Wednesday;
    cout << "Day " << today+1;
    return 0;
}
```



# Example 3: Changing Default Value of Enums

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

enum seasons { spring = 34, summer = 4, autumn = 9, winter = 32};

int main() {

    seasons s;

    s = summer;
    cout << "Summer = " << s << endl;

    return 0;
}
```

**Output**

```
Summer = 4
```

**Find the output of below program**

```
#include<iostream>
using namespace std;
enum color{
        Black,
        blue,
        red
};
int main()
{
    color obj = blue;
    cout<<obj;
    return 0;
}
```

(A) blue
(B) 2
(C) 1
(D) None of these

**Find the output of below program**

```
#include<iostream>
using namespace std;
enum color{
      Black,
      blue,
      red
};
int main()
{
   color obj = blue;
   cout<<obj;
   return 0;
}
```

(A) blue
(B) 2
(C) **1**
(D) None of these

**Find Output:**

```
#include<iostream>
using namespace std;
enum color{
      black=1,
      blue,
      red
};
int main()
{
   color obj = blue;
   cout<<obj;
      return 0;
}
```

(A) blue
(B) Compilation Error
(C) 1
(D) 2

**Find Output:**

```
#include<iostream>
using namespace std;
enum color{
      black=1,
      blue,
      red
};
int main()
{
   color obj = blue;
   cout<<obj;
      return 0;
}
```

(A) blue
(B) Compilation Error
(C) 1
(D) **2**

**Find Output:**

```
#include<iostream>
using namespace std;
enum color{
      black=1,
      blue,
      red
};
int main()
{
   color obj =yellow;
   cout<<obj;
      return 0;
}
```

(A) yellow
(B) Compilation Error
(C) 1
(D) 2

**Find Output:**

```
#include<iostream>
using namespace std;
enum color{
      black=1,
      blue,
      red
};
int main()
{
   color obj =yellow;
   cout<<obj;
      return 0;
}
```

(A) yellow
(B) **Compilation Error**
(C) 1
(D) 2

# Using Enum inside the class

```
#include <iostream>

using namespace std;

class Year {

public:

    enum Month {

        JANUARY, FEBRUARY, MARCH, APRIL,
MAY, JUNE,

        JULY, AUGUST, SEPTEMBER, OCTOBER,
NOVEMBER, DECEMBER

    };


    void setMonth(Month month) {

        currentMonth = month;

    }
```

```
    Month getMonth() {

        return currentMonth;

    }

string getMonthString(Month month) {

        switch (month) {

            case JANUARY: return "JANUARY";

            case FEBRUARY: return "FEBRUARY";

            case MARCH: return "MARCH";

            case APRIL: return "APRIL";

            case MAY: return "MAY";

            case JUNE: return "JUNE";

            case JULY: return "JULY";

            case AUGUST: return "AUGUST";
```

```
case SEPTEMBER: return "SEPTEMBER";
        case OCTOBER: return "OCTOBER";
        case NOVEMBER: return "NOVEMBER";
        case DECEMBER: return "DECEMBER";
        default: return "Invalid month";
      }
   }
private:
   Month currentMonth;
};
```

```
int main() {
    Year year;
    int month;
    cin >> month;
 if (month >= 1 && month <= 12) {
        // Subtracting 1 from the input to match the enum values
        Year::Month selectedMonth = static_cast<Year::Month>(month - 1);
        year.setMonth(selectedMonth);
```

```
cout << "Month: " <<
year.getMonthString(year.getMonth()) << endl;

    } else {

        cout << "Invalid month input" << endl;

    }


    return 0;

}
```

Thank you!

Any doubts?