

**Batch: A2**

**Roll No.: 16010123032**

**Experiment / assignment / tutorial No.3**

**Grade: AA / AB / BB / BC / CC / CD / DD**

**Signature of the Staff In-charge with date**

**TITLE : Implementing a billing application using OOP concepts using C++**

**AIM:** Develop a C++ application that generates an Electricity Bill using a Consumer class.

---

**Expected OUTCOME of Experiment:**

CO1:Apply the features of object oriented programming languages. (C++ and Java)

CO2:Explore arrays, vectors, classes and objects in C++ and Java

---

**Books/ Journals/ Websites referred:**

1. E. Balagurusamy, "Programming with Java", McGraw-Hill.
2. E. Balagurusamy, "Object Oriented Programming with C++", McGraw-Hill.

---

**Pre Lab/ Prior Concepts:**

Class Definition:

The Consumer class should encapsulate the following information:

- ❖ consumer\_no (integer): Unique identification number for the consumer.
- ❖ consumer\_name (string): Name of the consumer.
- ❖ previous\_reading (integer): Meter reading from the previous month.
- ❖ current\_reading (integer): Meter reading from the current month.
- ❖ connection\_type (string): Type of electricity connection (domestic or commercial).
- ❖ calculate\_bill (member function): This function should calculate the electricity bill amount based on the connection\_type and the number of units consumed

(current reading - previous reading). The function should utilize a tiered pricing structure as specified below:

**Tiered Pricing:**

***Domestic Connection:***

First 100 units: Rs. 1 per unit  
101-200 units: Rs. 2.50 per unit  
201-500 units: Rs. 4 per unit  
Above 501 units: Rs. 6 per unit

***Commercial Connection:***

First 100 units: Rs. 2 per unit  
101-200 units: Rs. 4.50 per unit  
201-500 units: Rs. 6 per unit  
Above 501 units: Rs. 7 per unit

Additional Considerations:

- ❖ The application should prompt the user to enter the details for a consumer (consumer number, name, previous reading, current reading, and connection type).
- ❖ The calculate\_bill function should implement logic to determine the applicable unit charges based on the connection type and the number of units consumed within each tier.
- ❖ The application should display a clear breakdown of the bill, including the consumer details, number of units consumed, charge per unit for each tier, and the total bill amount.

**Algorithm:**

- **Start**
  
- **Class Definition:**
- **Define class Consumer with private members: consumer\_no, consumer\_name, previous\_reading, current\_reading, connection\_type, rate, and total.**
  
- **Constructor:**

- Prompt the user for input and store values in class members.
- Calculate Bill:
- Compute units = current\_reading - previous\_reading.
- For 'd' (domestic) and 'c' (commercial) connection types:
- Use a switch-case structure to calculate the total based on unit ranges and respective rates.
- Display Bill:
- Print consumer\_no, consumer\_name, previous\_reading, current\_reading, connection\_type, and total.
- Main Function:
- Create an object of Consumer, call calculate\_bill(), and then call show\_bill().
- End

**Implementation details:**

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Consumer
```

```
{
```

```
    int consumer_no;
```

```
string consumer_name;  
  
int previous_reading;  
  
int current_reading;  
  
char connection_type;  
  
double rate;  
  
double total;  
  
public:  
  
Consumer()  
  
{  
  
    cout << "Enter consumer number: ";  
  
    cin >> consumer_no;  
  
    cout << endl;  
  
    cout << "Enter consumer name: ";  
  
    cin >> consumer_name;  
  
    cout << endl;  
  
    cout << "Enter previous reading: ";  
  
    cin >> previous_reading;  
  
    cout << endl;  
  
    cout << "Enter current reading: ";
```

```
cin >> current_reading;

cout << endl;

cout << "Enter connection type: ";

cin >> connection_type;

cout << endl;

}

void calculate_bill()

{

    int units = current_reading-previous_reading;

    if(connection_type == 'd')

    {

        switch(units/100)

        {

            case 0:

                total = units*1.0;

                break;

            case 1:

                total = 100 *1.0+(units-100)*2.5;

                break;
```

```
case 2:

case 3:

case 4:

    total = 100*1.0 + 100*2.5 + (units-200)*4.0;

    break;

case 5:

case 6:

case 7:

    total = 100*1.0 + 100*2.5 + 300*4.0 + (units-500)*6.0;

    break;

default:

    cout << "Invalid units: "<<endl;

}

}

else if(connection_type == 'c')

{

    switch(units/100)
```

```
{  
  
case 0:  
  
    total = units*2.0;  
  
    break;  
  
case 1:  
  
    total = 100 *2.0+(units-100)*4.5;  
  
    break;  
  
case 2:  
  
case 3:  
  
case 4:  
  
    total = 100*2.0 + 100*4.5 + (units-200)*6.0;  
  
    break;  
  
case 5:  
  
case 6:  
  
case 7:  
  
    total = 100*2.0 + 100*4.5 + 300*6.0 + (units-500)*7.0;  
  
    break;  
  
default:  
  
    cout << "Invalid units: "<<endl;
```

```
    }  
  
    }  
  
}  
  
void show_bill()  
{  
  
    cout << "-----" << endl;  
  
    cout << "TOTAL BILL : " << endl;  
  
    cout << "-----" << endl;  
  
    cout << "consumer no: " << consumer_no << endl;  
  
    cout << "consumer name: " << consumer_name << endl;  
  
    cout << "previous reading: " << previous_reading << endl;  
  
    cout << "current reading: " << current_reading << endl;  
  
    cout << "connection type: " << connection_type << endl;  
  
    cout << "Total = " << total << endl;  
  
}
```



```
};
```

```
int main()
```

```
{
```

```
    Consumer c1;
```

```
    c1.calculate_bill();
```

```
    c1.show_bill();
```

```
    return 0;
```

```
}
```

### **Output:**

```
Enter consumer number: 123
Enter consumer name: Aksh
Enter previous reading: 5000
Enter current reading: 5300
Enter connection type: d

-----
TOTAL BILL :
-----
consumer no: 123
consumer name: Aksh
previous reading: 5000
current reading: 5300
connection type: d
Total = 750
```

### **Conclusion:**

Developed a C++ application that generates an Electricity Bill using a Consumer class.

**Date:** \_\_\_\_\_

**Signature of faculty in-charge**

**Post Lab Descriptive Questions:**

Q.1 Explain the concept of constructors and destructors in C++.

**Constructors in C++:**

- **Purpose:** Constructors are special member functions in a class that are automatically called when an object of that class is created. They are used to initialize the object's data members and allocate resources if needed.
- **Characteristics:**
  - **Same Name as Class:** A constructor has the same name as the class.
  - **No Return Type:** Constructors do not have a return type, not even `void`
  - **Overloading:** Multiple constructors can be defined in a class, each differing by the number or type of parameters (constructor overloading).
  - **Types of Constructors:**
    - **Default Constructor:** Takes no arguments and initializes objects with default values.
    - **Parameterized Constructor:** Takes arguments and allows the initialization of objects with specific values.
    - **Copy Constructor:** Initializes an object using another object of the same class.
- **Implicit and Explicit Calls:** Constructors can be called implicitly when an object is created or explicitly by the programmer.

## Destructors in C++:

- **Purpose:** Destructors are special member functions in a class that are automatically called when an object goes out of scope or is explicitly deleted. They are used to clean up resources that the object may have acquired during its lifetime (e.g., memory deallocation).
- **Characteristics:**
  - **Same Name as Class with a Tilde (~):** A destructor has the same name as the class, but it is preceded by a tilde (~).
  - **No Return Type and No Parameters:** Destructors do not return a value and cannot take parameters. There is only one destructor per class.
  - **Automatic Invocation:** The destructor is invoked automatically when an object is destroyed (e.g., when it goes out of scope or is deleted).
  - **Non-Overloadable:** Unlike constructors, destructors cannot be overloaded.

## Common Usage:

- **Constructors:** Used to set initial states, allocate memory, or establish necessary conditions for an object to be used.
- **Destructors:** Used to release resources, close files, or perform any necessary cleanup when an object is no longer needed.

In summary, constructors prepare an object for use, and destructors ensure that any resources the object used are properly released when the object is destroyed.

Q.2 Write the output of following program with suitable explanation

```
#include<iostream>
```

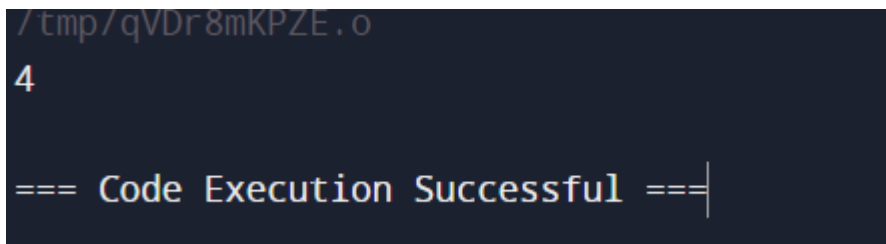
```
using namespace std;
```

```
class Test
{
    static int i;
    int j;
};

int Test::i;

int main()
{
    cout << sizeof(Test);
    return 0;
}
```

**Output:**



```
/tmp/qVDr8mKPZE.o
4
=== Code Execution Successful ===
```

Q.3 Explain all the applications of the scope resolution operator in C++.

**Accessing Global Variables and Functions:** Used to access global variables/functions when there is a local variable/function with the same name.

**Defining a Function Outside the Class:** Used to define a member function of a class outside its class definition.

**Accessing Static Members of a Class:** Used to access static member variables and static member functions of a class.

**Resolving Ambiguity Between Classes:** Used to distinguish between members of different classes that have the same name.

**Accessing Members of a Namespace:** Used to access variables, functions, and types defined within a namespace.

**Accessing Nested Classes or Types:** Used to access classes or types that are nested within another class.

**In Multiple Inheritance:** Used to resolve ambiguity when multiple base classes have members with the same name.