



Department of Electronic Engineering

Software Development 2

Year 4 Semester 2

Laptop Sales Record System Assignment Report

Student Name: Carl Sagrado

Student No: X00084403

Lecturer: Dr. Andrew Donnellan

Date of Submission: 30/04/2021

Declaration of Originality

I hereby certify that this report is entirely the work of the author and that any resources used for the completion of the work done are fully referenced within the text of this report.

Signature:

Date: 30/04/2021

Table of Contents

Assignment Report.....	1
Project Description	2
Class Diagram.....	3
NAVIGATION FLOW.....	4
Technical Project Structure – Documentation.....	5
[1] Basic Class with access member functions, Constructors and static member data	5
[2] Inheritance and Polymorphism	6
[3] Abstract Base Class and Concrete Derived Class.....	9
[4] Queue linked list	10
[5] Operator overloading, as a member function.....	13
[6] Operator overloading, as a nonmember function	14
[7] Composition	15
[8] Basic Association	17
[9] Qualified Association	18
[10] Association Class	20
[10] Dependency	25
[11] An interesting fact about the Project	27
Conclusion	29

Project Description

Record System App is an application which was developed to offer a record system platform for laptop suppliers to record transactions and item warranties for future reference. The application has an 'input and store' capabilities where all records of transactions as well as information of what was being sold are recorded and stored in a linked list, readily accessible for any member of staff to access and review.

Key Features:

Add New Transaction – allow users to record transaction with relevant information such as the following:

- Laptop Details
- Customer and Staff Details
- Purchase date
- Item Warranty

When selected, the system will prompt the user to select and enter relevant information pertaining to the current transaction.

View All Transaction – this feature allows users to all the recorded transaction to date.

Search Transaction– provides search capabilities which allow users to perform search operations by the following:

- Display all transaction references.
- Search a particular transaction using a transaction reference.

When this section of the menu is selected, the system will prompt the user to select from which operation listed above the user wishes to perform.

Search Warranty– provides search capabilities which allow users to perform search operations by the following:

- Display all warranty references.
- Search a particular warranty item using a warranty reference.

When this section of the menu is selected, the system will prompt the user to select from which operation listed above the user wishes to perform.

All key features mentioned are navigated via the main menu.

When a feature is selected, it will automatically execute the necessary functions to perform desired actions. For every successful operation of any selected feature listed in the main menu, the system will provide menu option again, as if it is back from the very beginning. This approach provides a bespoke, interactable, and convenience to the user allowing them easier navigation throughout the whole system.

A navigation flow is provided overleaf (see figure 1), to graphically show the menu flow and how the system proceeds.

Class Diagram

Laptop Supplier Record System

Carl Sagrado | April 30, 2021

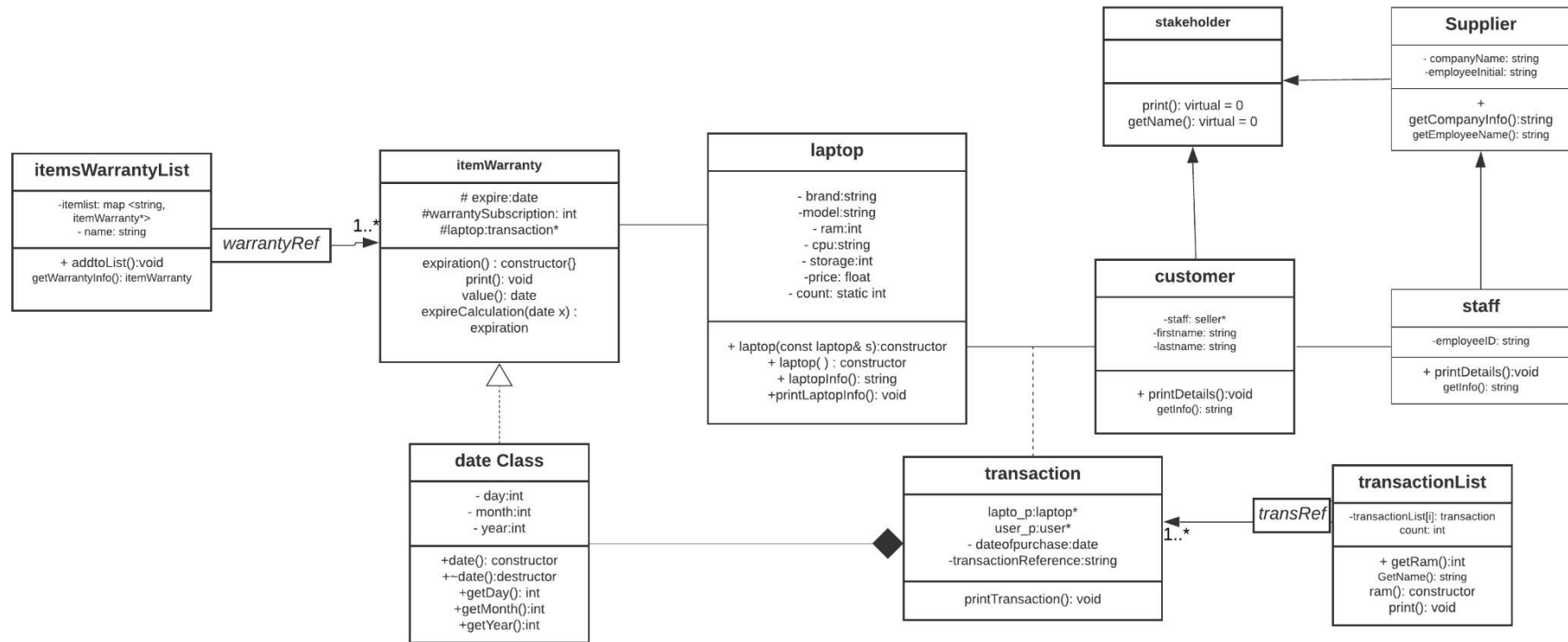


Figure 1: Class Diagram

NAVIGATION FLOW

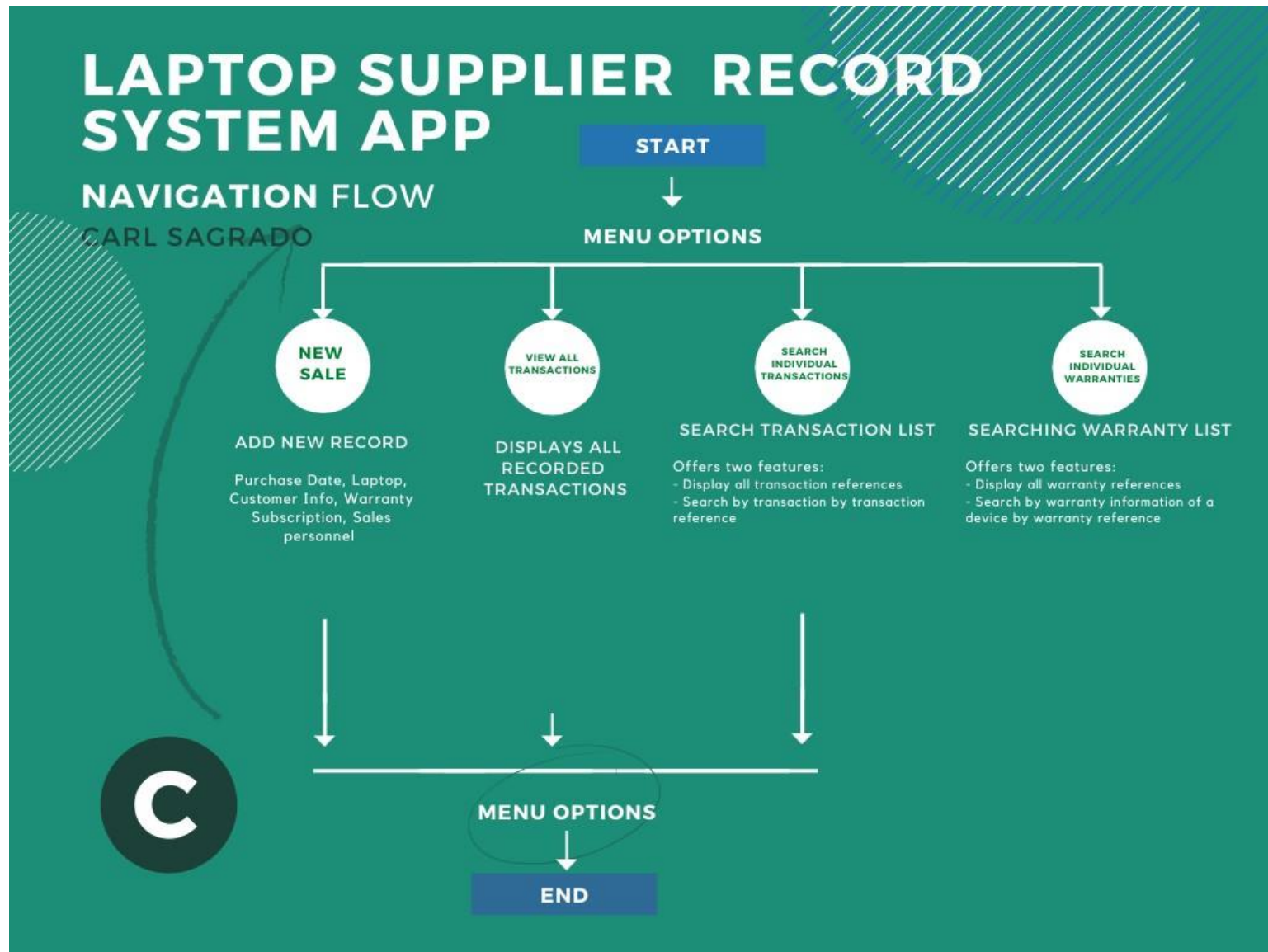


Figure 2: Navigation Flow

Technical Project Structure – Documentation

[1] Basic Class with access member functions, Constructors and static member data

A class is a defined code template for creating objects in object-oriented programming. A class is composed of member attributes and special methods like constructors, destructors, copy constructors, access member functions etc. to provide initialization and implementation behaviour of the class.

A **constructor** in a class is a method that is automatically called when an instance of the class is created. This method provides a structure template for arguments passed to the object and how it is going to be implemented inside the class. A **copy constructor** allows another class object to be passed in as an argument copying all the values of the class being passed to the new class. A **destructor** function denoted by '~' deletes a class object. Using access member functions allows access to attribute values of the class to which the function is set at. A static member data in a class is one that is declared static, there is only one copy of member data that exists and maintained for all objects of the class.

```
class laptop {
    friend ostream& operator<<(ostream& ostr, const laptop& b); // non-member function
public:
    laptop(const laptop& s); //copy constructor
    laptop(string br = "NA", string m = "NA", int r = 0, string c = "NA", int s = 0, double p
= 0) : brand(br), model(m), cp(c), storage(s), ram(r), price(p)
    {
        count++;}; //default constructort
    ~laptop(); //destructor
    //class member functions
    string getlaptopBrand() { return brand; }; //get the brand name of the laptop
    void printLaptopInfo() { //display laptop information
        cout << "Brand: " << brand << " " << model << "\nRam: " << ram <<
        " GB\nProcessor: " << cp << "\nStorage: " << storage << "GB\nPrice: €" << price;};
    static int getNum();
private:
    //attributes of a laptop
    string brand;
    string model;
    int ram;
    string cp;
    int storage;
    double price;
    static int count;
};

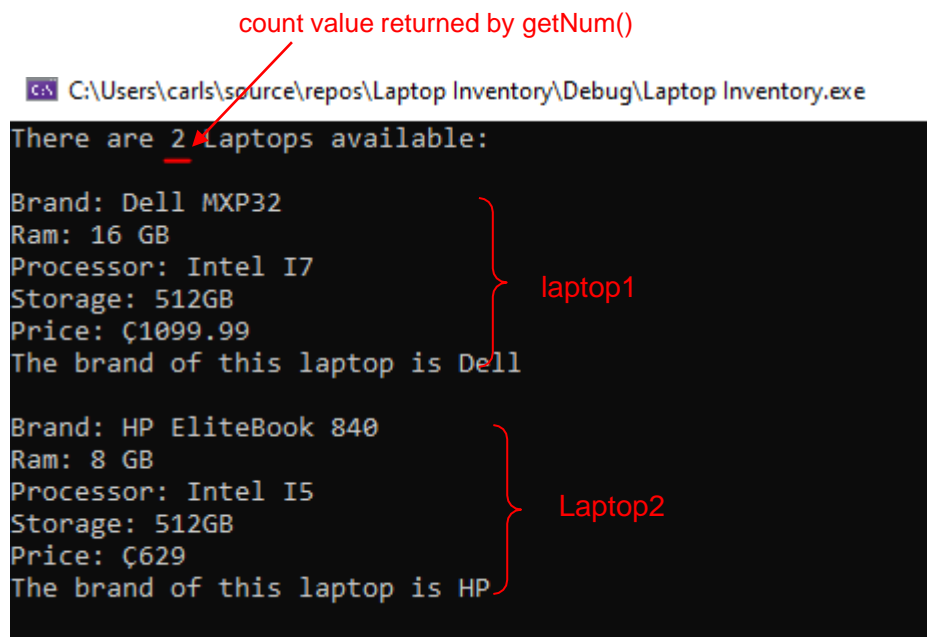
laptop::~laptop() { count--; } //when destructor is executed, decrement count
int laptop::count = 0; //initialize count
static int getNum() { return count; } //function count of instances of the laptop class
int main() {

    //Creating 2 laptops
    laptop laptop1("Dell", "MXP32", 16, "Intel I7", 512, 1099.99);
    laptop laptop2("HP", "EliteBook 840", 8, "Intel I5", 512, 629.00);
    //displaying the laptops
    cout << "There are " << laptop::getNum << " Laptops available: \n"; //calling the getNum
function to display number of laptops
    laptop1.printLaptopInfo(); //display laptop information
    cout << "\nThe brand of this laptop is " << laptop1.getlaptopBrand() << endl; //getting the
laptop brand
    laptop2.printLaptopInfo(); //display laptop information
    cout << "\nThe brand of this laptop is " << laptop2.getlaptopBrand() << endl; //getting
the laptop brand
}
```

To test the behaviour of the class "laptop", we created 2 laptop instances, laptop1 and laptop2 providing different attribute values for each object. The arrangement of the arguments is not ambiguous or random but a predefined arrangement of parameters by the class constructor. In order to successfully create class laptop objects, the arrangement of parameters must be met as well as the passing in the correct type of value.

Each time an object of the class laptop is created, the class attribute count is automatically incremented as defined in the laptop class constructor. To display the value of the count, we use the function getNum() which returns the count value.

count value returned by getNum()



```
C:\Users\carls\source\repos\Laptop Inventory\Debug\Laptop Inventory.exe
There are 2 Laptops available:
Brand: Dell MXP32
Ram: 16 GB
Processor: Intel I7
Storage: 512GB
Price: €1099.99
The brand of this laptop is Dell
Brand: HP EliteBook 840
Ram: 8 GB
Processor: Intel I5
Storage: 512GB
Price: €629
The brand of this laptop is HP
```

We can see that the laptop details are printed using printLaptopInfo() function which allows us access of to values of each laptop.

There are many ways to what we can do to the values of the class using different class member functions, but only through these functions we can access attribute values of the class. Without such functions, these attribute values will not be accessible outside the class.

Inheritance is one of the most fundamental feature of object oriented programming. It

[2] Inheritance and Polymorphism

Inheritance is one of the most fundamental features of object-oriented programming. It provides a capability for classes to derive and inherit properties from another class.

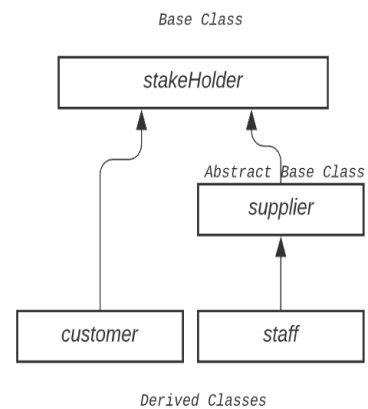
When a class is derived from an existing class, this kind of relationship is referred to as 'is-a' relationship. For example, if *customer* class is a derived class of a *stakeholder* class, this means that all data members and functions available on the stakeholder class (base class) is inherited by customer class.

There is an access control under inheritance, such as that while public members of the base class are accessible by all functions, the private members are only accessible by the member functions of the base class. Whilst the private members are inherited by the derived classes, these members will remain hidden. Thus, consideration towards use of access control such as private, protected or public is relevant.

Inheritance relationship can also be implemented in hierarchy similar to the diagram shown on the right. Where the customer class is a derived class of a stakeholder and the supplier class is an abstract base class which has an 'is a' relationship with the stakeholder class with staff class as its derived class. To turn an abstract base class to a derived class, the supplier must implement the virtual functions of the stakeholder class to observe polymorphism behaviour.

Polymorphism only occurs when there is a hierarchy of class and are related by inheritance. It is a behaviour where a call to a member function will cause a different function to be executed depending on the type of object that invokes it.

Consider the code below.



```

//Inheritance, Polymorphism, Basic Association, Base Class and Derived Class
class stakeHolder { //abstract base class
public:
    //pure virtual member functions
    virtual void printDetails() = 0;
    virtual string getInfo() = 0;
};

class Supplier : public stakeHolder { //abstract base class - can inherit all member and member
functions of the base class.
public:
    //to turn this into a derived class, virtual functions must be implemented inside the code
    Supplier(string tempComp = "Default", string empTemp = "STAFF") : companyName(tempComp),
employeeInitial(empTemp) {} //default constructor of the base class
    string getCompanyInfo() { return companyName; } //returning the resistance value without
changes+
    string getEmployeeName() { return employeeInitial; }
protected: //using protected modifier to make members inaccessible outside the class while still
accessible by derived/friend classes/functions.
    string companyName; //Company Name
    string employeeInitial; //employee initial only
};

class staff : public Supplier { // derived class - inherits all member and member functions of the
base classes.
public: //with inherited functions from base class
    staff(string tempComp = "Default", string empTemp = "STAFF", string tempId = "xx") :
Supplier(tempComp, empTemp), employeeID(tempId) {} //default constructor for the resistor class to
which value is assigned to the base class
    void printDetails() { cout << "Company: " << companyName << "\nStaff Name: " <<
employeeInitial << "\n Staff ID: " << employeeID << endl; }
    string getInfo() { return companyName; }
protected:
    string employeeID; //employee id
};

class customer : public stakeHolder { // derived class - inherits all member and member functions
of the base classes.
public:
    customer(staff* tempStaff, string fn = "n/a", string ln = "n/a") : firstName(fn),
lastName(ln), seller(tempStaff) {} //default constructor
    void printDetails() { cout << "Customer: " << firstName << " " << lastName << "\nServed
by: " << (seller->getEmployeeName())<< endl; }
    string getInfo() { //get purchaser information
        string purchaser = firstName + " " + lastName;
        return purchaser;
    }
protected:
    string firstName, lastName; //customer first name and last name
    staff* seller; //knows about the staff object passed to the customer object
};

```


The previously shown code is the implementation example of an inheritance relationship. Where the stakeholder and the supplier base classes in a hierarchal order and the staff class and the customer class are the derived classes that implements the virtual functions (see section [3] Abstract base class and concrete derived class).

To implement and observe inheritance and polymorphism behavior, we used the code below.

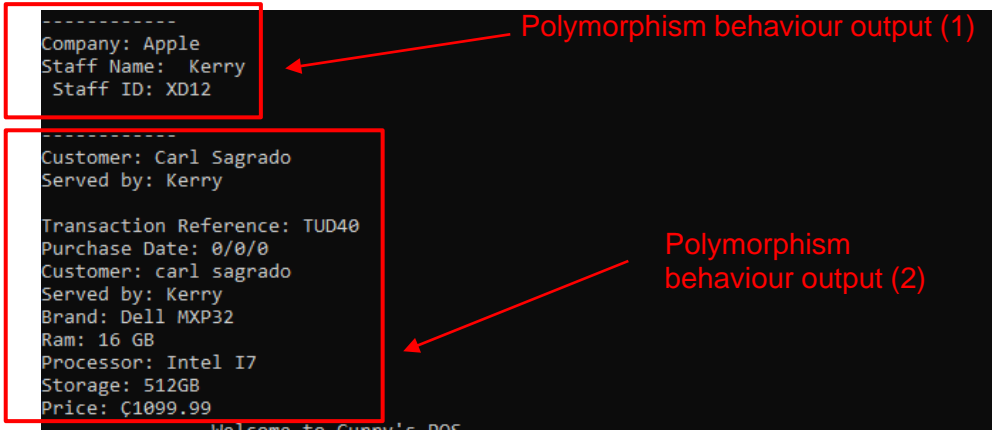
```
int main() { // Implementing the inheritance relationship and observing polymorphism

    laptop laptop1("Dell", "MXP32", 16, "Intel I7", 512, 1099.99); //creating laptop object
    staff salesPerson("Apple", "Kerry", "XD12"); //creating an object of class staff
    customer customer1(&salesPerson, "Carl", "Sagrado"); //creating an object of class
customer
    stakeHolder* Salesprofile;

    cout << "----- \n";
    // store the address of salesPerson
    Salesprofile = &salesPerson;
    // printDetails() function is a pure virtual function of base class and defined in derived
class customer
    Salesprofile->printDetails(); //calling printDetails function of the appropriate class

    cout << "\n----- \n";
    // store the address of customer1
    Salesprofile = &customer1;
    // printDetails() function is a pure virtual function of base class and defined in derived
class customer
    Salesprofile->printDetails(); //calling printDetails function of the appropriate class
}
```

Result:



```
-----
Company: Apple
Staff Name: Kerry
Staff ID: XD12

-----
Customer: Carl Sagrado
Served by: Kerry

Transaction Reference: TUD40
Purchase Date: 0/0/0
Customer: carl sagrado
Served by: Kerry
Brand: Dell MXP32
Ram: 16 GB
Processor: Intel I7
Storage: 512GB
Price: C1099.99
Welcome to Gurus's POS
```

Polymorphism behaviour output (1)

Polymorphism behaviour output (2)

The result above shows a runtime polymorphism behaviour . This type of polymorphism is achieved by function overriding where the derived classes such as staff and customer class has the definition of the pure virtual functions of the base class allowing 'salesProfile' object to have many forms and still the compiler is able to correctly identify which function of an any class within the scope of relationship to call.

[3] Abstract Base Class and Concrete Derived Class

Abstract Base Class is a class that is used as a base class from which classes are derived from. An abstract base class has the functions written but there is no object instantiated for it. The purpose of an abstract base class is to provide an interface or implementation from which classes can inherit. Classes from which objects can be instantiated are called Concrete classes. A **concrete derived class** is a class that implements all virtual functions written in the base class.

A **virtual** function is a function in a base class declared by using the keyword **virtual**. Setting a base class with a virtual function and redefining the same virtual function in a derived class signals the compiler to execute dynamic linkage or binding. To redefine a virtual function of the base class inside a derived class to better suit objects of that class, the keyword **pure virtual** is used. The = 0 tells the compiler that the function has no body and is defined inside derived classes.

Consider the code below:

```
// Base Class and Derived Class
class stakeHolder { //abstract base class
public:
    //pure virtual member functions
    virtual void printDetails() = 0;
    virtual string getInfo() = 0;
};

class staff : public stakeHolder { // derived class - inherits all member and member functions of
the base classes.
public: //with inherited functions from base class
    staff(string empTemp = "STAFF", string tempId = "xx") : employeeID(tempId),
employeeInitial(empTemp) {} //default constructor for the resistor class to which value is
assigned to the base class
    void printDetails() { cout << "Company: " << companyName << "\nStaff Name: " <<
employeeInitial << "\n Staff ID: " << employeeID << endl; }
    string getInfo() { return employeeInitial; }
protected:
    string employeeID; //employee id
    string employeeInitial; //employee id
};

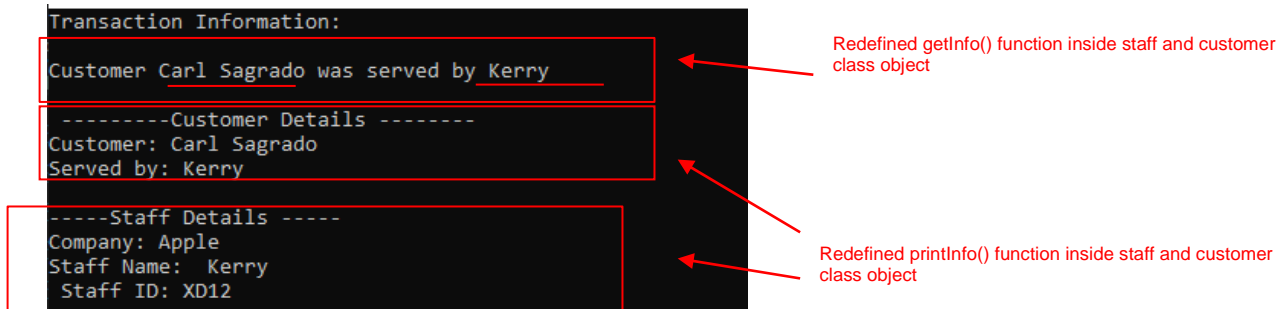
class customer : public stakeHolder { // derived class - inherits all member and member functions
of the base classes.
public:
    customer(staff* tempStaff, string fn = "n/a", string ln = "n/a") : firstName(fn),
lastName(ln), seller(tempStaff) {} //default constructor
    void printDetails() { cout << "Customer: " << firstName << " " << lastName<< endl; }
    string getInfo() { //get purchaser information
        string purchaser = firstName + " " + lastName;
        return purchaser;
    }
protected:
    string firstName, lastName; //customer first name and last name
    staff* seller; //knows about the staff object passed to the customer object
};

int main() {
    staff salesPerson("Apple", "Kerry", "XD12"); //creating an object of class staff
    customer customer1(&salesPerson, "Carl", "Sagrado"); //creating an object of class
customer
    cout << "Transaction Information: "<<endl;

    cout << "Customer "<<customer1.getInfo()<< " was served by " <<
salesPerson.getEmployeeName() << endl;

    cout << "-----Staff Details----- \n";
    salesPerson.printDetails();
}
```

The laptop and staff objects were created separately with appropriate parameters to satisfy its constructor. As we observed in the implementation code, we used same function which was declared and inherited from the base class. While both are different classes, we were able to inherit the same function and redefine the function inside each class during execution the compiler would know which function is being called. Thus, the output result would be as below.



```
Transaction Information:
Customer Carl Sagrado was served by Kerry
-----Customer Details -----
Customer: Carl Sagrado
Served by: Kerry
-----Staff Details -----
Company: Apple
Staff Name: Kerry
Staff ID: XD12
```

Redefined getInfo() function inside staff and customer class object

Redefined printInfo() function inside staff and customer class object

As observed, despite using the same function name in both derived classes, the compiler still recognizes which function is being called. Without pure virtual function and setting the function =0, the compiler would recognize the function as the default function for both derived classes.

[4] Queue linked list

A linked list refers to nodes which are connected in nature. It is a linear data structure which contains the data part and the next part. We see from our code below where we created a class called 'node' where we have transaction data which represents the data that holds class object and the node next which represents the node pointer called next.

```
class node
{
    friend class queue;
public:
    //node();
    node(const transaction& x) : data(x), next(NULL) {};
    node(); //member class function using function overloading

private: //private data
    transaction data;
    node* next;
};
node::node() :next(NULL) {} //definition of the member class function node
//class queue
class queue
{
public:
    queue(); //class constructor
    ~queue() {} //class destructor

    //Member class function definition
    queue(const queue&); //another function same with the constructor using function
overloading
    int push(transaction value); //push function
    transaction pop(void); //pop function
private: //private data
    node* listhead, * listtail;
};

queue::queue() //function definition for queue
{
    //setting listhead and listtail to null when called
    listhead = NULL;
    listtail = NULL;
}
```

```

queue::~~queue() //destructor definition
{
    node* temp;
    while (listhead != NULL)
    {
        // get the address of next node
        temp = listhead;
        // move to the next node on queue
        listhead = listhead->next;
        delete temp; //delete
    }
}

int queue::push(transaction value) //PUSH FUNCTION DEFINITION
{
    int error = 1;
    // get a new node
    node* temp = new node(value);
    if (temp == NULL) //queue full
    {
        cout << "queue overflow" << endl;
        error = -32000;
    } else
    { //temp->data = value; // NB Don't need for integers as
      // already done by constructor above
        if (listhead == NULL) //if empty
        {
            listhead = temp; //assign new node to listhead
        }
        else
        {
            listtail->next = temp; //else shift listtail
        }
        listtail = temp; //assign new node to listtail
    }
    return error;
}

transaction queue::pop(void) //POP FUNCTION DEFINITION
{
    node* temp;
    transaction value;
    if (listhead == NULL) //queue empty
    {
        cout << "queue underflow" << endl;
        //value = -32068; //have to return a value
        // NB can't do this if queue of some other class eg transaction
    } else { // get the value at the top of the list
        value = listhead->data;
        // remove the top entry from the list
        temp = listhead;
        listhead = listhead->next;
        delete temp;
        //Check if Queue empty
        if (listhead == NULL)
            listtail = NULL; //clear listtail
    } return value; // return the value
}

queue::queue(const queue& r)
{
    node* hold = r.listhead;
    node* temp, * oldtemp;
    if (hold == NULL) //queue empty
    {
        listhead = NULL;
        listtail = NULL;
    }
    else
    {
        temp = new node;
        listhead = temp;
        while (hold != NULL)
        {
            // get the value at the node looking at on queue 'c' list
            temp->data = hold->data;
            // move to the next entry on 'c' list
            hold = hold->next;
            oldtemp = temp;
            if (hold != NULL)
            {
                temp = new node;
                oldtemp->next = temp;
            }
        } listtail = temp;
    }
}

```

We manipulate the link list by inserting new nodes into the linked list (push) and removing node entry (pop). We use the push function to insert a new node at the start of the list which will serve as the head with the pointer value as null, then when it detects that it is the beginning of the list, it will assign temp to the head and the tail otherwise it will point and assign temp to the next node.

The pop function is used to retrieve data from the node and return it. In doing so, it will delete the current entry node.

Consider the implementation code below.

```
int main() {
    /*---Creating necessary objects to satisfy transaction class constructor-- */
    staff staff1("Apple", "Kerry", "XD12"); //creating a staff object
    date saleDate(26, 04, 2020); //creating a date object
    customer customer1(&staff1, "Carl", "Sagrado"); //creating customer object for each
    customer
    customer customer2(&staff1, "Earl", "Murphy");
    int transactionCount = 0;

    /*available laptops*/
    laptop laptop1("Dell", "MXP32", 16, "Intel I7", 512, 1099.99);
    laptop laptop2("HP", "EliteBook 840", 8, "Intel I5", 512, 629.00);

    //creating 2 transactions to observe queue linked list operation
    transaction transaction1(&laptop1, &customer1, saleDate);
    transaction transaction2(&laptop2, &customer2, saleDate);
    transaction transVal;

    queue transactionQueue; //create queue object
    /*---Using linked list to queue the two transactions--- */

    transactionQueue.push(transaction1); //pushing transaction1 object to the list
    transactionQueue.push(transaction2); //pushing transaction2 object to the list

    /*At this point, the queue would have two objects linked, next we will pop or extract
    those objects and display to the screen*/
    for (int i = 0; i < 2; i++) {
        transVal = transactionQueue.pop();
        transVal.printTransaction();
    }
}
```

Result:

```
Transaction Reference: TUD40
Purchase Date: 26/4/2020
Customer: Carl Sagrado
Served by: Kerry
Brand: Dell MXP32
Ram: 16 GB
Processor: Intel I7
Storage: 512GB
Price: C1099.99
Transaction1

Transaction Reference: TUD32
Purchase Date: 26/4/2020
Customer: Earl Murphy
Served by: Kerry
Brand: HP EliteBook 840
Ram: 8 GB
Processor: Intel I5
Storage: 512GB
Price: C629
Transaction2
```

The output result of the code above displays the behaviour of the queue linked list. The transaction value output from the pop function is maintained meanwhile taking in new transaction value from the queue list.

As we push the transaction object into the list, the first item pushed to the queue will serve as the head and when next transaction object is being pushed to the then when it detects that it is the beginning of the list, it will assign temp to the head and the tail otherwise it will point and assign temp to the next node.

[5] Operator overloading, as a member function

Operator overloading refers to redefining or overloading built-in operators in C++ with custom definition of the operator. Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

As a member function, this means that the overloaded function is defined inside the class and only its member has access to the function.

Consider the following code below.

```
class date {
    friend ostream& operator<<(ostream& ostr, const date& b); //output stream operator non-
    member function
public:
    date(int d = 1, int m = 1, int y = 2020) : day(d), month(m), year(y) {}; //constructor
    ~date() {}; //destructor

    //member function

    int getDay() { return day; }
    int getMonth() { return month; }
    int getYear() { return year; }
    date operator+(const date& add); //function overloading operator
    date operator-(const date& add); //function overloading operator
    int convertDatetoDays(date x) { //Converting date to days operator
        int dd, remainingD;
        int z = x.getDay();
        if (x.getMonth() > 0) {
            for (int i = 0; i < x.getMonth(); i++) {
                dd += 30; //1 month = 30 days
            }
        }
        if (x.getYear() > 0) {
            for (int i = 0; i < x.getYear(); i++) {
                dd = +365; //1 year ignoring leap year
            }
        }
        return remainingD = dd + z;
    }

private:
    int day, month, year;
};

date date :: operator+(const date& add) { //creating a function definition for '+' operator using
    scope resolution
    date ans;
    ans.day = day + add.day;
    ans.month = month + add.month;
    ans.year = year + add.year;
    return ans;
}

date date :: operator-(const date& add) { //creating a function definition for '-' operator using
    scope resolution
    date ans;
    ans.day = day - add.day;
    ans.month = month - add.month;
    ans.year = year - add.year;
    return ans;
}
```

Inside the date class, we redefined or overloaded two mathematic operators (addition and subtraction). These operators will perform addition or subtraction accordingly when any of the operators are used with the date objects. Not providing overloading functions would result to an error as the compiler would know what such operation intends to do.

See below testing of the overloaded member and nonmember function operators and its output.

Using overloaded function operator-, a member function of class date.

```
int main() {  
    date saleDate(26, 04, 2020); //creating a date object for purchase date  
    date expiryDate(26, 05, 2020); // creating a date object for warranty date  
    date daysRemaining = expiryDate - saleDate; // using function overloading operator- to  
    perform operation between dates  
    cout << "Purchased on: " << saleDate << endl;  
    cout << "Warranty date: " << expiryDate << endl;  
    cout << "Days remaining " << daysRemaining.calculateDaysRemaining(daysRemaining) << "days  
    \n" << endl; ; //displaying the date
```

Using overloaded output stream operator, a nonmember function of class date.

Result:

```
C:\Users\carls\source\repos\Laptop Inventory\Debug\Laptop Inventory.exe  
Purchased on: 26/4/2020  
Warranty date: 26/5/2020  
Days remaining 30days
```

[6] Operator overloading, as a nonmember function

This section discusses operator overloading, as a nonmember function. Similar to the discussed purpose of operator overloading in the previous section, the distinct difference of a nonmember operator overloading function is that it is denoted with a 'friend' function. This function has direct access to the private data of the class and the function is defined outside the class yet its argument of that class has unrestricted access to all class members.

To show an example for this topic, we are going to use the same date class used in the previous section. As we can see, the first item an output stream operator is declared outside the scope of the public or private and is denoted by the 'friend' keyword. That output stream operator function declared in the prototype is an example of a nonmember overloaded function of the class date.

The function definition of the ostream operator for the class date is defined as follows which is defined outside the class.

```
ostream& operator<<(ostream& ostr, const date& b) { //overloaded function definition of the non-  
member output stream operator for the date class  
    ostr << b.day << "/" << b.month << "/" << b.year; return ostr; //custom output  
}
```

The result of using the output stream operator to display saleDate and expiryDate objects of class date resulted to a format as defined within the function as seen in in the image above.

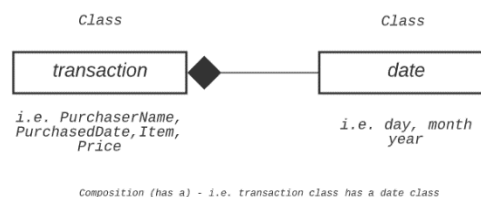
[7] Composition

A composition relationship in C++ refers to objects that is part of another object. The object that is part of another object is known as sub-object. Object composition models has a “has-a” relationship between objects. The relationship between these objects is a part/whole relationship where the class that defines the “part” of the class represents the “whole” of another class. This means that when a class object is created with a composition type relationship, the “part” is created by the constructor of the “whole” class and destroyed by the “whole” class.

For example, if a class date is part of the transaction class with a composition relationship, whenever a transaction object is created with parameters satisfying the constructor including a class date object the parameters passed to the transaction object for the date class values is created inside the class date.

The classes have a coincident life cycles, meaning that whenever the main class object that is of composition the sub-class of the main class will also get destroyed. The destruction of the subclass is done by its own destructor.

We can see the code below shows how it is defined and used where transaction class has a date class as a part of the whole transaction details, as seen depicted in the image below.



See implementation code below.

```
class date {
public:
    date(int d = 1, int m = 1, int y = 2020) : day(d), month(m), year(y) {};
    ~date() {};
    //Member function to display date object in a custom format

    void printPurchasedDate() {
        cout << "Purchased Date: " << day << "/" << month << "/" << year;
    }
private:
    int day, month, year;
};

class transaction {
public:
    transaction() {};
    transaction(date dop, string name, string x, double initial) : purchasedDate(dop),
        purchaserName(name), item(x), price(initial) {}; //constructor
    ~transaction() {}; //destructor

    //Printing transaction using member function
    void printTransaction() {
        cout << "-----Transaction Details-----\n\n";
        cout << "Purchased item: " << item;
        cout << "\nItem price: " << price << "EUR";
        cout << "\nPurchaser Name: " << purchaserName << endl;
        purchasedDate.printPurchasedDate(); //calling member function of date class
        cout << "\n\n-----";
    }
private:
    string purchaserName; //name of buyer
    string item; //item name
    double price; //price
    date purchasedDate; //when it was bought
};
```

Function of a date class

The code shown in the previous shows two classes, namely the transaction class and the date class. As observed the date class is a sub-class of the transaction class, thus the lifecycle of a date object inside the class is dependent on the creation and destruction of the transaction class to which it is part in.

To implement this relationship, we created an date object which we will pass as part of the qualifying arguments of the transaction class object.

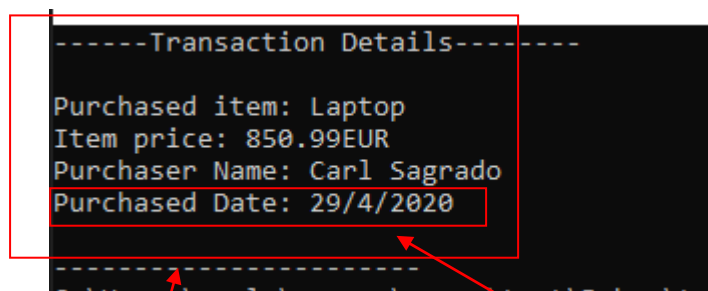
```
int main() {
    /*Testing Composition Relationship
    - Creating objects for both classes and observe the relationship between the
    two objects
    */

    date purchaseDate(29, 4, 2020); //create date object
    transaction transaction1(purchaseDate, "Carl Sagrado", "Laptop", 850.99);
    //create transaction object
    /*the transaction1 object HAS A date object to which its values is of
    purchasedDate
    - i.e. transaction class is composed of date class and other elements*/

    transaction1.printTransaction(); //calling a member function of the
    transaction

    return 0;
}
```

Result:



```
-----Transaction Details-----
Purchased item: Laptop
Item price: 850.99EUR
Purchaser Name: Carl Sagrado
Purchased Date: 29/4/2020
-----
C:\Users\user\source\repos\test\Debug>
```

Output of the printTransaction() function of the transaction class. The printTransaction function uses a function that is of a member function of a date class to display the purchased date

Output of the purchasedDate() function of a date class

[8] Basic Association

A basic association refers to a type of association relationship where one or both objects of different classes are associated to, but not, related to each other. Unlike composition where a class object is a part of the whole object, in association, objects only 'knows about' each other (depending on type of association) but not related or part of the object. In C++, basic association is implemented with a reference (pointer) in one class pointing to an object of the other class i.e. General POS system for laptop suppliers would require to know about who bought a laptop and who served the customer. Therefore, to implement this, a Customer class has a pointer type staff class as a data member that points to the object of the type staff. This means that while both classes exist independently to each other, Customer class object would know about staff class object rather than the staff object as part of the whole customer object.

For the case of basic association, a typical example for implementing this type of relationship is to use Customer and Staff classes which we discussed previously.

```
class staff { // standard class object
    friend ostream& operator<<(ostream& os, staff& c);
public:
    staff(string empTemp = "STAFF", string empID = "N/Z") : employeeID(empID),
    employeeInitial(empTemp) {} //default constructor for the resistor class to which value is
    assigned to the base class
    void printDetails() { cout << "\nStaff Name: " << employeeInitial << "\nStaff ID: " <<
    employeeID << endl; }
protected:
    string employeeID; //Comepany Name
    string employeeInitial; //employee initial only
};
class customer { // customer class - knows about staff.
public:
    customer(staff* tempStaff, string fn = "n/a", string ln = "n/a") : firstName(fn),
    lastName(ln), seller(tempStaff) {} //default constructor
    void printDetails() {
        cout << "Customer: " << firstName << " " << lastName << endl;
        seller->printDetails();
    }
    string getInfo() { //get purchaser information
        string purchaser = firstName + " " + lastName;
        return purchaser; }
protected:
    string firstName, lastName; //customer first name and last name
    staff* seller; //knows about the staff object passed to the customer object };

ostream& operator<<(ostream& os, staff& c) { // output stream operator for Employee
    os << "Served by: " << c.employeeInitial << "\nEmployee ID: " << c.employeeID << endl;
    return os; }

int main() {
    staff salesPerson("Kerry", "XD12"); //creating a staff object
    customer customer1(&salesPerson, "Carl ", "Sagrado"); //creating a customer object
    customer1.printDetails(); //testing class member function of customer class
    /*We passed in the salesPerson object with reference as part of the customer object but it
    only knows the address and does not copy it.
    It only needs to 'know about' the sales person without having its own staff data member,
    so we used a pass by reference to pass in the address of staff object */

    return 0;
}
```

Result:

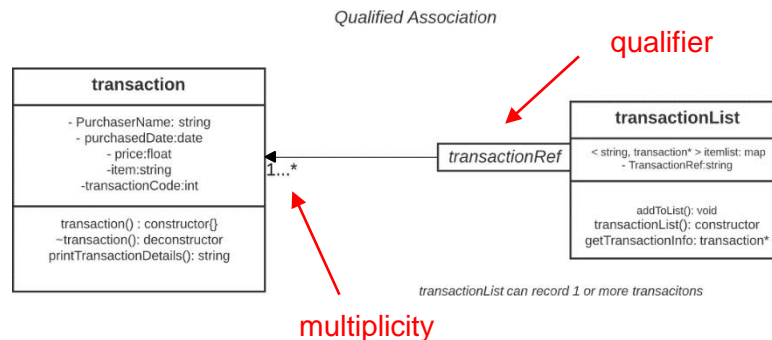
Microsoft Visual Studio Debug Console

```
Customer Information
-----
Customer: Carl  Sagrado
Staff Name: Kerry
Staff ID: XD12
```

Using printDetails() function of the customer class. The customer class knows about the staff class while the staff object does not.

[9] Qualified Association

A Qualified association is a type of association relationship that uses a qualifier. The qualifier is not part of the associated objects (i.e., not part of the class) but uniquely identifies the associated object of that class. Most practical application for this particular association relationship is with lookup tables, arrays, maps, etc. where we use a qualifier to select an object out of an object list. In the case of image shown below, the transRef is the qualifier for the associated class which identifies the transaction class in the transactionList class. The qualifier separates and identifies each transaction in the transaction list.



The image above shows the UML class diagram of a qualified association, where the connections between the class where the transactionList takes in one or more transactins with qualifiers. The implementation for this code is shown below. Note that for the actual project, the implementation will be modified to suit a pragmatic program for a POS device for laptops.

```
#include <iostream>
#include <map>
using namespace std;
class date {
public:
    date(int d = 1, int m = 1, int y = 2020) : day(d), month(m), year(y) {};
    ~date() {};
    //Member function to display date object in a custom format

    void printPurchasedDate() { //Member function to display date
        cout << "Purchased Date: " << day << "/" << month << "/" << year;
    }
private:
    int day, month, year; //private data member
};

class transaction { //modified transaction class
    friend ostream& operator<<(ostream& ostr, transaction& a); //non member output stream
public:
    transaction() {};
    transaction(const transaction& s);
    transaction(date dop, string name, string x, double initial) : purchasedDate(dop),
    purchaserName(name), item(x), price(initial) {}; //constructor
    ~transaction() {}; //destructor
    //Printing transaction using member function
    void printTransaction() {
        cout << "-----Transaction Details----- \n\n";
        cout << "Purchased item: " << item;
        cout << "\nItem price: " << price << "EUR";
        cout << "\nPurchaser Name: " << purchaserName << endl;
        purchasedDate.printPurchasedDate(); //calling member function of date class
        cout << "\n\n";
    }
private:
    string purchaserName; //name of buyer
    string item; //item name
    double price; //price
    date purchasedDate; //when it was bought
};
```

```

transaction::transaction(const transaction& s) {
    purchaserName = s.purchaserName;
    item = s.item;
    price = s.price;
    purchasedDate = s.purchasedDate;
}
ostream& operator<<(ostream& os, transaction& c) { // output stream operator for Employee
    c.printTransaction();
    return os;
}

class transactionList { //transaction class with qualifier
public:
    //constructor
    transactionList() {};
    transactionList(string c = "ReferenceNA") : transactionReference(c) { } //default
    constructor
    ~transactionList() {}; //destructor

    void AddtoList(transaction* d, string saleCode) { //adding transaction to a map array with
a qualifier
        itemlist[saleCode] = d;
    }

    transaction* getTransactionInfo(string OrderCode) {
        return itemlist[OrderCode]; //using qualifier to identify the object inside the item
    }

private:
    string transactionReference;
    std::map < string, transaction* > itemlist; //map itemlist for
};

int main() {
    transactionList salesList("Transaction List ");
    date purchasedDate(29, 04, 2020); //Example date object

    /*creating 3 transactions that occurred on the same date*/
    transaction transaction1(purchasedDate, "Carl Sagrado", "Laptop", 1003.99);
    transaction transaction2(purchasedDate, "Steven Giblin", "Monitor AOC 144Hz", 250.99);
    transaction transaction3(purchasedDate, "Andy Mahony", "Dell Keyboard", 40.99);

    /*listing all transactions to transactionList with a qualifier*/
    salesList.AddtoList(&transaction1, "trans1");
    salesList.AddtoList(&transaction2, "CODE321");
    salesList.AddtoList(&transaction3, "LOCK2020");

    /*Using a Qualifier to extract a particular transaction from the list*/

    transaction* recentTrans1 = salesList.getTransactionInfo("LOCK2020");
    cout << *recentTrans1; //display the transaction

    transaction* recentTrans2 = salesList.getTransactionInfo("CODE321");
    cout << *recentTrans2; //display the transaction

    transaction* recentTrans3 = salesList.getTransactionInfo("trans1");
    cout << *recentTrans3; //display the transaction
}

```

Result:

Every transaction is recorded to the transaction list record, with a qualifier. The qualifier determines which transaction record is being recorded and retrieved. We made 3 transactions and added to the queue with specified qualifier and retrieved those transactions and displayed them.

The result on the right shows the output of the code that implements a qualifier association.

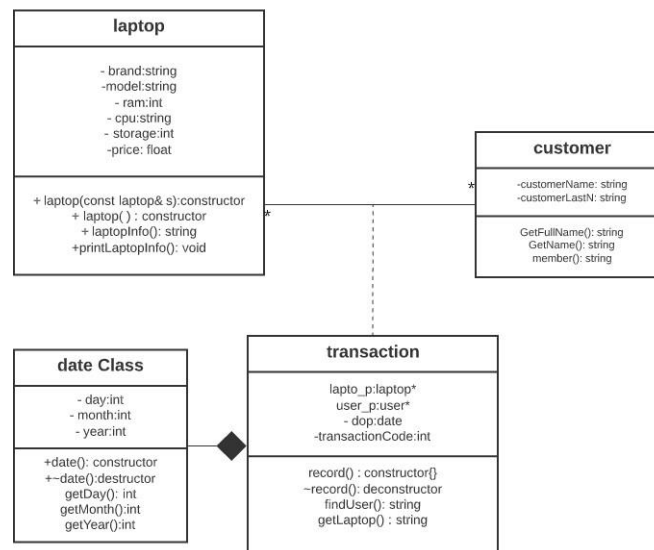
```

-----Transaction Details-----
Purchased item: Dell Keyboard
Item price: 40.99EUR
Purchaser Name: Andy Mahony
Purchased Date: 29/4/2020
-----Transaction Details-----
Purchased item: Monitor AOC 144Hz
Item price: 250.99EUR
Purchaser Name: Steven Giblin
Purchased Date: 29/4/2020
-----Transaction Details-----
Purchased item: Laptop
Item price: 1003.99EUR
Purchaser Name: Carl Sagrado
Purchased Date: 29/4/2020

```

[10] Association Class

An association class refers to a class that is part of an association relationship between two other classes. The associated class will have attributes and operations of other classes. When building an association class requires the associated class to know about the two other classes and at the same time have its own attributes to provide additional information about the relationship. Consider the UML Class Diagram below.



The above images show an example of an associated class, where the transaction class is the associated class that knows about the laptop and customer class objects. The transaction class keeps hold of all the transaction including the date, laptop, customer and transaction code as a whole functional object. Whenever someone buys a laptop, the system will keep record of the details of the laptop and the customer who bought it which represents a sale, the transaction class then further defines the relationship between the laptop and the customer by providing date of purchase and unique transaction code information related to the association relationship.

```
#include <iostream>
#include <string>

using namespace std;

class date {
    friend ostream& operator<<(ostream& os, date& c); //non member function
public:
    date(int d = 1, int m = 1, int y = 2020) : day(d), month(m), year(y) {}; //default
    constructor
    ~date() {}; //destructor
    //Member function to display date object in a custom format
    void printPurchasedDate() {
        cout << "Purchased Date: " << day << "/" << month << "/" << year;
    }
private:
    int day, month, year; //data member
};

ostream& operator<<(ostream& os, date& c) { //display purchasedDateVal
    c.printPurchasedDate(); return os;
}
```

```

class staff { // standard class object
public:
    staff(string empTemp = "STAFF", string empID = "N/Z") : employeeID(empID),
    employeeInitial(empTemp) {} //default constructor for the resistor class to which value is
    assigned to the base class
    void printDetails() { cout << "Staff Name: " << employeeInitial << "\nStaff ID: " <<
    employeeID << endl; }
protected:
    string employeeID; //Comepany Name
    string employeeInitial; //employee initial only
};

class customer { // customer class - knows about staff.
public:
    customer(staff* tempStaff, string fn = "n/a", string ln = "n/a") : firstName(fn),
    lastName(ln), seller(tempStaff) {} //default constructor
    void printDetails() {
        cout << "Customer: " << firstName << " " << lastName << endl;
        seller->printDetails();
    }
    string getInfo() { //get purchaser information
        string purchaser = firstName + " " + lastName;
        return purchaser;
    }
protected:
    string firstName, lastName; //customer first name and last name
    staff* seller; //knows about the staff object passed to the customer object };
};

class laptop {
    friend ostream& operator<<(ostream& ostr, const laptop& b); // non-member function
public:
    laptop(const laptop& s); //copy constructor
    laptop(string br = "NA", string m = "NA", int r = 0, string c = "NA", int s = 0, double p
    = 0) : brand(br), model(m), cp(c), storage(s), ram(r), price(p)
    {
        count++;
    }; //default constructort
    ~laptop(); //destructor
    //class member functions
    string getLaptopBrand() { return brand; }; //get the brand name of the laptop
    void printLaptopInfo() { //display laptop information
        cout << "Brand: " << brand << " " << model << "\nRam: " << ram <<
        " GB\nProcessor: " << cp << "\nStorage: " << storage << "GB\nPrice: €" <<
        price;
    };
    static int getNum() { return count; }////function count of instances of the laptop class
private:
    //attributes of a laptop
    string brand;
    string model;
    int ram;
    string cp;
    int storage;
    double price;
    static int count;
};

laptop::~laptop() { count--; } //when destructor is executed, decrement count
int laptop::count = 0; //initialize count

laptop::laptop(const laptop& s) { //copy constructor
    brand = s.brand;
    model = s.model;
    ram = s.ram;
    cp = s.cp;
    storage = s.storage;
    price = s.price;
    count++;
}

```

```

class transaction {
    friend ostream& operator<<(ostream& os, transaction& c); //non member function output
    stream operator
public:
    transaction() {};
    transaction(const transaction& s); //copy constructor
    transaction(laptop* Laptop, customer* User, date dop) : laptop_p(Laptop),
customer_p(User), dateofPurchase(dop)
    {
        int y = rand() & 1000; //generate transaction reference
        transactionReference = "TUD" + std::to_string(y);
        transactionCount++;
    };
    ~transaction() //destructor
    {
        transactionCount--;
    };
    //finding user - will display user name + laptop records
    string findUser(string firstName, string lastName)
    {
        return customer_p->getInfo();
    }
    string getlaptop()const { return(laptop_p->getlaptopBrand()); };
    //get user name - will return name of the person
    string getName() const { return(customer_p->getInfo()); };
    //get reference record - TO DO!!
    string getRef()const { return transactionReference; };
    static int getNum() { return transactionCount; }
    //print transaction information
    void printTransaction() {
        cout << "Transaction Reference: " << transactionReference << endl;
        cout << dateofPurchase << endl;
        customer_p->printDetails(); //calling member function for class customer
        laptop_p->printLaptopInfo(); //calling member functin for laptop class
        cout << "\n";
    }

private:
    laptop* laptop_p;
    customer* customer_p;
    string transactionReference;
    date dateofPurchase;
    static int transactionCount;
};

int transaction::transactionCount = 0; //initialize count

transaction::transaction(const transaction& s) {
    customer_p = s.customer_p;
    laptop_p = s.laptop_p;
    transactionReference = s.transactionReference;
    dateofPurchase = s.dateofPurchase;
    transactionCount++;
}
ostream& operator<<(ostream& os, transaction& c) { // output stream operator for Employee
    c.printTransaction();
    return os;
}

```

```

int main() {

    /*---Creating necessary objects to satisfy transaction class constructor---
    */
    staff staff1("Kerry", "XD12"); //creating a staff object
    date saleDate(26, 04, 2020); //creating a date object
    customer customer1(&staff1, "Carl", "Sagrado"); //creating customer object
    for each customer
        customer customer2(&staff1, "Earl", "Murphy");
    int transactionCount = 0;

    /*available laptops*/
    laptop laptop1("Dell", "MXP32", 16, "Intel I7", 512, 1099.99);
    laptop laptop2("HP", "EliteBook 840", 8, "Intel I5", 512, 629.00);

    //creating 2 transactions to observe queue linked list operation
    transaction transaction1(&laptop1, &customer1, saleDate);
    transaction transaction2(&laptop2, &customer2, saleDate);

    //customer1.printDetails(); //testing class member function of customer
class
    cout << " -----Transaction 1----- \n";
    transaction1.printTransaction();

    cout << " \n-----Transaction 2 ----- \n";
    transaction2.printTransaction();
    cout << "\n----- \n";
    cout << "Number of transactions: " << transaction::getNum();
    cout << "\n----- \n";
    return 0;
}

```

Result:

Microsoft Visual Studio Debug Console

```

-----Transaction 1 -----
Transaction Reference: TUD40
Purchased Date: 26/4/2020
Customer: Carl Sagrado
Staff Name: Kerry
Staff ID: XD12
Brand: Dell MXP32
Ram: 16 GB
Processor: Intel I7
Storage: 512GB
Price: €1099.99

-----Transaction 2 -----
Transaction Reference: TUD32
Purchased Date: 26/4/2020
Customer: Earl Murphy
Staff Name: Kerry
Staff ID: XD12
Brand: HP EliteBook 840
Ram: 8 GB
Processor: Intel I5
Storage: 512GB
Price: €629

-----
Number of transactions: 2
-----

```

To observe and test the association class topic, we created two transaction objects by passing a set of relative objects to the argument such as pointer reference to the laptop object, customer object and date object. Each time a transaction object is created the objects are counted by the transactionCount member of the transaction class.

Transaction1, an association class object that has an association relationship between customer and laptop objects.

Transaction2

Transaction object counter

A linked list refers to nodes which are connected in nature. It is a linear data structure which contains the data part and the next part. We see from our code below where we created a class called 'node' where we have transaction data which represents the data that holds class object and the node next which represents the node pointer called next.

```
#include <iostream>
#include <map>
using namespace std;
class date {
public:
    date(int d = 1, int m = 1, int y = 2020) : day(d), month(m), year(y) {};
    ~date() {};
    //Member function to display date object in a custom format

    void printPurchasedDate() { //Member function to display date
        cout << "Purchased Date: " << day << "/" << month << "/" << year;
    }
private:
    int day, month, year; //private data member
};

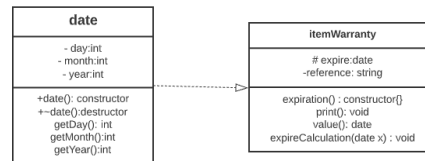
class transaction { //modified transaction class
    friend ostream& operator<<(ostream& ostr, transaction& a); //non member output stream
operator
public:
    transaction() {};
    transaction(const transaction& s);
    transaction(date dop, string name, string x, double initial) : purchasedDate(dop),
purchaserName(name), item(x), price(initial) {}; //constructor
    ~transaction() {}; //destructor
    //Printing transaction using member function
    void printTransaction() {
        cout << "-----Transaction Details -----\\n\\n";
        cout << "Purchased item: " << item;
        cout << "\\nItem price: " << price << "EUR";
        cout << "\\nPurchaser Name: " << purchaserName << endl;
        purchasedDate.printPurchasedDate(); //calling member function of date class
        cout << "\\n\\n";
    }
private:
    string purchaserName; //name of buyer
    string item; //item name
    double price; //price
    date purchasedDate; //when it was bought
};
```

[10] Dependency

As the name implies, dependency is a relationship where one class depends on another class. It is essentially a type of relationship where a change to an independent object affects another dependent object.

Dependency relationship is the weakest class relationship and is implemented by another class as a function parameter of a class.

Dependency Relationship



With reference to the above diagram, a using relationship means that if a change is applied to the date (i.e., change of date, etc.) affects the warranty date, but the reverse is not true.

Consider the code below.

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

class itemWarranty {
    friend ostream& operator<<(ostream& ostr, itemWarranty& a); //output stream operator
public:
    itemWarranty(const itemWarranty& s) { expire = s.expire; warrantyReference =
s.warrantyReference; } //copy constructor of the itemWarranty class
    itemWarranty(int dd, int mm, int yy, string l) : expire(dd, mm, yy), warrantyReference(l)
{};

    itemWarranty(date purchasedDate, int warrantyExp = 2) {
        calculateExpiration(purchasedDate, warrantyExp); //date expire value
        int a = rand() & 1000; //generate transaction reference
        warrantyReference = "TUD" + std::to_string(a);
    } //constructor for date class;

    //Member Functions

    void print() {
        cout << "Warranty Expiration: " << expire <<"Reference Code:" <<warrantyReference
<<endl;
    }
    void calculateExpiration(date a, int warrantyExp) { //expire data object of this class is
dependent on object value of another class
        int x = a.getDay();
        int y = a.getMonth();
        int z = a.getYear() + warrantyExp;
        date b(x, y, z);
        expire = b;
    }

private: //using protected modifier to make members inaccessible outside the class while still
accessible by derived/friend classes/functions.
    date expire;
    string warrantyReference;
};

ostream& operator<<(ostream& ostr, itemWarranty& a) {
    a.print();
    return ostr;
}
```

```

class date {
    friend ostream& operator<<(ostream& os, date& c); //non member function
public:
    date(int d = 1, int m = 1, int y = 2020) : day(d), month(m), year(y) {}; //default
    constructor
    ~date() {}; //destructor
    //Member function to display date object in a custom format
    void printPurchasedDate() {
        cout << day << "/" << month << "/" << year << endl;
    }

    int getDay() { return day; }
    int getMonth() { return month; }
    int getYear() { return year; }
private:
    int day, month, year; //data member
};

ostream& operator<<(ostream& os, date& c) { //display purchasedDateVal
    c.printPurchasedDate(); return os;
}

int main()
{
    /*Dependency Testing*/
    date *purchasedDate = new date(29, 04, 2021); //object of date
    int yearsofWarranty = 2;
    itemWarranty* warrantyDate = new itemWarranty(*purchasedDate, yearsofWarranty); //creating
    an object of warranty class which uses a uses the class object being passed in the parameter
    (purchaseDate) as a parameter for calculateExpiration function to calculate the value of the
    expiration date. This occurs in the constructor itemWarranty constructor

    cout << "----Relevant Dates:--- \n" << //printing initial itemWarranty object value
        "Purchased Date: " << *purchasedDate << *warrantyDate;

    /*Changing the purchase date*/
    cout << "\n***Changing the purchase date ***** \n";
    date* x = new date(01, 06, 2021);

    *purchasedDate = *x;

    /*updating the warrantyDate*/
    warrantyDate->calculateExpiration(*purchasedDate, yearsofWarranty);

    cout << "\n----New Relevant Dates:--- \n" << //printing new itemWarranty object value
        "Purchased Date: " << *purchasedDate << *warrantyDate;

    delete x; //delete date object
}

```

Result:

```

----Relevant Dates:---
Purchased Date: 29/4/2021
Warranty Expiration: 29/4/2023
Reference Code:TUD40

***Changing the purchase date *****

----New Relevant Dates:---
Purchased Date: 1/6/2021
Warranty Expiration: 1/6/2023
Reference Code:TUD40

```

The image on the right displays the output result of the implemented code. Notice that the expiration date has changed according to the relevant date. The dependency occurs where whenever a new object of class itemWarranty is created, it calls a function inside the class that requires a date object as a parameter to calculate and set the expiration date.

[11] An interesting fact about the Project

The most novel and challenging part in developing this program was at initial stage of the design where we have to decide which functionalities and features is in relationship with which class. We were able to provide a solution to this particular challenge by redefining the goal of the project and its purpose, who are the user and what could this platform do for them. At the beginning, it was also challenging understand the key differences between relationship and how it would be applied in this application. Further understanding and tutorials from the lecturer helped. Finally, creating a menu that allows all functionalities to work and output desired result was challenging as there were many duplications within the code which was addressed and resolved.

Testing:

Scenario: A customer purchases a laptop from Currys IT Shop. Kerry, the sales attendant processes this purchase.

Menu:

```
===== Welcome to Curry's POS =====  
Select from the menu  
[1]: New Sale [2] View Transactions [3] Check Transaction References [4] Check Warranty References  
Select a value from the menu:
```

[1] New Sale

```
C:\> C:\Users\carls\source\repos\Laptop Inventory\Debug\Laptop Inventory.exe  
Enter Transaction date:
```

Selecting a laptop

```
C:\> C:\Users\carls\source\repos\Laptop Inventory\Debug\Laptop Inventory.exe  
-----List of Laptops on sale -----  
[1]Brand: Dell MXP32  
Ram: 16 GB  
Processor: Intel I7  
Storage: 512GB  
Price: C1099.99  
[2]Brand: Dell MXP32  
Ram: 16 GB  
Processor: Intel I7  
Storage: 512GB  
Price: C1099.99  
[3]Brand: HP EliteBook 840  
Ram: 8 GB  
Processor: Intel I5  
Storage: 512GB  
Price: C629  
[4]Brand: Lenovo IdeaPad 750  
Ram: 8 GB  
Processor: Intel I5  
Storage: 256GB  
Price: C674.99  
Select a Laptop:
```

Entering Customer info

```
C:\> C:\Users\carls\source\repos\Laptop Inventory\Debug\Laptop Inventory.exe  
Enter Customer Name:
```

New Sale Confirmation:

```
The item has been registered successfully!

TransactionID: POSDX41
WarrantyID: SDA668
Customer: Carl Sagrado
Served by: Kerry

Add another transaction? Y/N:
```

Scenario2: Few hours later, a customer walks in to the door and wishes to purchase a new laptop from. Again, Kerry processes this purchase.

```
The item has been registered successfully!

TransactionID: POSDX500
WarrantyID: SDA1448
Customer: Terry Cruz
Served by: Kerry

Add another transaction? Y/N:
```

Next the manager wishes to check the list of transactions that was done today.

```
C:\Users\carls\source\repos\Laptop Inventory\Uebug\Laptop Inventory.exe
***** Welcome to Curry's POS *****

Select from the menu
[1]: New Sale [2] View Transactions [3] Check Transaction References [4] Check Warranty References
Select a value from the menu: 2
```

```
Transaction Reference: TUD32
Purchase Date: 29/4/2021
Customer:
Served by: Kerry
Brand: HP EliteBook 840
Ram: 8 GB
Processor: Intel I5
Storage: 512GB
Price: €629
```

```
Transaction Reference: TUD736
Purchase Date: 29/4/2021
Customer:
Served by: Kerry
Brand: Dell MXP32
Ram: 16 GB
Processor: Intel I7
Storage: 512GB
Price: €1099.99
```

Few month later, the first customer came back to complain a broken laptop. Kerry checks the transaction information using the transactionID.

```
[1] View Transaction References [2] Search Transaction Details
Enter value from the menu: 1
-----Transaction References-----
Transaction:0:POSDX41
Transaction:1:POSDX500
Would you like to view more Transaction Records? Y/N
```

After knowing the transaction references, she proceed to search for the transaction details.

```
[1] View Transaction References [2] Search Transaction Details
Enter value from the menu: 2
===== Checking Transaction Details =====

Enter Transaction Reference: POSDX41
Transaction References: Transaction Reference:
Purchase Date: 29/4/2021
Customer:
Served by: Kerry
Brand: Dell MXP32
Ram: 16 GB
Processor: Intel I7
Storage: 512GB
Price: C1099.99
Would you like to view more Transaction Records? Y/N
```

And checks the warranty details.

```
[1] View Warranty References [2] Check Item Warranty Details1
List of Item Warranty References

Transaction: 0:SDA668
```

```
===== Checking Warranty Details =====

Enter Item Warranty Reference: SDA668
Warranty References
Transaction: 0:SDA668
===== WARRANTY INFORMATION =====

Warranty Expiration: 29/4/2023
Duration: 2 Years
Laptop: Dell MXP32
Ram: 16GB
Processor: Intel I7
Storage: 512GB
Price: C1099.99

Would you like to view more Warranty Records? Y/N :
```

Conclusion

The system app has been developed to meet the user requirements allowing them to list any purchases and customer information they wish. As seen in the Testing section of this report, we have proven that each feature of this program works and as intended. The relevant section of the codes are written and commented, and important topics of the program (i.e., classes, association relationships, inheritance, composition, C++ features, etc) are explained under Technical project structure of this document. The source code is provided along with this document contains the original code and the comments related to the purpose for most lines of code. Having met the requirements specified in developing the Supplier App program and successful testing of the program concludes this report.

Source Code:

```
#include <iostream>
#include<fstream>
#include <string>
#include<stdlib.h>
#include<conio.h>
#include <map>

using namespace std;

string referenceGenerator(int y);

class laptop {
    friend ostream& operator<<(ostream& ostr, const laptop& b); // non-member function
public:
    laptop(const laptop& s); //copy constructor
    laptop(string br = "NA", string m = "NA", int r = 0, string c = "NA", int s = 0, double p
= 0) : brand(br), model(m), cp(c), storage(s), ram(r), price(p)
    {
        count++;}; //default constructort
    ~laptop(); //destructor

    //class member functions
    string getlaptopBrand() { return brand; }; //get the brand name of the laptop
    void printLaptopInfo() { //display laptop information
        cout << "Brand: " << brand << " " << model << "\nRam: " << ram <<
            " GB\nProcessor: " << cp << "\nStorage: " << storage << "GB\nPrice: €" << price;
    };
    static int getNum() { return count; } //getting the count of instances of the laptop class
private:
    //attributes of a laptop
    string brand;
    string model;
    int ram;
    string cp;
    int storage;
    double price;
    static int count;
};

laptop::~laptop() { count--; } //when destructor is executed, decrement count
int laptop::count = 0; //initialize count

class date {
    friend ostream& operator<<(ostream& ostr, const date& b); //output stream operator non mem
public:
    //date(int, int, int);
    date(const date& s); ///*-----inline copy constructor declaration---
    date(int d = 1, int m = 1, int y = 2020) : day(d), month(m), year(y) {};
    ~date() {};

    //member function

    int getDay() { return day; }
    int getMonth() { return month; }
    int getYear() { return year; }
    date operator+(const date& add); //function overloading operator
    //date operator-(const date& add); //function overloading operator
    int calculateDaysRemaining(date x) {
        int dd = 0, remainingD = 0;
        int z = x.getDay();
        if (x.getMonth() > 0) {
            for (int i = 0; i < x.getMonth(); i++) {
                dd +=30; //1 month } }
            if (x.getYear() > 0) {
                for (int i = 0; i < x.getYear(); i++) {
                    dd = +365; //1 year }
                };
            return remainingD = dd + z;
        }
    }

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

```

date date :: operator+(const date& add) { //creating a function definition for '+' operator using
scope resolution
    date ans;
    ans.day = day + add.day;
    ans.month = month + add.month;
    ans.year = year + add.year;
    return ans;
}

ostream& operator<<(ostream& ostr, const date& b) { //overloaded function definition of the non-
member output stream operator for the date class
    ostr << b.day << "/" << b.month << "/" << b.year; return ostr; //custom output
}

date::date(const date& s) {
    day = s.day;
    month = s.month;
    year = s.year;
}

class itemWarranty {
    friend ostream& operator<<(ostream& ostr, itemWarranty& a); //output stream operator
public:
    itemWarranty(const itemWarranty& s) { expire = s.expire; laptop_p = s.laptop_p;
warrantySubscription = s.warrantySubscription; } //copy constructor of the expiration class
    itemWarranty(date purchasedDate, laptop* device, int warrantyExp = 2) : laptop_p(device),
warrantySubscription(warrantyExp), expire(purchasedDate) {
        calculateExpiration(purchasedDate, warrantyExp);
    } //constructor for date class;

    //Member Functions
    void printVal() { cout << "test"; }
    //void printType() { cout << "resistance "; } //overloading printType function inside the
derived Resistor class

    itemWarranty operator+(const itemWarranty& add) { //creating a function definition for '+'
operator using scope resolution
        itemWarranty ans = add;
        ans.warrantySubscription = add.warrantySubscription;
        //ans.laptop_p = add.laptop_p;
        int x = ans.expire.getDay();
        int y = ans.expire.getMonth();
        int z = ans.expire.getYear() + ans.warrantySubscription;
        date newDate(x, y, z);

        ans.expire = newDate;
        return ans;
    }
    void print() { cout << "Warranty Expiration: " << expire << endl; } //overloading
printType function inside the derived Inductor class
    date getExpiryDate() { return expire; };
    void calculateExpiration(date a, int warrantyExp) {
        int x = a.getDay();
        int y = a.getMonth();
        int z = a.getYear() + warrantyExp;
        date b(x, y, z);
        expire = b;
    }

protected: //using protected modifier to make members inaccessible outside the class while still
accessible by derived/friend classes/functions.
    date expire;
    int warrantySubscription;
    laptop* laptop_p;
    //saleReference* sale;
};

ostream& operator<<(ostream& ostr, itemWarranty& a) {
    ostr << "***** WARRANTY INFORMATION *****\n\n";
    ostr << "Warranty Expiration: " << a.expire << "\nDuration: " << a.warrantySubscription <<
" Years\n" << *(a.laptop_p) << endl;
    return ostr;
}

```



```

class warrantyRecords {
public:
    warrantyRecords(string c = "noname") : Name(c) { }

    void AddtoList(itemWarranty* d, string saleCode) {
        itemlist[saleCode] = d;
        cout << "The item has been registered successfully!";
    }

    itemWarranty* getWarrantyInfo(string OrderCode) {
        return itemlist[OrderCode];
    }

private:
    string Name;
    std::map < string, itemWarranty* > itemlist;
};

ostream& operator<<(ostream& ostr, const laptop& s) {
    ostr << "Laptop: " << s.brand << " " << s.model << "\nRam: " << s.ram << "GB\nProcessor: "
    << s.cp << "\nStorage: " << s.storage << "GB\nPrice: €" << s.price << endl;
    return ostr;
}

laptop::laptop(const laptop& s) {
    brand = s.brand; model = s.model; ram = s.ram;    cp = s.cp; storage = s.storage;
    price = s.price; count++;
}

//Inheritance WITH BASIC ASSOCIATION
class stakeHolder { //abstract base class
public:
    //pure virtual member functions
    virtual void printDetails() = 0;
    virtual string getInfo() = 0;
};

class Supplier : public stakeHolder { //abstract base class - can inherit all member and member
functions of the base class.
public:
    //to turn this into a derived class, virtual functions must be implemented inside the code
    Supplier(string tempComp = "Default", string empTemp = "STAFF") : companyName(tempComp),
employeeInitial(empTemp) {} //default constructor of the base class
    string getCompanyInfo() { return companyName; } //returning the resistance value without
changes+
    string getEmployeeName() { return employeeInitial; }

protected: //using protected modifier to make members inaccessible outside the class while still
accessible by derived/friend classes/functions.
    string companyName; //Company Name
    string employeeInitial; //employee initial only
};

class staff : public Supplier { // derived class - inherits all member and member functions of the
base classes.
public:
    staff(string tempComp = "Default", string empTemp = "STAFF", string tempId = "xx") :
Supplier(tempComp, empTemp), employeeID(tempId) {} //default constructor for the resistor class to
which value is assigned to the base class
    void printDetails() { cout << "Company: " << companyName << "\nStaff Name: " <<
employeeInitial << "\n Staff ID: " << employeeID << endl; }
    string getInfo() { return companyName; }

protected:
    string employeeID; //employee id
};

class customer : public stakeHolder { // derived class - inherits all member and member functions
of the base classes.
public:
    customer(staff* tempStaff, string fn = "n/a", string ln = "n/a") : firstName(fn),
lastName(ln), seller(tempStaff) {} //default constructor
    void printDetails() { cout << "Customer: " << firstName << " " << lastName << "\nServed
by: " << (seller->getEmployeeName())<< endl; }
    string getInfo() { //get purchaser information
        string purchaser = firstName + " " + lastName;
        return purchaser;
    }

protected:
    string firstName, lastName; //customer first name and last name
    staff* seller; //knows about the staff object passed to the customer object
};

```

```

//using Association Class
class transaction {
    friend ostream& operator<<(ostream& ostr, transaction& a); //non member
    output stream operator
public:

    //constructor
    transaction() {};
    transaction(const transaction& s);
    transaction(laptop* Laptop, customer* User, date dop) : laptop_p(Laptop),
customer_p(User), dateofPurchase(dop)
    {
        int y = rand()%1000; //generate transaction reference
        transactionReference = "TUD"+ std::to_string(y);
    };
    ~transaction() //destructor
    {
    };
    //finding user - will display user name + laptop records
    string findUser(string firstName, string lastName )
    {
        return customer_p->getInfo();
    }
    string getlaptop()const { return(laptop_p->getlaptopBrand()); };
    //get user name - will return name of the person
    string getName() const { return(customer_p->getInfo()); };
    //get reference record - TO DO!!
    string getRef()const { return transactionReference; };

    //print transaction information
    void printTransaction() {
        cout <<"Transaction Reference: " <<transactionReference << endl;
        cout << "Purchase Date: " << dateofPurchase << endl;
        customer_p->printDetails(); //calling member function for class
customer
        laptop_p->printLaptopInfo(); //calling member functin for laptop
class
        cout << "\n";
    }

private:
    laptop* laptop_p;
    customer* customer_p;
    string transactionReference;
    date dateofPurchase;
};

transaction::transaction(const transaction& s) {
    customer_p = s.customer_p;
    laptop_p = s.laptop_p;
    transactionReference = s.transactionReference;
    dateofPurchase = s.dateofPurchase;
}

ostream& operator<<(ostream& os, transaction& c) { // output stream operator for
Employee
    c.printTransaction();
    return os;
}

```

```

class transactionList {
public:
    transactionList(string c = "ReferenceNA") : transactionReference(c) { }

    void AddtoList(transaction* d, string saleCode) {
        itemlist[saleCode] = d;
    }

    transaction* getTransactionInfo(string OrderCode) {
        return itemlist[OrderCode];
    }

private:
    string transactionReference;
    std::map < string, transaction* > itemlist;
};

class node
{
    friend class queue;
public:
    //node();
    node(const transaction& x) : data(x), next(NULL) {};
    node(); //member class function using function overloading

private: //private data
    transaction data;
    node* next;
};

node::node() :next(NULL) {} //definition of the member class function node
class queue
{
public:
    queue(); //class constructor
    ~queue(); //class destructor

    //Member class function definition
    queue(const queue&); //another function same with the constructor using function
overloading
    int push(transaction value); //push function
    transaction pop(void); //pop function
private: //private data
    node* listhead, * listtail;
};

queue::queue() //function definition for queue
{
    //setting listhead and listtail to null when called
    listhead = NULL;
    listtail = NULL;
}

queue::~queue() //destructor definition
{
    node* temp;
    while (listhead != NULL)
    {
        // get the address of next node
        temp = listhead;
        // move to the next node on queue
        listhead = listhead->next;
        delete temp; //delete
    }
}

```

```

int queue::push(transaction value)
{
    int error = 1;
    // get a new node
    node* temp = new node(value);
    if (temp == NULL) //queue full
    {
        cout << "queue overflow" << endl;
        error = -32000;
    }
    else
    { //temp->data = value; // NB Don't need for integers as
      // already done by constructor above
        if (listhead == NULL) //if empty
        {
            listhead = temp; //assign new node to listhead
        }
        else
        {
            listtail->next = temp; //else shift listtail
        }
        listtail = temp; //assign new node to listtail
    }
    return error;
}

transaction queue::pop(void)
{
    node* temp;
    transaction value;
    if (listhead == NULL) //queue empty
    {
        cout << "queue underflow" << endl;
        //value = -32068; //have to return a value
        // NB can't do this if queue of some other class eg student
    }
    else
    {
        // get the value at the top of the list
        value = listhead->data;
        // remove the top entry from the list
        temp = listhead;
        listhead = listhead->next;
        delete temp;
        //Check if Queue empty
        if (listhead == NULL)
            listtail = NULL; //clear listtail
    }
    return value; // return the value
}

queue::queue(const queue& r)
{
    node* hold = r.listhead;
    node* temp, * oldtemp;
    if (hold == NULL) //queue empty
    {
        listhead = NULL;
        listtail = NULL;
    }
    else
    {
        temp = new node;
        listhead = temp;
        while (hold != NULL)
        {
            // get the value at the node looking at on queue 'c' list
            temp->data = hold->data;
            // move to the next entry on 'c' list
            hold = hold->next;
            oldtemp = temp;
            if (hold != NULL)
            {
                temp = new node;
                oldtemp->next = temp; }
        }
        listtail = temp;
    }
}

```

```

int main() {
    staff staff1("Apple", "Kerry", "XD12"); //creating a staff object
    string refStorage[100], warrantyRef[100];
    int menuOptionTransaction = 0, menuOption = 0;
    transactionList salesList("start");
    warrantyRecords warrantyList("start");
    int itemWarrantyCount = 0, transactionCount = 0;
    string transactRef[100];
    stakeHolder* Salesprofile;
    queue transactionQueue;

    /*available laptops*/
    laptop laptop1("Dell", "MXP32", 16, "Intel I7", 512, 1099.99);
    laptop laptop2("HP", "EliteBook 840", 8, "Intel I5", 512, 629.00);
    laptop laptop3("Lenovo", "IdeaPad 750", 8, "Intel I5", 256, 674.99);
    laptop laptop4("Lenovo", "ThinkPad X1", 16, "AMD 3015e", 512, 749.50);
    laptop laptop5("Lenovo", "IdeaPad Flex", 8, "Intel I3", 128, 400.00);

    /*---Creating necessary objects to satisfy transaction class constructor-- */

    date saleDate(26, 04, 2020); //creating a date object for purchase date
    date expiryDate(26, 05, 2020); // creating a date object for warranty date

    transaction transactionTemp1;

    //menu options
    do {
        cout << "===== Welcome to Curry's POS =====" << "\n\nSelect from
the menu" << endl;
        cout << "[1]: New Sale [2] View Transactions [3] Check Transaction References [4]
Check Warranty References ";

        int y = 0;
        cout << "\n\nSelect a value from the menu: "; cin >> y;
        system("CLS");
        //add new transaction
        if (y == 1) {
            do {
                /*entering Date*/
                string dd, mm, yy;
                int day, month, year;
                cout << "Enter Transaction date: ";
                cin >> dd; cin >> mm; cin >> yy;
                day = stoi(dd);
                month = stoi(mm);
                year = stoi(yy);
                date* saleDate1= new date(day, month, year);
                saleDate = *saleDate1;

                system("CLS");

                /*Displaying all the laptops on sale with specs and price*/
                cout << "-----List of Laptops on sale ----- \n\n[1]";
                laptop1.printLaptopInfo();
                cout << "\n" << endl;
                cout << "[2]";
                laptop1.printLaptopInfo();
                cout << "\n" << endl;
                cout << "[3]";
                laptop2.printLaptopInfo();
                cout << "\n" << endl;
                cout << "[4]";
                laptop3.printLaptopInfo();
                cout << endl;

                //getting Laptop selection
                string selection;
                cout << "\n\nSelect a Laptop: ";
                cin >> selection;
                int selectIn = stoi(selection);
                system("CLS");
            } while (y != 0);
        }
    } while (y != 0);
}

```

```

string firstN, lastN;
cout << "Enter Customer Name: ";
cin >> firstN; cin >> lastN;
customer i(&staff1, firstN, lastN);
Salesprofile = &i;

//Getting warranty subscription info (default warranty is 2 years)
cout << "Please enter device warranty (years of warranty): ";
string tempWarr;
cin >> tempWarr;
int warrantySubs = stoi(tempWarr);

//generateReference
string tempRef, tempRefWarr;

system("CLS");
/*IF else statement below refers to the selection of laptop i.e. if 1 is
selected it will choose to select laptop1 object class*/
if (selectIn == 1) {
    tempRef = referenceGenerator(1); //generate transaction reference
    itemWarranty warrnt(saleDate, &laptop1, warrantySubs); //create new
    transaction trans(&laptop1, &i, saleDate); //create new transaction
    salesList.AddtoList(&trans, tempRef); //add to list map
    tempRefWarr = referenceGenerator(2); //generate warranty reference
    warrantyList.AddtoList(&warrnt, tempRefWarr);
    transactRef[transactionCount] = tempRef; //store transaciton reference
    warrantyRef[itemWarrantyCount] = tempRefWarr; //store warranty
    itemWarrantyCount++;
    transactionCount++;
    transactionQueue.push(trans); //push to queue
    cout << "\n\nTransactionID: " << tempRef << "\nWarrantyID: " <<
    Salesprofile->printDetails(); //display transaction info for sale
}
else if (selectIn == 2) {
    tempRef = referenceGenerator(1);
    itemWarranty warrnt(saleDate, &laptop2, warrantySubs);
    transaction trans(&laptop2, &i, saleDate);
    salesList.AddtoList(&trans, tempRef);
    tempRefWarr = referenceGenerator(2);
    warrantyList.AddtoList(&warrnt, tempRefWarr);
    transactRef[transactionCount] = tempRef;
    warrantyRef[itemWarrantyCount] = tempRefWarr;
    itemWarrantyCount++;
    transactionCount++;
    transactionQueue.push(trans);
    cout << "\n\nTransactionID: " << tempRef << "\nWarrantyID: " <<
    Salesprofile->printDetails();
}
else if (selectIn == 3) {
    tempRef = referenceGenerator(1);
    itemWarranty warrnt(saleDate, &laptop3, warrantySubs);
    transaction trans(&laptop3, &i, saleDate);
    salesList.AddtoList(&trans, tempRef);
    tempRefWarr = referenceGenerator(2);
    warrantyList.AddtoList(&warrnt, tempRefWarr);
    transactRef[transactionCount] = tempRef;
    warrantyRef[itemWarrantyCount] = tempRefWarr;
    itemWarrantyCount++;
    transactionCount++;
    transactionQueue.push(trans);
    cout << "\n\nTransactionID: " << tempRef << "\nWarrantyID: " <<
    Salesprofile->printDetails();
}
else if (selectIn == 4) {
    tempRef = referenceGenerator(1);
    itemWarranty warrnt(saleDate, &laptop4, warrantySubs);
    transaction trans(&laptop4, &i, saleDate);
    salesList.AddtoList(&trans, tempRef);
    tempRefWarr = referenceGenerator(2);
    warrantyList.AddtoList(&warrnt, tempRefWarr);
    transactRef[transactionCount] = tempRef;
    warrantyRef[itemWarrantyCount] = tempRefWarr;
    itemWarrantyCount++;
    transactionCount++;
    transactionQueue.push(trans);
    cout << "\n\nTransactionID: " << tempRef << "\nWarrantyID: " <<
    Salesprofile->printDetails();
    //~trans();
}
else {
    cout << "Entered value invalid" << endl;
}
}

```

```

string addNew;

        cout << "\n\nAdd another transaction? Y/N: "; cin >> addNew;

        if (addNew == "Y" || addNew == "y" || addNew == "yes" || addNew == "YES") {
            menuOption = 0;
        }
        else {
            menuOptionTransaction = 1;
        }
        //system("CLS");
    } while (menuOptionTransaction == 0);
} //view all transaction
else if (y == 2) {

    for (int i = 0; i < transactionCount; i++) {
        transactionTemp1 = transactionQueue.pop(); //Calling the queue's pop function
        //which returns data inside listhead and assign to Student1 object
        transactionTemp1.printTransaction(); //print values inside transaction

        cout << "\n\n";
    }
} //Check transactions
else if (y == 3) {

    int transactionMenu1 = 0;
    do {
        cout << "[1] View Transaction References [2] Search Transaction Details";
        string tempSelect;
        cout << "\nEnter value from the menu: "; cin >> tempSelect;
        int selectVal = stoi(tempSelect);
        //system("CLS");
        //view references
        if (selectVal == 1) {
            //transactionCount = 10;
            if (transactionCount > 0) {
                cout << "-----Transaction References ----- \n";
                for (int temp = 0; temp < transactionCount; temp++) {
                    cout << "Trasaction:" << temp << ":" <<
transactionRef[temp] << endl; //print all recorded transaction references
                }
            }
            else {
                cout << "\n---No records found---\n";
            }
        }
        else { //search transaction
            string tempRef1;
            cout << "===== Checking Transaction Details =====" << "\n\nEnter
Transaction Reference";

            cin >> tempRef1;
            cout << "Transaction References";
            for (int temp = 0; temp < transactionCount; temp++) {
                //cout << warrantyRef[temp] << endl;
                transaction* c = salesList.getTransactionInfo(tempRef1);
                //search transaction using transaction reference
                cout << *c; //display transaction
            }
        }
    } while (transactionMenu1 == 0);

    string MenuRotation1;
    cout << "Would you like to view more
Transaction Records? Y/N "; cin >> MenuRotation1;
    if (MenuRotation1 == "Y" || MenuRotation1 == "y" || MenuRotation1 == "yes" ||
MenuRotation1 == "YES") {
        transactionMenu1 = 0;
    }
    else {
        transactionMenu1 = 1;
    }
} while (transactionMenu1 == 0);

}
else if (y == 4) {

    int warrantyMenu = 0;
    do {
        //Menu for warranty record features
        cout << "[1] View Warranty References [2] Check Item Warranty Details";
        string tempWarrSelect;
        cin >> tempWarrSelect;
        int selectVal = stoi(tempWarrSelect);
        if (selectVal == 1) { //view recorded warranty references
            cout << "List of Item Warranty References\n\n";
            if (transactionCount > 0) {
                for (int temp = 0; temp < transactionCount; temp++) {
                    cout << "Trasaction:" << temp << ":" << warrantyRef[temp] << endl; }
                }
            }
            else {
                cout << "No warranties recorded";
            }
        }
    } while (warrantyMenu == 0);
}
}

```

```

else { //Search warranty details using a reference
    string tempWarrRef;
    cout << "==== Checking Warranty Details =====>> <<
    "\n\nEnter Item Warranty Reference: ";
    cin >> tempWarrRef;

    cout << "Warranty References\n" ;
    for (int temp = 0; temp < transactionCount; temp++) {
        cout << "Trasaction:" << temp << ":" <<

warrantyRef[temp] << "\n";
        itemWarranty* c =
warrantyList.getWarrantyInfo(tempWarrRef);
        cout << *c;
    }

    string MenuRotation1;
    cout << "Would you like to view
more Warranty Records? Y/N :"; cin >> MenuRotation1;
    if (MenuRotation1 == "Y" || MenuRotation1 == "y" || MenuRotation1
== "yes" || MenuRotation1 == "YES") {
        warrantyMenu = 0;
    }
    else {
        warrantyMenu = 1;
    }

    } while (warrantyMenu == 0);
}

else {
    cout << "Entered Value is incorrect"; //do nothing
    menuOption = 1;
}

string MenuRotation;
cout << "Would you like to go back to the main
Menu? Y/N :"; cin >> MenuRotation;
    if (MenuRotation == "Y" || MenuRotation == "y" || MenuRotation == "yes" ||
MenuRotation == "YES") {
        menuOption = 0;
    }
    else {
        menuOption = 1;
    }

    system("CLS");
} while (menuOption == 0);

return 0;
}

string referenceGenerator(int y) {
    if (y == 1) {

        string x = "POSDX" + to_string(rand()%1000);
        return x;
    }
    else {
        string x = "SDA" + to_string(rand() % 1000*2);
        return x;
    }
}

```