UTM
UNIVERSITI TEKNOLOGI MALAYSIA

FACULTY OF COMPUTING
UTM Johor Bahru

Semester II 2023/2024

Subject : SECJ1023 Programming Technique II

Section : 08

Task : Project Final

Lecturer: Dr. Lizawati binti Mi Yusuf

Group 7 : 3Q

Student Name	Matric Number
Tan Keqin	A23CS0184
Mavis Lim Hui Qing	A23CS0110
Chong Lun Quan	A23CS0067

Section A

Project Description

Nowadays, individuals encounter myriads of challenges to manage their finances well. From tracking their spending and analyzing their investment portfolio. The Personal Finance Manager system is intended to help individuals manage their finances effectively by tracking the users' transactions and creating reports to provide a clearer view of their monthly financial activities.

Objectives and/or Purpose

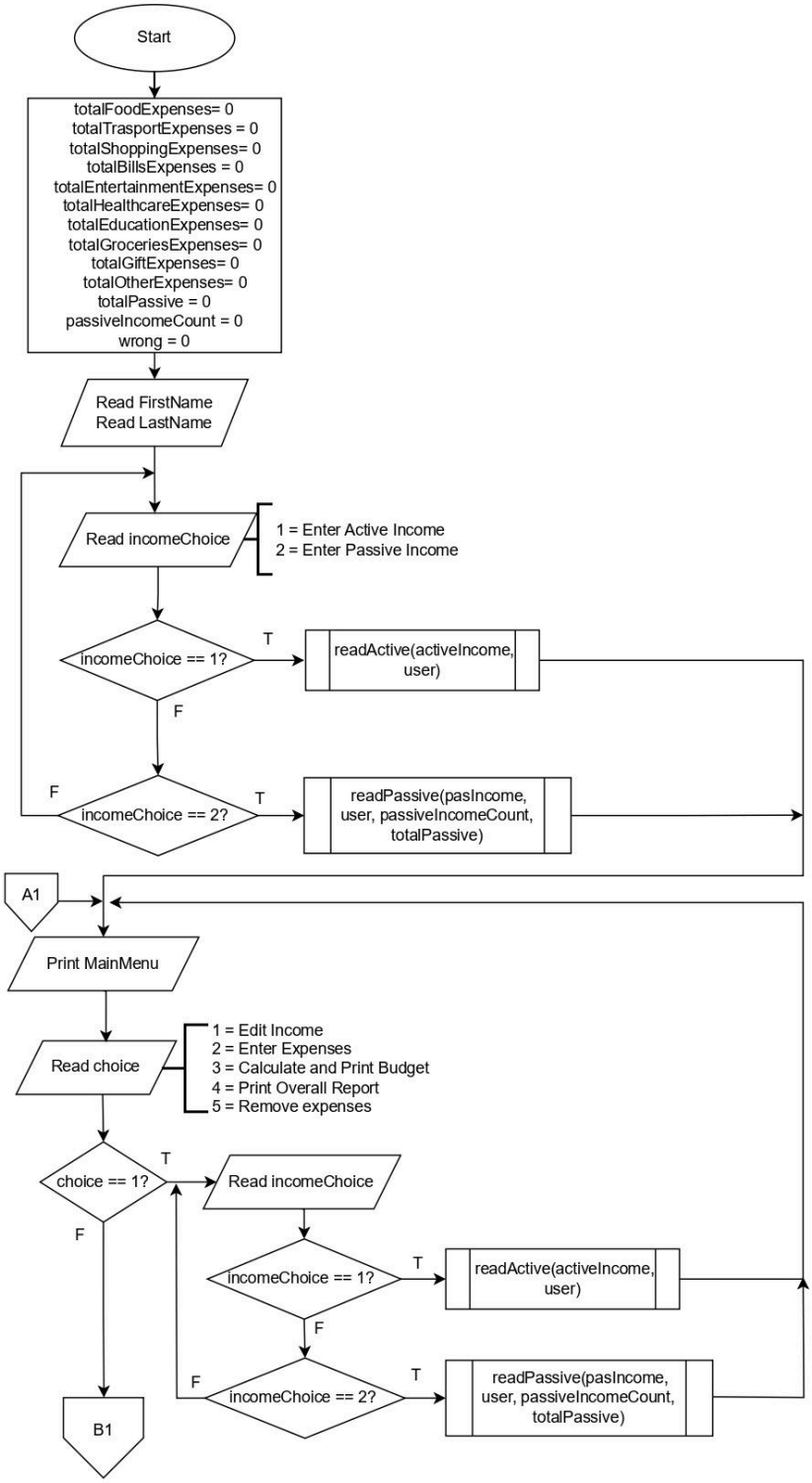
1. To record users' income which can be divided into passive income and active income

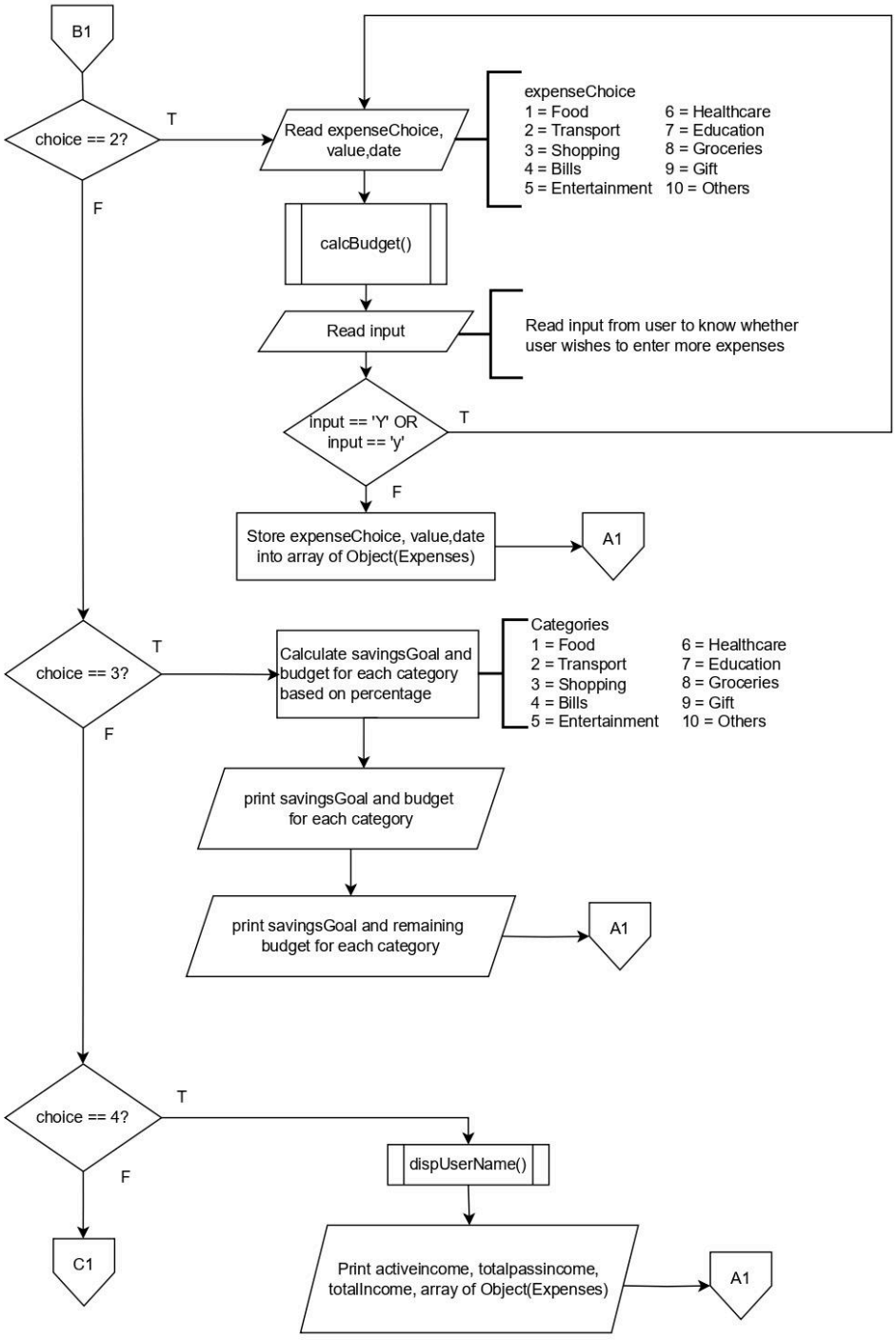
2. To record the users' expenses of different categories such as food, shopping, transportation and so on.

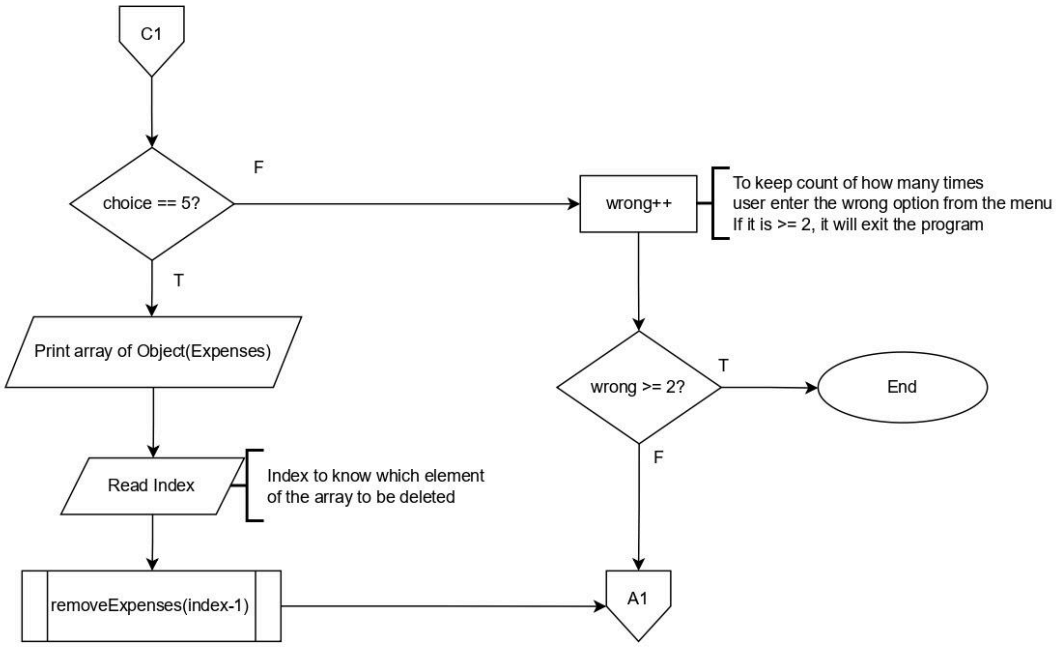
3. To help users plan their budget to maintain a positive cash flow.

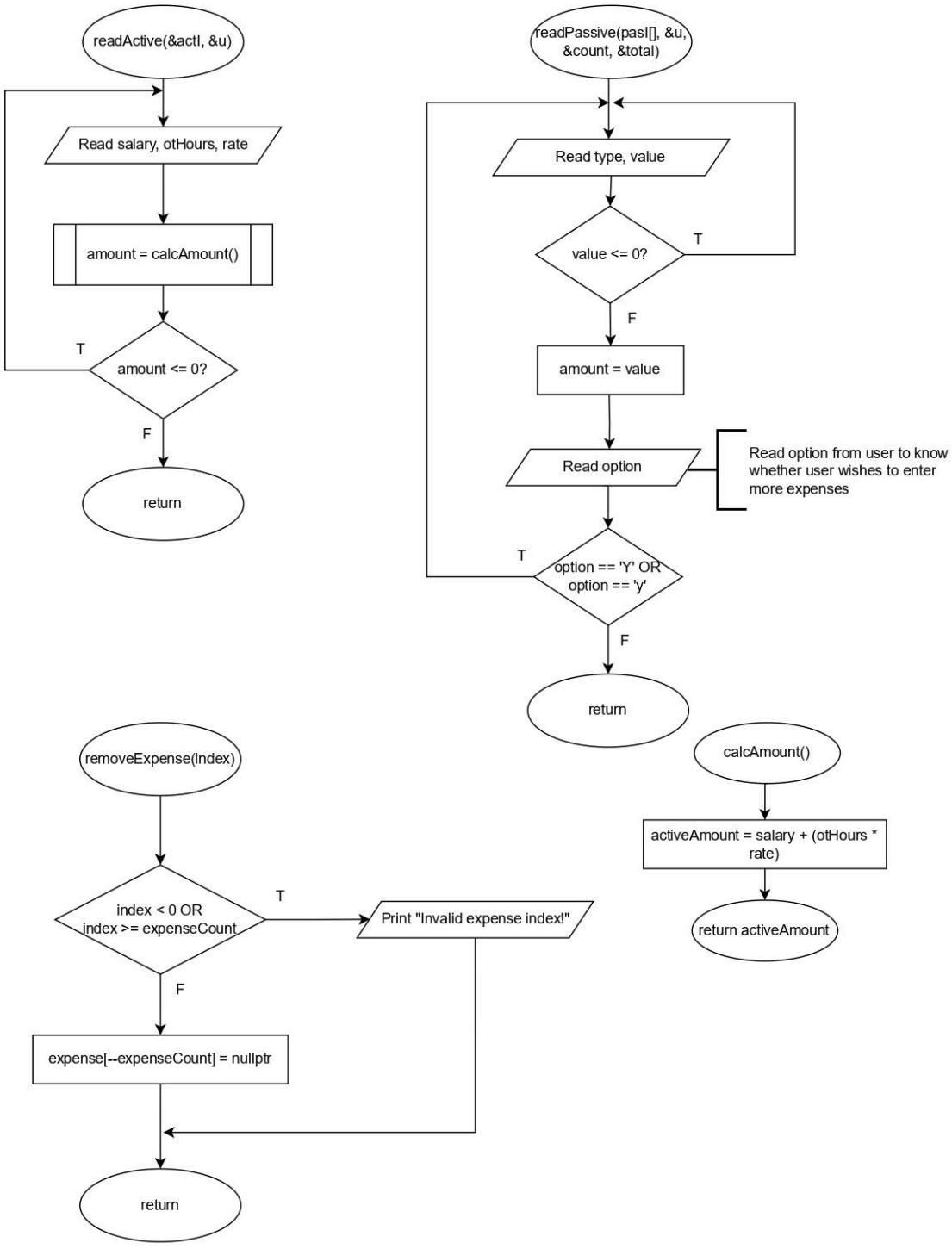
Analysis and design

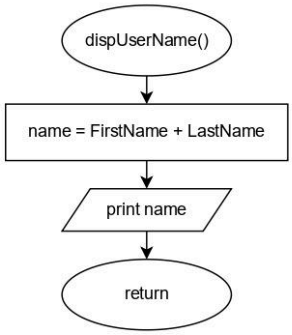
- Flow Chart

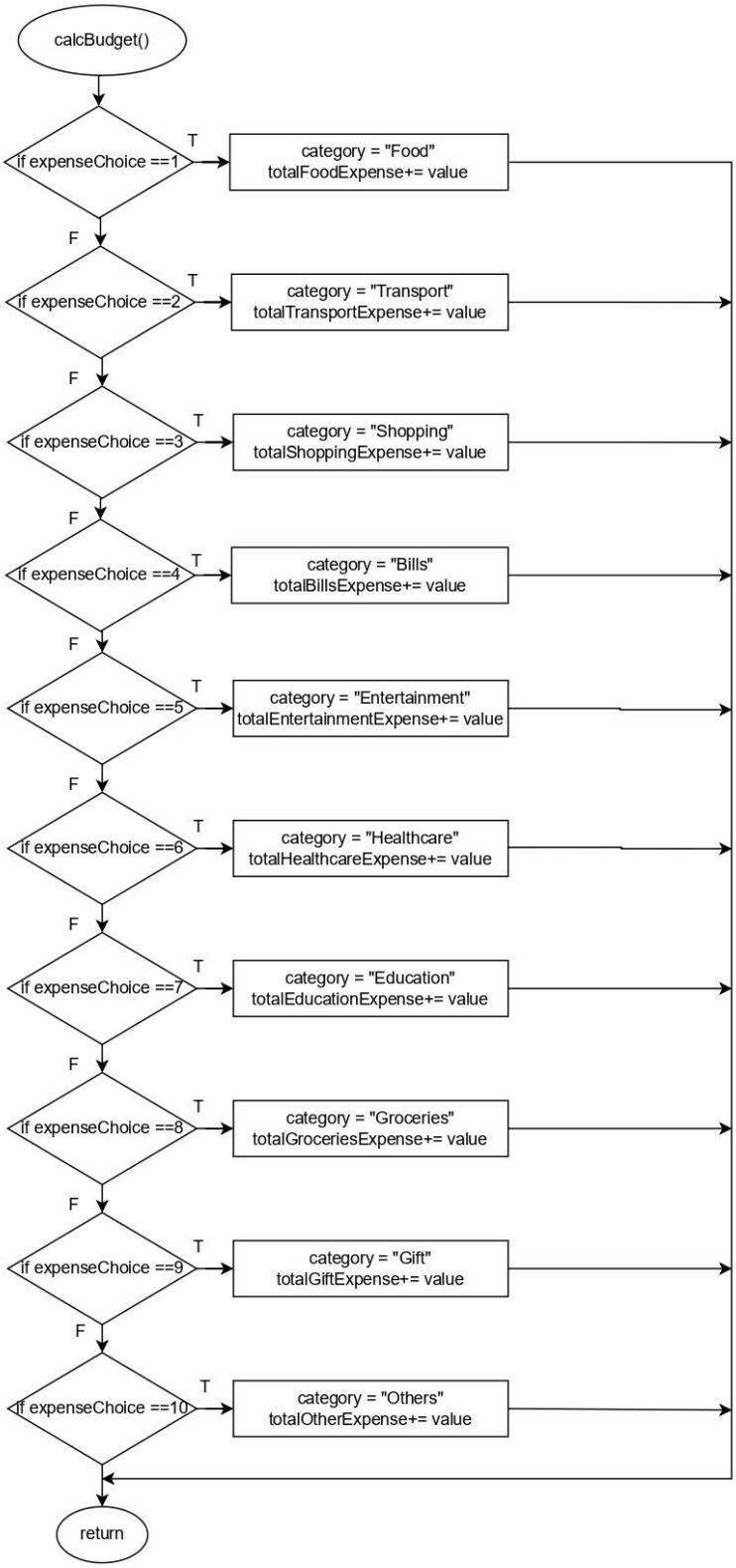



B1
choice == 2?
T
F
Read expenseChoice, value,date
expenseChoice
1 = Food
2 = Transport
3 = Shopping
4 = Bills
5 = Entertainment
6 = Healthcare
7 = Education
8 = Groceries
9 = Gift
10 = Others
calcBudget()
Read input
Read input from user to know whether user wishes to enter more expenses
input == 'Y' OR input == 'y'
T
F
Store expenseChoice, value,date into array of Object(Expenses)
A1
choice == 3?
T
F
Calculate savingsGoal and budget for each category based on percentage
Categories
1 = Food
2 = Transport
3 = Shopping
4 = Bills
5 = Entertainment
6 = Healthcare
7 = Education
8 = Groceries
9 = Gift
10 = Others
print savingsGoal and budget for each category
print savingsGoal and remaining budget for each category
A1
choice == 4?
T
F
dispUserName()
C1
Print activeincome, totalpassincome, totalIncome, array of Object(Expenses)
A1

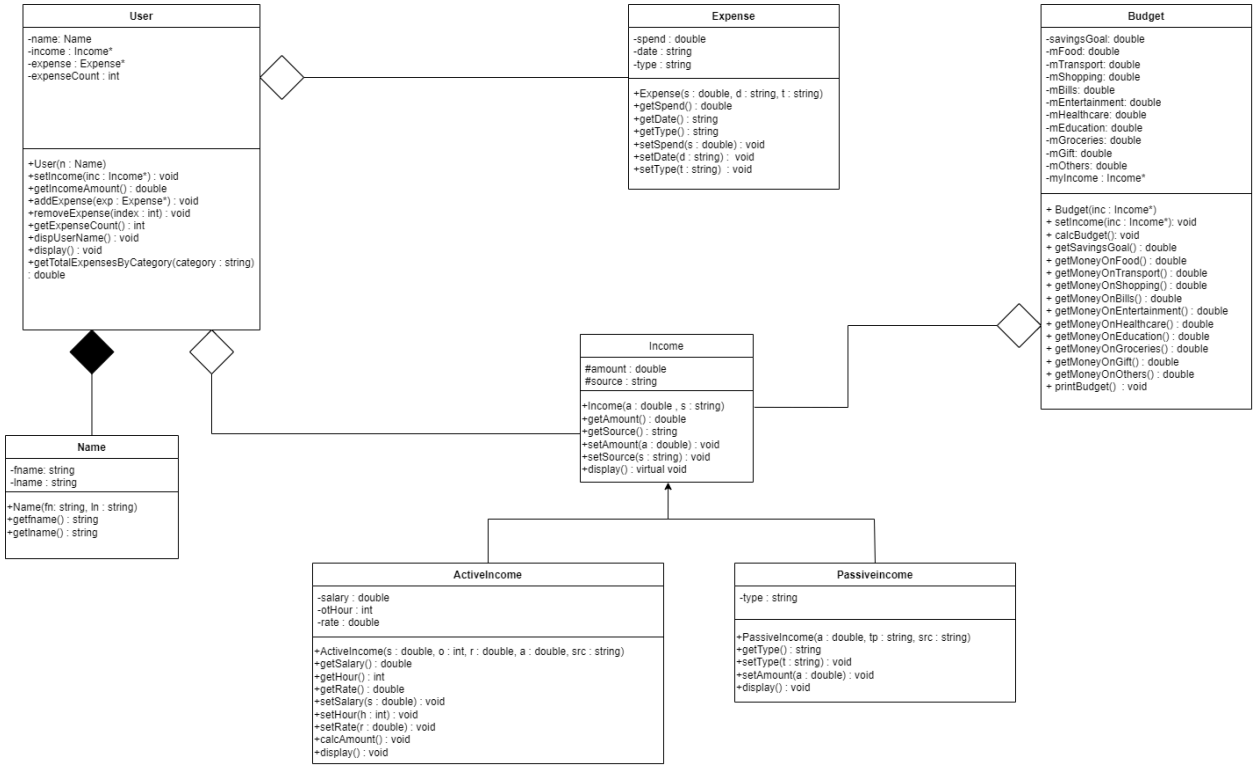

C1
choice == 5?
F
T
wrong++
To keep count of how many times
user enter the wrong option from the menu
If it is >= 2, it will exit the program
Print array of Object(Expenses)
wrong >= 2?
T
End
F
Read Index
Index to know which element
of the array to be deleted
removeExpenses(index-1)
A1


readActive(&actI, &u)
Read salary, otHours, rate
amount = calcAmount()
amount <= 0?
T
F
return
readPassive(pasI[], &u, &count, &total)
Read type, value
value <= 0?
T
F
amount = value
Read option
Read option from user to know whether user wishes to enter more expenses
option == 'Y' OR option == 'y'
T
F
return
removeExpense(index)
index < 0 OR index >= expenseCount
T
Print "Invalid expense index!"
F
expense[--expenseCount] = nullptr
return
calcAmount()
activeAmount = salary + (otHours * rate)
return activeAmount


dispUserName()
name = FirstName + LastName
print name
return


calcBudget()
if expenseChoice ==1
T
category = "Food"
totalFoodExpense+= value
F
if expenseChoice ==2
T
category = "Transport"
totalTransportExpense+= value
F
if expenseChoice ==3
T
category = "Shopping"
totalShoppingExpense+= value
F
if expenseChoice ==4
T
category = "Bills"
totalBillsExpense+= value
F
if expenseChoice ==5
T
category = "Entertainment"
totalEntertainmentExpense+= value
F
if expenseChoice ==6
T
category = "Healthcare"
totalHealthcareExpense+= value
F
if expenseChoice ==7
T
category = "Education"
totalEducationExpense+= value
F
if expenseChoice ==8
T
category = "Groceries"
totalGroceriesExpense+= value
F
if expenseChoice ==9
T
category = "Gift"
totalGiftExpense+= value
F
if expenseChoice ==10
T
category = "Others"
totalOtherExpense+= value
return

- UML Class Diagram



Section B

- Encapsulation

  Encapsulation is the process of combining attributes and methods in one package and hiding the implementation of the data from the user of the object. Our system has 7 classes: ‘Expense’, ‘User’, ‘Name’, ‘Income’, ‘ActiveIncome’, ‘PassiveIncome’ and ‘Budget’.

  1) class Expense

     Private attributes:
     -double spend
     -string date
     -string type
     Direct access to these attributes is restricted by making them private.

     Public methods:
     -Expense(double s, string d, string t) : This is the constructor of the class
     -double getSpend()
     -string getDate()
     -string getType()
     -setSpend(double s)
     -setDate(double d)
     -setType(double t)
     Access and modification can be done through public getter and setter methods.

  2) class User

     Private attributes:
     - Name name
     - Income *income
     - Expense *expense
     - int expenseCount

     Public methods:
     - User(Name n) : This is the constructor of the class
     - void setIncome(Income *inc)
     - double getIncomeAmount()
     - void addExpense(Expense *exp)
     - void removeExpense(int index)
     - int getExpenseCount()
     - void dispUserName()
     - void display()
     - double getTotalExpenseByCategory(string category)

3) class Name
   Private attributes:
   - string fname
   - string lname

   Public methods:
   - Name(string fn, string ln)
   - string getfname()
   - string getlname()

4) class Income
   Protected attributes:
   - double amount
   - string source

   Public methods:
   - Income(double a, string s) : This is the constructor of the class
   - double getAmount()
   - string getSource()
   - void setAmount(double a)
   - void setSource(string s)
   - virtual void display()

5) class ActiveIncome
   Private attributes:
   - double salary
   - int otHour
   - double rate

   Public methods:
   - ActiveIncome(double s, int o, double r, double a, string src) : This is the constructor of the class
   - double getSalary()
   - int getHour()
   - double getRate()
   - void setSalary(double s)
   - void setHour(int h)
   - void setRate(double r)
   - void calcAmount()
   - void display()

6) class PassiveIncome
   Private attributes:
   - string type

   Public methods:
   - PassiveIncome(double a, string tp, string src) : This is the constructor of the class
   - string getType()
   - void setType(string t)
   - void setAmount(double a)
   - void display()

7) class Budget
   Private attributes:
   - double savingsGoal
   - double mFood
   - double mTransport
   - double mShopping
   - double mBills
   - double mEntertainment
   - double mHealthcare
   - double mEducation
   - double mGroceries
   - double mGift
   - double mOthers
   - Income *myIncome

   Public methods:
   - Budget(Income *inc)
   - void setIncome(Income *inc)
   - void calcBudget()
   - double getSavingsGoal()
   - double getMoneyOnFood()
   - double getMoneyOnTransport()
   - double getMoneyOnShopping()
   - double getMoneyOnBills()
   - double getMoneyOnEntertainment()
   - double getMoneyOnHealthcare()
   - double getMoneyOnEducation()

- double getMoneyOnGroceries()
- double getMoneyOnGift()
- double getMoneyOnOthers()
- void printBudget()

- Composition

  Composition is a restricted version of aggregation in which the enclosing and enclosed objects are highly dependent on each other. In our system, we are implementing this concept in

  1) User consists of Name

  - 'User' contains 'Name'. This means that 'Name' objects are part of the 'User', and if the 'User' object is destroyed, its 'Name' objects are also destroyed. A 'User' must have a 'Name', else the user is not existing.

```
class User {
    Name name; //Composition
    Income *income;
    Expense *expense[100];
    int expenseCount;
```

- Aggregation

  Aggregation is a special type of association which is a one way relationship. The existence of the objects are independent.

  1) User has an Income

  - 'User' and 'Income' are independent where a 'User' may not have 'Income'. The 'User' class uses 'Income' to get the user's income details. The 'Income' object can be shared or changed without affecting the 'User' class.

```
class User {
    Name name;
    Income *income;
    Expense *expense[100]; //Aggregation
    int expenseCount;
```

  2) Budget has an Income

- ‘Budget’ and ‘Income’ are independent where the budget may not have ‘Income’. The ‘Budget’ class uses the ‘Income’ class to calculate the budget. If the ‘Budget’ object is destroyed, ‘Income’ object remains unaffected. A ‘User’ that has an income does not necessarily have a budget.

```
class Budget {
    double savingsGoal;
    double mFood;
    double mTransport;
    double mShopping;
    double mBills;
    double mEntertainment;
    double mHealthcare;
    double mEducation;
    double mGroceries;
    double mGift;
    double mOthers;
    Income *myIncome; //Aggregation
```

3) User has an Expense

- ‘User’ has a collection of expense objects. ‘User’ and ‘Expense’ are independent where the existence of an expense object is independent of the User. A User may or may not have expenses. The ‘User’ object can exist without any Expense objects, and Expense objects can exist or be shared without depending on the User.

```
class User {
    Name name;
    Income *income; //Aggregation
    Expense *expense[100];
    int expenseCount;
```

- Inheritance

  Inheritance is the ability of one class to extend the capabilities of another. It occurs when we have many classes that are related to each other by inheritance.

  - Parent Class : Income

    We chose Income as the parent class because it contains the attributes amount and source where they are common to all types of income.

```
class Income {
protected:
    double amount;
    string source;
public:
    Income(double a = 0, string s = "");
    double getAmount() const;
    string getSource() const;
    void setAmount(double a);
    void setSource(string s);
    virtual void display() const;
};
```

- Child Classes :
    - ActiveIncome

      ActiveIncome is a specific type of income that comes with additional attributes and methods that are unique. It inherits the Income to gain the common properties but also adds its specific attributes and methods. We update the amount in the Parent Class using the attributes inside the derived class(ActiveIncome) such as salary, otHours, and rate.

```
class ActiveIncome : public Income {
    double salary;
    int otHour;
    double rate;
public:
    ActiveIncome(double s = 0, int o = 0, double r = 0, double a = 0.0, string src = "Active Income");
    double getSalary() const;
    int getHour() const;
    double getRate() const;
    void setSalary(double s);
    void setHour(int h);
    void setRate(double r);
    void calcAmount();
    void display() const;
};
```

    - Passiveincome

      PassiveIncome is a specific type of income that comes with additional attributes and methods that are unique. It inherits the Income to gain the common properties but also adds its own specific attributes and methods. We can display the amount in the Parent Class (that is updated in the readPassive method using a for loop to calculate the total passive income).

```
class PassiveIncome : public Income {
    string type;
public:
    PassiveIncome(double a = 0.0, string tp = "", string src = "Passive Income");
    string getType() const;
    void setType(string t);
    void setAmount(double a);
    void display() const;
};
```

- Polymorphism

  Polymorphism is the ability of objects to perform the same actions differently. It is a concept that extends from inheritance. In our system, we have a parent class Income, and child classes PassiveIncome and ActiveIncome. In the parent class Income, we have a display method, and the same method is also defined in the child classes PassiveIncome and ActiveIncome. We apply this concept by adding the 'virtual' keyword to the display method in the Income class, which allows derived classes to override this method to provide their own specific implementation.

```
class Income {
protected:
    double amount;
    string source;
public:
    Income(double a = 0, string s = "");
    double getAmount() const;
    string getSource() const;
    void setAmount(double a);
    void setSource(string s);
    virtual void display() const; //Polymorphism
};
```

- Array of objects

  We use an array of objects to store similar data items. For example, in the main function, we declare an array pasIncome with a maximum size of 100 to store up to 100 user's passive income. Additionally, within the User class, we declare an array of pointers expense with a maximum size of 100 to store pointers to Expense objects. Both of the objects have different functions where the pasIncome array is declared in the main

function for object initialization, while the expense array is declared in the User class to show aggregation.

1) Array to store list of passive income

```
int main() {
    Income income;
    ActiveIncome activeIncome;
    PassiveIncome passiveIncome;
    PassiveIncome pasIncome[100]; //Array of Object
```

```
do{
    cout << "What is the passive income type : ";
    cin >> type;
    cout << "Enter passive income amount     : ";
    cin >> value;

    while(value<=0){
        cout << "***Please re-enter your passive income and make sure it is more than 0***\n\n";
        cout << "Enter passive income amount     : ";
        cin >> value;
    }

    pasI[count].setType(type);
    pasI[count].setAmount(value);
    count++;
    total+=value;
    clearScreen();
    cout << "Do you wish to continue entering your passive income? (press Y/y for yes) : ";
    cin >> input;
    cout << endl;
}while(input == 'y' || input == 'Y');
```

2) Array to store list of expenses

In the User class :

```
class User {
    Name name;
    Income *income;
    Expense *expense[100];
    int expenseCount;
```

```
void User::addExpense(Expense* exp) {
    if (expenseCount < 100) {
        expense[expenseCount++] = exp;
    } else {
        cout << "Expense limit reached!" << endl;
    }
}
```

In the main function :

```
user.addExpense(new Expense(value, date, category));
```

- Exception handling

  When executing C++ code, different errors can occur, coding errors made by programmers due to wrong input or other unforeseeable things. Exception handling is important to make sure the system is able to deal with errors that occur when we run the system. In our system, we mainly apply this concept when there is an error and to display an error message by using the concept of try, throw and catch.

  1) Exception to verify the date format

```
try {
    if (isValidDateFormat(date) == false) {
        throw "Invalid date format!\n";
    }
}
catch (const char *msg) {
    cout << msg;
}
```

  2) Exception to verify the index entered

```
try {
    if (index > 0 && index <= user.getExpenseCount()) {
        user.removeExpense(index - 1);
        throw "Expense removed successfully.\n";
    }
    else
        throw "Invalid index.\n";
}

catch (const char *msg) {
    cout << msg;
}
```