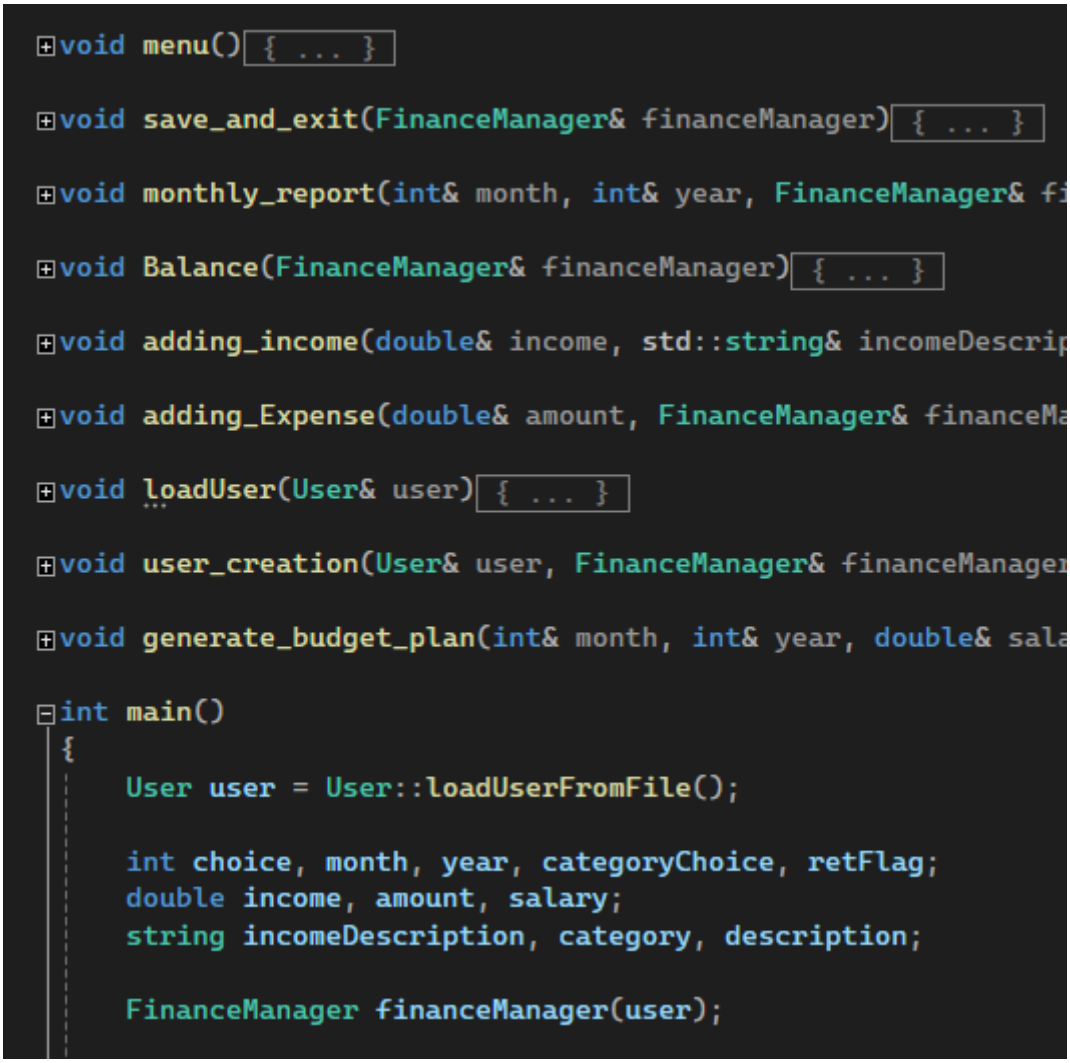
Clean Code Development Cheat Sheet

1. Meaningful Names

Use function, variable and class names that describe their operation by name, making it easier to navigate through the code. It is important to avoid one-letter names or ambiguous abbreviations.

Example in the code:

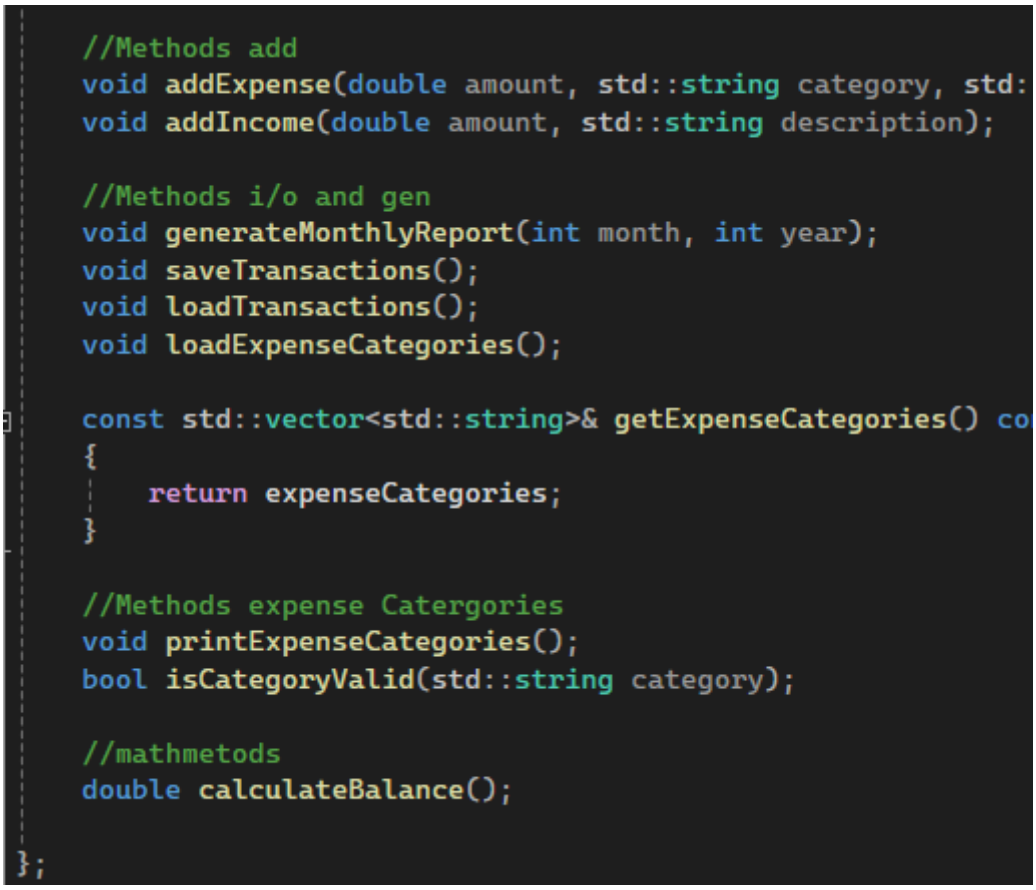


2. Comments

Try to write your code to avoid the need to add comments, the code should be largely self-describing. You can use comments to separate specific areas to

make it easier to find. If there is a need to use comments, use them to explain non-obvious or complex decisions or logic.

Example in the code:



3. Functions

Remember to make functions/methods small and focused on a specific responsibility (Single Responsibility Principle). It is important to aim for a function to have no more than 20 lines of code, of course this is not always possible but it is important to keep this in the back of your mind. Use a nomenclature that explains the function itself.

Example in the code:

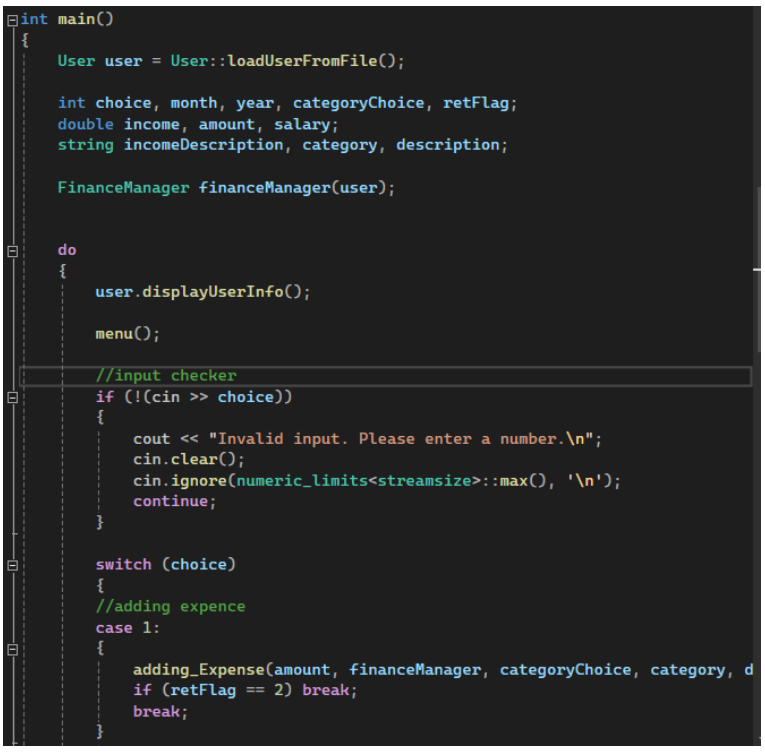
```cpp
void menu()
{
    cout << "\nPersonal Finance Manager Menu:\n";
    cout << "1. Add Expense\n";
    cout << "2. Add Income\n";
    cout << "3. Calculate Balance\n";
    cout << "4. Generate Monthly Report\n";
    cout << "5. Budget Plan\n";
    cout << "6. Create User\n";
    cout << "7. Save and Quit\n";
    cout << "Enter your choice: ";
}

void save_and_exit(FinanceManager& financeManager)
{
    financeManager.saveTransactions();
    cout << "Transactions saved. Goodbye!\n";
}
```

4. Formatting

Use consistent formatting and adhere to the chosen style. Try to keep any tabs or spaces the same across all files. At the same time, try to keep the code readable

Example in the code:



```cpp
int main()
{
    User user = User::loadUserFromFile();

    int choice, month, year, categoryChoice, retFlag;
    double income, amount, salary;
    string incomeDescription, category, description;

    FinanceManager financeManager(user);


    do
    {
        user.displayUserInfo();

        menu();

        //input checker
        if (!(cin >> choice))
        {
            cout << "Invalid input. Please enter a number.\n";
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            continue;
        }

        switch (choice)
        {
        //adding expence
        case 1:
        {
            adding_Expense(amount, financeManager, categoryChoice, category, d
            if (retFlag == 2) break;
            break;
        }
```

5. Error Handling

Use exceptions only in exceptional situations, do not use them to call up information in the console that a function has worked or is working. When using exceptions, provide clear information about what went wrong

Example in the code:

```
void Budgeting::writeToCSV(const std::string& filename, const std::string& data)
{
    std::ofstream file(filename);
    if (file.is_open()) {
        file << data;
        file.close();
    }
    else {
        std::cerr << "Error: Unable to open file " << filename << " for writing." << std::endl;
    }
```

6. STL Usage

Be familiar with and use the STL C++ Standard Library. Don't forget container classes for example vector. Try to use the functions available in them to improve the writing of algorithms, functions or methods

Example in the code:

```
private:
    User user;
    std::vector<Transaction> transactions;
    std::vector<std::string> expenseCategories;
```

7. Class Design

Follow the principle of one responsibility per class and prefer composition over inheritance. Use access specifiers such as public, private and protected to avoid unwanted use of or access to a variable/method/function

Example in the code:

```
class FinanceManager
{
private:
    User user;
    std::vector<Transaction> transactions;
    std::vector<std::string> expenseCategories;

public:

    FinanceManager(User user) : user(user)
    {
        loadTransactions();
        loadExpenseCategories();
    }

    //Methods add
    void addExpense(double amount, std::string category, std::str
    void addIncome(double amount, std::string description);

    //Methods i/o and gen
    void generateMonthlyReport(int month, int year);
    void saveTransactions();
    void loadTransactions();
    void loadExpenseCategories();
```

8. Refactoring

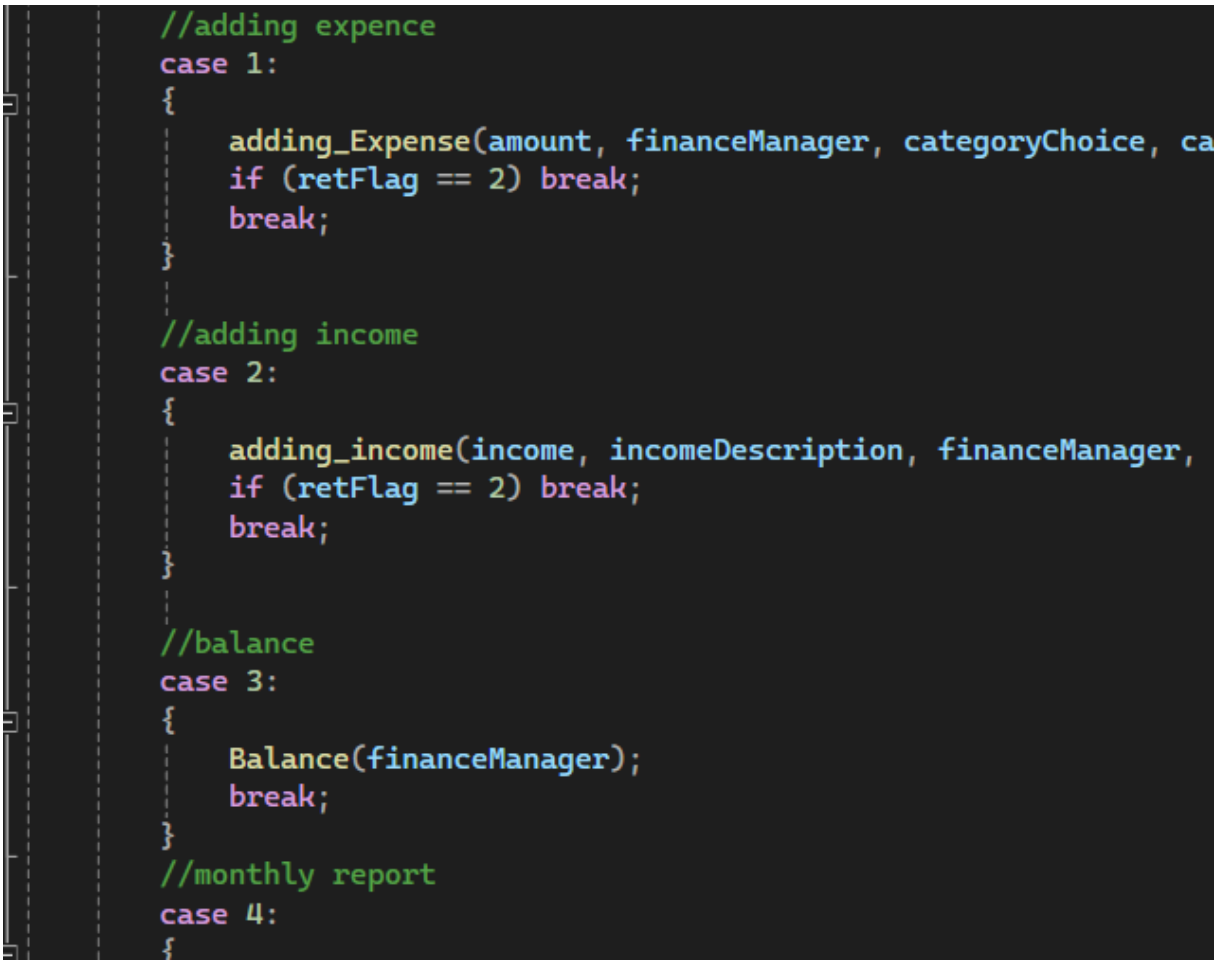
When writing code, even when we try to keep it clean we sometimes make a mess of it making the code less readable. Remember to refactor your code to improve code readability.

Example in the code:

Before

```
                case 1:
                {
-                       double amount;
-                       string category, description;
-               int categoryChoice;
-
-                       cout << "Enter expense amount: ";
-
-                       if (!(cin >> amount))
-                       {
-                               cout << "Invalid input for amount. Please enter a valid number.\n";
-                               break;
-                       }
-
-                       cin.ignore();
-
-               financeManager.printExpenseCategories();
-
-               cout << "Choose an expense category (enter the corresponding number): ";
-
-               if (!(cin >> categoryChoice) || categoryChoice < 1 || categoryChoice > financeManager.getExpenseCategories().size())
-               {
-                   cout << "Invalid category choice. Please enter a valid number.\n";
-                   break;
-               }
-
-               category = financeManager.getExpenseCategories()[categoryChoice - 1];
-
-               cin.ignore();
-
-                       cout << "Enter expense description: ";
-                       getline(cin, description);
-
-                       financeManager.addExpense(amount, category, description);
-                       cout << "Expense added successfully.\n";
```

Now



```
//adding expence
case 1:
{
    adding_Expense(amount, financeManager, categoryChoice, ca
    if (retFlag == 2) break;
    break;
}

//adding income
case 2:
{
    adding_income(income, incomeDescription, financeManager,
    if (retFlag == 2) break;
    break;
}

//balance
case 3:
{
    Balance(financeManager);
    break;
}
//monthly report
case 4:
{
```

9. Testing

For critical functionality, write unit tests to eliminate potential bugs in the code. Unit tests will support your work on the code and, before the code goes into production, help you to improve the performance of the program

Example in the code:

```cpp
#include "gtest/gtest.h"

#include "/test/Personal Finance Manager/src/PFMLib/FinanceManager.h"

TEST(FinanceManagerTest, AddExpense) { ... }

TEST(FinanceManagerTest, AddIncome) { ... }

TEST(FinanceManagerTest, CalculateBalance)
{
    User user("MyTESTER", 25, 0);
    FinanceManager financeManager(user);

    financeManager.addIncome(2000, "Salary");
    financeManager.addExpense(500, "Food", "Groceries");

    //Checker for correctness
    ASSERT_EQ(2000 - 500, financeManager.calculateBalance());
}

int main(int argc, char** argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Const-Correctness

Use const to prevent messy overwriting of data in a variable as well as to ensure its immutability. Correct use of const will help you avoid getting messy erroneous data or results.

Example in the code:

```cpp
const std::vector<std::string>& getExpenseCategories() const
{
    return expenseCategories;
}
```