**Design Specifications**

Note italics indicate a self-made and implemented class. Parenthesis indicate a primitive data type or additional grouping of information.

**How a *Currency* Enum works:**

-A *currency* can be a USD (American Dollar), CNY (Chinese Yuan or Renminbi), EUR (Euro), GBP (British Pound) or nullCurrency (only for error handling) in that order. Note only USD, CNY, EUR, and GBP are considered as actual currencies and only they are used for calculations. Note because USD is initialized to equal 1, CNY = 2, EUR = 3, GBP = 4, and nullCurrency = 5.

-The *Currency* enum is used to represent the currency of an event and the budget.

-*Currency* has overridden 2 operators: << and >>

<< cout prints an actual *Currency*’s 3 letter abbreviation. Note for some reason, none of the currency symbols (¥, €, £) besides the dollar sign ($) were able to be cout printed by Visual Studios, thus the 3-letter abbreviation was chosen to represent the currencies instead.

>> cin inputs a *Currency* using a string to represent an int for error handling against input with non-numbers. If the input is valid, then *Currency* is initialized to its corresponding numerical value using a static\_cast. If it is not, then *Currency* is initialized to nullCurrency using a static\_cast.

**How a *Spending\_Cat* Enum works:**

-A *Spending\_Cat* (spending category) can be FIXED, VARIABLE, SAVINGS, or nullSpending. Note FIXED, VARIABLE, and SAVINGS are considered as actual *Spending\_Cat*. Note because FIXED is initialized to equal 1, VARIABLE = 2, SAVINGS = 3, nullSpending = 4.

- *Spending\_Cat* enum is used to represent the Spending type of an event

-The *Spending\_Cat* has overridden 2 operators: << and >>

<< cout prints the string name of each actual *Spending\_Cat*, and prints “INVALID INPUT, MUST BE AN INTEGER (1-3)” for nullSpending. Note the numerical value of *Spending\_Cat* is used to determine which output is needed.

>> cin inputs a *Spending\_Cat* just like *Currency* – using a string to represent an int for error handling against input with non-numbers. If the input is valid, then *Spending\_Cat* is initialized to its corresponding numerical value using a static\_cast. If not, then *Spending\_Cat* is initialized to nullSpending using a static\_cast.

**How a *Month* enum works:**

-A *Month* can be JAN, FEB, MARCH, APRIL, MAY, JUNE, JULY, AUG, SEPT, OCT, NOV, DEC, or nullMonth. Note because Jan = 1, each month is initialized to its numerical order in the year, with nullMonth = 13.

- *Month* is used to represent a calendar month and is part of a larger effort to represent an *Event* date and *Budget* month, since I was not able to find any standard C++ libraries that represented a Date.

- *Month* has one function: +void PrintFullName(*Month*) that prints out the full name of each month, with the first letter capitalized and all others lowercase and uses a *Month’s* numerical value to determine which month name to actually print. The purpose of this method is for the header of the budget when << cout printed.

- *Month* has overridden 2 operators: << and >>

<< cout prints simply the capitalized letter acronym for each month and uses the month’s numerical value to determine which month specifically to print. If the *Month* to be printed is not an actual month, then “INVALID MONTH INPUT, MUST BE AN INTEGER (1-12)” is printed instead.

>> cin inputs a *Month* just like *Spending\_Cat* and *Currency* – using a string to represent an int for error handling against input with non-numbers. If the input is valid, then *Month* is initialized to its corresponding numerical value using a static\_cast. If not, then *Month* is initialized to nullMonth using a static\_cast.

**How a *Date* class works:**

-A *Date* has a: *Month* and a (int) date, #includes <Month.h>

-Its default constructor initializes a *Date* with nullMonth and a (int) date of -1. Note that realistically, this type of date is impossible.

- *Date* is part of a larger effort to represent an *Event* date, since I was not able to find any standard C++ libraries that represented a Date.

-The *Date* class has public getter and setter methods for each of the private data members.

-For getters:

+*Month* getMonth()

+int getDate()

-For setters

+void setMonth(*Month)*

+void setDate(*Month,* int) – note that a date is a combination of a *Month* and date. This method can simultaneously assign whether the input is of a valid *Month* and (int) date combination and assign *Month* and (int) date. If both *Month* and (int) date are invalid (nullMonth and -1), then they both stay that way. If *Month* is valid but (int) date is not, then they both stay that way as well.

-The *Date* class overrides 2 operators: << and >>

<< cout prints the *Month* and the (int) date. Note the use of setw() for printing (int) date. This is so that for printing *Events*, *Dates* with (int) date 0-9 and with (int) date 10-28/30/31 are printed with the same spacing.

>> cin inputs a *Date* by its private data members. Note *Month* is only initialized if the current *Month* is nullMonth, and using >> *Month,* that loop continues until *Month* is no longer nullMonth. After *Month* is initialized, a personalized month statement is printed, depending on what *Date’s* *Month* is. Then (int) date is initialized similarly to *Month, Currency,* and *Spending\_Cat*. Using a string to represent an int for error handling against input with non-numbers. If the input is a number from 1-31, then setDate(*Month,* int) is called to check whether that is a valid combination and to initialize *Date* accordingly. If not, then an error message is printed, with the asked input ranges differing based on *Date’s* *Month*. This loop continues until (int) date is of a valid *Month* (int) date combination.

**How an *Event* class works:**

-An *event* has a: *Date(Date has a* Month *and a (int) date)*, *Currency*, *Spending\_Cat*, (double) amount, (string) name, (string) note, and (boolean) done that is used to determine whether user needs to reinput data. Note that (boolean) done is just a boolean flag for the >> (cin) operation and is not used in any other Event calculations. Thus *Date(Date has a* Month *and a (int) date)*, *Currency*, *Spending\_Cat*, (double) amount, (string) name, and (string) note are the primary private data members of *Event*.

-*Event* #includes<Date.h (by extension Month.h), *Currency.h, and Spending\_Cat >*

-Its default constructor initializes an *event* with *nullCurrency*, *nullSpending*, monetary amount of -1, name of “”, note of “”, a *Date* of (*nullMonth*, -1), and (boolean) done = false. Note that realistically, this type of event is impossible.

-Note in practice, the (double) monetary amount is rounded to the nearest hundredth, and the *Month* and (int) date of *Date* is set through using *Date*::setDate(*Month*, int).

-Note how the *event* file also contains the methods +int numDigits(double) and +double round(double). The first one is used to calculate how many digits does a number contain for printing width purposes and the second one is used to round values to the nearest hundredth. Any class that #includes <Event.h> can access these methods.

-The *event* class has public getter and setter methods for each of the primary private data members.

-For setters:

+void changeDate(*Month*, int) – note this method simply references *Date*::setDate(*Month*, int) to change *Date*’s private members, as *Event* does not have direct access to them.

+void changeSpendingType(*Spending\_Cat*)

+void changeCurrency(*Currency*)

+void changeAmount(double)

+void changeTitle(string) – note name and title are interchangeable for referencing the same variable

+void changeNote(string)

For getters:

+*Date* getDate()

+*Spending\_Cat* getSpendingType()

+double getAmount()

+string getTitle()

+string getNote()

+*Currency* getCurrency()

-The *Event* class also has a clear() method, used to reset a user’s input when they wish to re-input their Event data in again. Note that *Spending\_Cat,* (double) amount, (string) name, (string) note, and Boolean (done)are reset to *nullSpending*, -1, “”, “”, and false respectively. However, because all *events* in a *budget* have the same month and currency, month and currency do not change.

-The *Event* class overrides 2 operators: << and >>

<< cout prints an *Event* by its primary private data members by *Date*, (string) name, (double) amount, (*Spending\_Cat*) type, and (string) note, in that order. Note because Visual Studio is unable to print a euro, renminbi, or British pound sign, the amount is printed as ### (3 letter Currency initial). Note the use of setW() and its various const int variables to help maintain formatting. Also note how (double) amount is printed using setprecision(numDigits(a.amount) + 2) to print the amount to the hundredth’s place, which is the number of non-decimal digits + 2 (additionally adding tenth & hundredth place).

>> cin inputs an *Event* by its primary private data members by *Date*, (*Spending\_Cat*) type, (double) amount, (string) name, and (string) title, in that order. Note that in >> *Date*, one only has to input a *Month* if *Month* is *nullMonth*, however because *Month* is already initialized in *Budget*, >> *Date* only initializes the Event (double) date. Note if the user inputs an invalid input for *Spending\_Cat*, the program will force the user to re-input until it is of an acceptable answer. Note for >> (double) amount, the program uses stringstream for error handling of inputs containing non-numbers. Because (double) realAmount is initialized to 0, all input that begins with non-numers are interpreted as 0. Note for while the user input is of a negative amount, the program will for the user to re-input until it is of an acceptable answer. Note in the end, the actual *Event* (double) amount is the (double) realAmount rounded to the nearest hundredth. Note for << *Event* (string) name and << *Event* (string) note, getline() is used instead of >> to account for input with multiple words and space in between, thus in.ignore() is needed to help >> and getline() stay separate. Note after all primary data is inputted, the user must confirm whether all their data is correct, to where using (string) checkDone, the only valid answer is “0” for re-entering and “1” for correct. If “0”, then clear() is called and the process restarts again.

**How a *Budget* class works:**

-A *Budget* has a:

*-Event[] –* isthe backing array for all the *Events*

*-Currency,*

*-Month*,

-(int) numResizes, (int) numEvents, (int) currArrLength – are all used during the backing array resizing

-(double) fixedPercentage, (double) varPercentage, (double) savePercentage – represent the user-chosen conditional percentages for each spending category, with default being 50/30/20%.

-(double) afterTaxIncome,

-(double) amountLeft – is the total monetary amount left in the budget to spend

-(double) amountSpent – is the total monetary amount in the budget already spent

-(double) condFixedAmount, (double) condVarAmount, (double) condSaveAmount – total monetary amounts allocated for each spending type (afterTaxIncome \* respective Spending percentage)

-(double) currFixedAmount, (double) currVarAmount, (double) currSaveAmount – current monetary amount left to spend for each spending type (cond\_\_\_Amount – sum of all monetary amounts for each spending type)

-(double) currFixedSpent, (double) currVarSpent, (double) currSaveSpent – current monetary amount already spent for each spending type (sum of all monetary amounts for each spending type)

-(double) currFixedAmountPercentage, (double) currVarAmountPercentage, (double) currSaveAmountPercentage – corresponding percentage of curr\_\_\_\_Amount ((cond\_\_\_Amount – sum of all monetary amounts for each spending type) / cond\_\_\_\_Amount)

-(double) currFixedSpentPercentage, (double) currVarSpentPercentage, (double) currSaveSpentPercentage – corresponding percentage of cur\_\_\_Spent ((sum of all monetary amounts for each spending type) / cond\_\_\_\_Amount)

-(double) currTotalAmountPercentage, (double) currTotalSpentPercentage – corresponding percentage of amountSpent & amountLeft (amountLeft / afterTaxIncome), (amountSpent / afterTaxIncome)

-(static const int) INITIAL\_CAPACITY = 5 – initial capacity of backing array

-(static const int) PERCENT = 100 – used for percent calculations

-Its default constructor initializes a *Budget* with nullCurrency, nullMonth,

default 50/30/20 (fixedPercentage = .5, varPercentage = .3, savePercentage = .2),

afterTaxIncome = 0, numEvents = 0,

condFixedAmount = round(a \* fixedPercentage),

condVarAmount = round(a \* varPercentage),

condSaveAmount = round(a \* savePercentage),

currFixedAmount = condFixedAmount,

currVarAmount = condVarAmount,

currSaveAmount = condSaveAmount,

currFixedSpent = currVarSpent = currSaveSpent = 0,

currFixedAmountPercentage = currVarAmountPercentage = currSaveAmountPercentage = currTotalAmountPercentage = 1

currFixedSpentPercentage = currVarSpentPercentage = currSaveSpentPercentage = currTotalSpentPercentage = 0

Amount Spent = numResizes = 0

currArrLength = INITIAL\_CAPACITY,

and initializes the *Event[]* backing array to of size INITIAL\_CAPACITY. Note the only parameters the *Budget* constructor takes are for *Currency, Month,* afterTaxIncome, and the 3 conditional percentages for each spending type. All the other variables can either be calculated using those variables or are trivial enough to initialize to a set number each time, either 0, 1, or 5.

*-Budget* #includes <Event.h (by extension Date.h (by extension Month.h), *Currency.h, and Spending\_Cat*)>

-The *Budget* class has 9 different public modification methods. Those all stem from the modification phase and they are:

+void addEvent(*Event*) – adds Event to backing array, and all numerical analysis are updated accordingly (monetary amount spent & corresponding percentages increase, monetary amount left & corresponding percentages decrease). Note if backing array is already full, then resize() is called. If resize() is called, then addEvent runs in O(n) time (worst case), if not, then O(1) (best case).

+void deleteEvent(int I) – deletes Event at index I and all numerical analysis are updated accordingly (monetary amount spend & corresponding percentages decrease, monetary amount left & corresponding percentages increase). Note that deleteEvent() treats the backing array just like an ArrayList in Java, and shifts all valid indices greater than I back one. While all the numerical calculations are done in O(1) time, the shift itself is done in amortized O(n) time.

+void changeEventAmount(int I , double) – changes monetary amount of Event at index I and all uses the difference of the old and new amounts to update the numerical analysis accordingly, which is done in O(1) time. References Event::changeEventAmount(double) to actually change monetary amount of Event at index I.

+void changeEventType(int I , *Spending\_Cat*) – changes *Spending\_Cat* of Event at index I. Uses the event at index I’s amount to add to the new *Spending\_Cat* numerical analysis and subtract from the old *Spending\_Cat* numerical analysis, all done in O(1) time. Has written code for all 6 possible combinations (3 options for old *Spending\_Cat*, 2 options for new *Spending\_Cat*, thus 3 \* 2 = 6). References Event::changeSpendingType() to actually change *Spending\_Cat* of Event at index I.

+void changeEventName(int I, string) – changes name of Event at index I. References Event::changeEventName(string) to actually change name of Event at index I. No new numerical analysis needed, entire operation done in O(1) time.

+void changeEventNote(int I, string) – changes note of Event at index I. References Event::changeEventNote(string) to actually change note of Event at index I. No new numerical analysis needed, entire operation done in O(1) time.

+void changeConditionalP(double, double, double) – changes the 3 conditional percentages for each spending type. Assumes inputs are from 0.00 to 1.00. Note how it rounds both the percentages and the cond\_\_\_\_Amounts to the hundredth’s place. References calculate() to help recalculate all other numerical analysis, runs in O(n) time.

+void changeAfterTaxIncome(double a) – changes the budget’s after-tax income. Rounds a and all cond\_\_\_\_amount calculations to the nearest hundredth, and references calculate() to help recalculate all other numerical analysis, runs in O(n) time.

+void changeCurrency(*Currency)* – changes the budget’s currency, performs for-ex operations between USD, CNY, EUR, and GBP. Exchange rates were pulled from the internet, note how many decimal places they go to. All new calculations are rounded to the nearest hundredth, all monetary amounts are updated accordingly, references Event::ChangeCurrency(*Currency*) to change each individual event’s *currency*. Finally calls calculate() to help recalculate all other numerical analysis, runs in O(n) time.

-Note for all modification methods modifying an event at index I, each method assumes user input of a range of events from (1 – n), but because the array itself indexes from (0 – (n – 1)), each method shifts the user input of index back by one.

-In addition to those 9 modification methods, *Budget* also has a couple more auxiliary functions that help those modification methods:

+void resize() – invoked during addEvent(*Event*) when backing array is already full. Creates a new temporary array twice the length of the current array length, copies all values over, then assigns temporary array as new backing array. Similar idea as with ArrayLists in Java. Runs in O(n) time.

+void calculate() – recalculates all relevant data and numerical analysis from scratch. Originally thought to be not needed, but C++ doesn’t handle small decimal math very well, especially with the percentages and tends to round in the wrong places. Used to guarantee correctness of data and numerical analysis, runs in O(n) time.

-*Budget* also has plenty of getter methods:

+int getNumEvents()

+Month getMonth()

+Currency getCurr()

+double getCurrTotalAmountPercentage()

+double getCurrTotalSpentPercentage()

+double getCurrFixedAmountPercentage()

+double getCurrFixedSpentPercentage()

+double getCurrVarAmountPercentage()

+double getCurrVarSpentPercentage()

+double getCurrSaveAmountPercentage()

+double getCurrSaveSpentPercentage()

+double getAmountSpent()

+double getAmountLeft()

+double getAfterTaxIncome()

+double getFixedPercentage()

+double getVarPercentage()

+double getSavePercentage()

+double getCondFixedAmount()

+double getCondVarAmount()

+double getCondSaveAmount()

+double getCurrFixedAmount()

+double getCurrVarAmount()

+double getCurrSaveAmount()

+double getCurrFixedSpent()

+double getCurrVarSpent()

+double getCurrSaveSpent()

-*Budget* has overridden 3 operators : [], <<, >>

[] allows *Budget* to directly access the backingArray at index I using the hard brackets, just like an array. Thus the *Budget* object in the runner file can easily access any index for any Event. Index I must be in between 0 – (n – 1).

<< cout prints a *Budget* by its primary data members. The header prints out the *Currency*, the *Month*, and numEvents. From there, a for-loop prints out all the *Budget’s Events* using *Event*’s overridden << cout print. Note the many different const int variables to help maintain formatting. After all the *Events* are printed, the conditional amounts and percentages of each category are printed, for money still left to spend and money already spent. *Budget* then prints the 3 conditional percentages of each spending category are listed along with their accompanying conditional monetary amount of the total after-tax income. Then *Budget* prints the total amounts and total percentages for money still left to spend and money already spent. Lastly, *Budget* then checks whether the user has exceeded their budget in any of the 3 allocated spending types, and outputs a warning message accordingly. Finally, the *Budget* checks whether the user has exceeded their entire budget and prints a very very very special message for them.

>> cin inputs a *Budget* by its primary data members and in 2 parts. The first part cin inputs *Currency*, *Month*, after-tax income using stringstream and rounds to the nearest hundredth, and gives the user a choice for the 3 conditional percentages for each spending type whether to use the 50/30/20 default or make their own combination that also sums to 100. If the user chooses to make their own combination, then their input for each percentage is read in using stringstream and further rounds to the nearest hundredth for error handling. If their combination does not sum to 100, then the program through a boolean flag has them re-enter their numbers again. If they do sum to 100, then the program uses a boolean flag to exit the loop. From there, the program outputs to the user the *Currency*, *Month*, after-tax income, and three conditional percentages for each spending type and prompts them to confirm them. If they choose to re-enter them, then they are lead to repeat their actions again. If not, then the program automatically calculates the cond\_\_\_\_Amounts and curr\_\_\_\_Amounts and notifies the user how much money they have allocated to each spending type. From there, the user now inputs all of their events until they signal that they are done. For one last validity check, the program runs calculate() to guarantee the correctness of the numerical analysis.

**How the Runner Class works**

-The runner class #includes <*Budget.h (by extension* Event.h (by extension Date.h (by extension Month.h), *Currency.h, and Spending\_Cat)*)>

-A brief introduction is first cout printed to the page, followed by clarifying assumptions, flashy features, and a step-by-step tutorial. From there, the user begins to input their *Budget* using cin <<.

-After they complete inputting their *Budget*, the program then takes them to the modification stage, where they can modify 9 different things:

1) Add an Additional event

2) Delete an event at index I

3) change an event’s monetary amount at index I

4) change an Event's spending type at index I

5) change an Event's name at index I

6) change an Event's note at index I

7) change conditional percentages for the spending types

8) change total After-tax income

9) change currency used (forex foreign exchange)

-Using a string for error handling input with non-numbers and the above numbering system, the program then enters the appropriate if-block. If the input is invalid, the user is prompted to re-enter their input until it is accepted.

-For all choices that modify data “at index I”, the program uses stringstream for error handling input with non-numbers and for reading them into integers, where their input is checked whether it is within (1 – numEvents inclusive) and if it is not, the user is asked to re-enter their index.

-For all choices in total, before the modification is done, the user is prompted whether all newly inputted data is correct or not using a string for checking.

-For 1), the user inputs data for a new *Event* using cin >> and that event is then added to the *budget*

-For 2), the user first inputs an index then confirms their choice and then that event is removed from the *budget.*

-For 3), the user first inputs an index then inputs a new monetary amount that is read by a stringstream for error handling then confirms their choices.

-For 4), the user first inputs an index then inputs their new S*pending\_Cat* read through >> and then confirms their choices

-For 5), the user first inputs an index then inputs the new *Event* name using cin.ignore() and then confirms their choices.

-For 6), the user first inputs an index then inputs the new *Event* note using cin.ignore() and then confirms their choices.

-For 7), the input process for the three conditional percentages for each spending type is the same as how they were inputted in *Budget* >> cin and then the user confirms their choices.

-For 8), the input process for the after-tax income is the same as how it was inputted in *Budget* >> cin and then the user confirms their choices.

-For 9), the relevant exchange rates are first printed out to the user, then the user cin inputs their new *Currency* and confirms their choices.

-After each additional modification, the *Budget* calls calculate() to guarantee calculation correctness then prints out the new, current *Budget* back to the user.

-This modification loop continues until the user signals that they are done.

-The very final part of the program is one last cout print of the *Budget* back to the user. Good job!