1 Instructions

You may work in pairs (that is, as a group of two) with a partner on this lab project if you wish or you may work alone. If you work with a partner, only of you will submit the lab project to Canvas for grading. To ensure you each receive the same score, it is imperative that you follow the instructions in §5 *What to Submit for Grading and by When* (i.e., put both of your ASURITE ID's in the submission archive filename and write both of your names, ASURITE ID's, and email addresses in the AUTHOR1: and AUTHOR2: lines of the header comment block for each source code file). What to submit for grading, and by when, is discussed in §5; read it now.

2 Lab Objectives

After completing this assignment the student should be able to:

- Complete all of the objectives of the previous lab projects.
- Write function definitions, call functions, pass parameters, define local variables, return a value from a function.
- Open a text file for reading/writing and read/write from/to the text file.
- Write if, if-else, and if-elseif-... (if/else if) statements.
- Document formal test cases and perform testing to verify program correctness.

3 Prelab Exercises

3.1 Prelab Exercise 3: Read Software Requirements

Read §4 *Software Requirements* of this lab project document which describes what the lab project program will do, i.e., the **software requirements**. Then come back here and complete the remaining prelab exercises.

3.2 Prelab Exercise 2: Create Repl.it Project and Add *main.cpp* to the Project

You may use any C++ development environment you wish to complete the project. As we did for Lab Project 6, we will be opening an input file for reading and an output file for writing in this program. If you wish to use an online C++ IDE, we recommend that you use Repl.it.

If you have not already done so, before your lab session, create a free Repl.it account. After doing that, create a new C++ project and rename the project "Lab 7" (Note: if you do not have an account and are logged in as @anonymous, you will not be able to rename the project from the default random project name that was assigned to it when the project was created.) Then upload the *main.cpp* source code file you extracted from the Lab Project 7 zip archive to your Repl.it project, overwriting the *main.cpp* file that was added to the project when you created it.

Before proceeding to the next prelab exercise, edit the header comment block of the provided *main.cpp* file and add your information to the AUTHOR1: comment. If you work with a partner, add the AUTHOR2: comment including his or her information. Remember that only one member of the team has to upload the submission zip archive to Canvas but you will each earn the same score as long as you follow the instructions in this section and in §5.

For help using Repl.it, click the ? button in the lower-right corner of the IDE window. If you require more assistance, ask your lab TA or a UGTA for help in your lab session. Note that when when you are finished with the lab project, you may download your project files in a zip archive: Click the three dots symbol in the title bar of the *File Pane* and select *Download as Zip* from the menu. This will download a zip archive named *Lab-7.zip* (assuming you named your project *Lab 7* as requested).

Note: The downloaded zip archive is **not** the zip archive you are required to upload to Canvas for grading. Please consult §5 for that information.

3.3 Prelab Exercise 3: Create Test Input File *payroll.txt*

Lab 7 will read input from a text file named *payroll.txt*. A sample *payroll.txt* file is included in the Lab 7 zip archive. Upload this file to your Repl.it project and use it for testing. Now proceed to Prelab Exercise 4.

3.4 Prelab Exercise 4: Study the Software Design

I am fairly constrained in the design of this program by our limited C++ knowledge, so this design is not the design I would create if we knew some material that we will eventually learn in Chapter 7. This program involves several variables, which need to be shared among the functions of the program, and if we declared some of those variables as global variables, then I could improve this design, but global variables are discouraged for very good reasons, so I did not want to go that direction. In the end, I do not like the size of the *main*() function in the pseudocode and the source code file template. The function is too large and contains too many separate and unrelated operations, but given our constraints, it is what it is. Therefore, the program shall consist of the following eight functions.

`main () → int`
The program starts executing here. *main*() will call *open_input_file*() to open *payroll.txt* for reading. It will read the employee's information from the file and then close it. *main*() then calls functions to calculate the employee's paycheck information. *main*() then calls *open_output_file*() to open *paycheck.txt* for writing and then sends the employee's paycheck information to the output file and then closes it. *Note: In an ideal design, I would break main() up into at least three separate functions: (1) One that reads the employee payroll information from the input file and returns the information; (2) One that calls the functions to calculate the various paycheck values; and (3) One that writes the paycheck information to the output file. But since we are constrained by not being able to use C++ structs, classes, or global variables, then it would be more difficult to implement the code by trying to break main() up into these three separate functions, so although putting this much functionality in main() is bad design, it is what we shall work with.*

`calc_gross_pay(pay_rate : double, hrs_worked : double) → double`
Calculates and returns an employee's gross pay which is based on the number of hours worked in parameter *hrs_worked* and the employee's pay rate in parameter *pay_ rate.*

`calc_med_ins_deduct(med_ins_status : int) → double`
Determines and returns the employee's medical insurance deduction which is based on the employee's medical insurance status in parameter *med_ins_status.*

`calc_tax_fed(fed_tax_gross_pay : double) → double`
Calculates and returns the employee's federal income tax which is based on his or her federal taxable gross pay in parameter *fed_tax_gross_pay* and the federal tax withholding percentage table in the lab project document.

`calc_tax_state(fed_tax_gross_pay : double) → double`
Calculates and returns the employee's state income tax which is based on his or her federal taxable gross pay in parameter *fed_tax_gross_pay* and the state tax withholding percentage table in this lab project document.

`open_input_file(fin : ifstream&, filename : string) → nothing`
This is the same function we wrote and used in Lab Project 6, so you may copy-and-paste the code for this function from your Lab Project 6 source code file. *In the real world, we would put the compiled object code for this function in a library and then when we need to use the function, we would just sort of "check it out" in the same way we use functions from the C++ Std Lib, by writing an #include preprocessor directive that includes a header file containing the prototype for this function.*

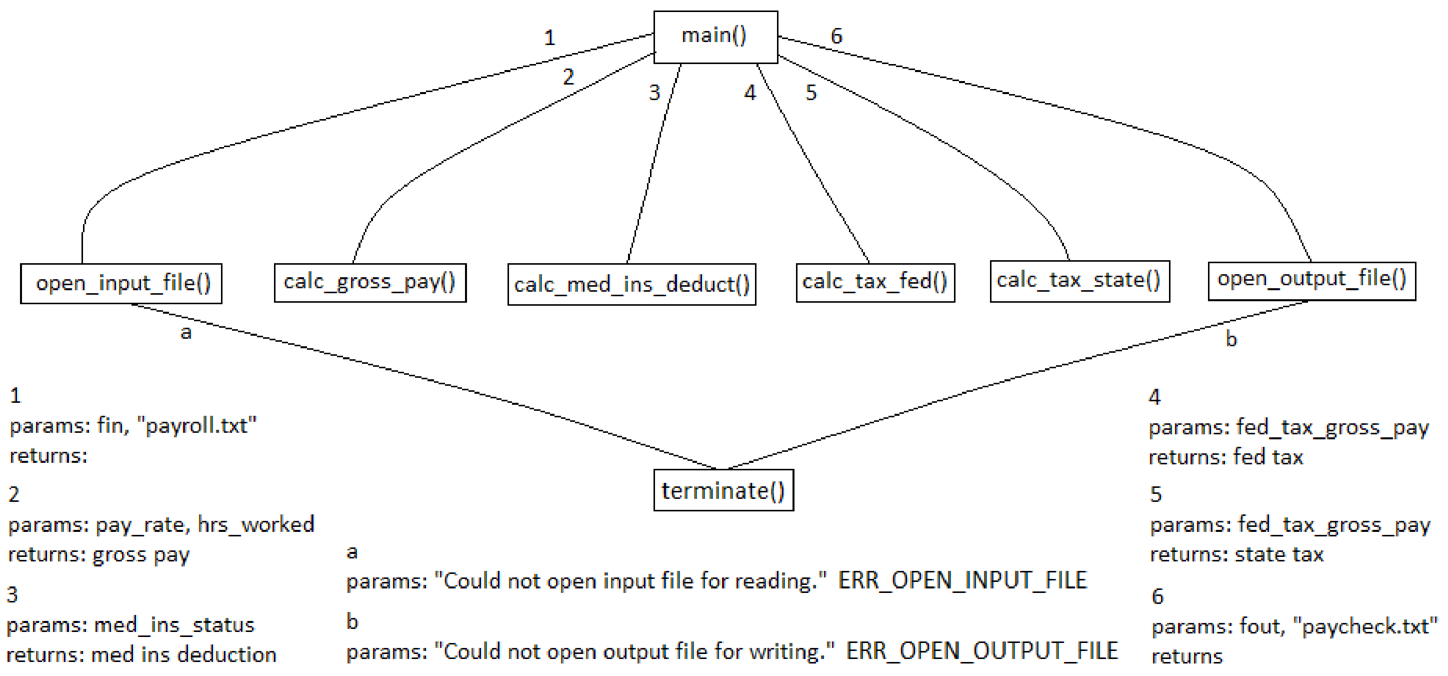
`open_output_file(fout : ofstream&, filename : string) → nothing`
This is the same function we wrote and used in Lab Project 6, so you may copy-and-paste the code for this function from your Lab Project 6 source code file. *See the comments in* `open_input_file()` *about putting this function in a library.*

`terminate(message : string, exit_code : int) → nothing`
This is the same function we wrote and used in Lab Project 6, so you may copy-and-paste the code for this function from your Lab Project 6 source code file. *See the comments in* `open_input_file()` *about putting this function in a library.*

The structure chart showing each of the functions and the function calls is on the next pabe. The numbers 1-6 indicate the order in which *main*() calls the other functions. *open_input_ file*() and *open_output_file*() only call *terminate*() when the respective file cannot be opened.

And below the structure chart is the pseudocode, which you are to implement. A few pseudocode notes:

1. I use ← to represent the assignment operator.
2. I use → in the function header to specify the data type of the return value.
3. I omit data types when the type can be inferred from the SW Requirements, the comments in the pseudocode, or the values being assigned to variables of the data type.
4. I use an em dash — to represent the beginning of a comment, which extends to the end of the line.


main()
1
2
3
4
5
6
open_input_file()
calc_gross_pay()
calc_med_ins_deduct()
calc_tax_fed()
calc_tax_state()
open_output_file()
a
b
terminate()
1
params: fin, "payroll.txt"
returns:
2
params: pay_rate, hrs_worked
returns: gross pay
3
params: med_ins_status
returns: med ins deduction
a
params: "Could not open input file for reading." ERR_OPEN_INPUT_FILE
b
params: "Could not open output file for writing." ERR_OPEN_OUTPUT_FILE
4
params: fed_tax_gross_pay
returns: fed tax
5
params: fed_tax_gross_pay
returns: state tax
6
params: fout, "paycheck.txt"
returns

```
file main.cpp
    — These constants are parameters to terminate() when the input and output file cannot be opened
    define constant ERR_OPEN_INPUT_FILE ← 1
    define constant ERR_OPEN_OUTPUT_FILE ← 2

    — These constants represent percentages discussed in the Software Requirements
    define constant OASDI_RATE ← 6.2%
    define constant FOUR01K_RATE ← 6%
    define constant MEDICARE_RATE ← 1.45%

    — These constants represent the dollar amounts for medical insurance based on the employee's medical insurance status code
    define constant MED_INS_DEDUCT_EMPL_ONLY ← $32.16
    define constant MED_INS_DEDUCT_EMPL_ONE ← $64.97
    define constant MED_INS_DEDUCT_EMPL_FAMILY ← $110.13

    — These constants represent three possible medical insurance status codes
    define constant MED_INS_STATUS_EMPL_ONLY ← 0
    define constant MED_INS_STATUS_EMPL_ONE ← 1
    define constant MED_INS_STATUS_EMPL_FAMILY ← 2

    — ------------------------------------------------------------------------------------------------------------
    — calc_gross_pay() – Calculates the employees gross pay, which is based on his/her payrate and hours worked.
    — ------------------------------------------------------------------------------------------------------------
    function calc_gross_pay (pay_rate : double, hrs_worked : double) → double
        define gross_pay ← 0.0
        if hrs_worked ≤ 80 then
            gross_pay ← hrs_worked × pay_rate
        else
            gross_pay ← (80 × pay_rate) + (hrs_worked - 80) × (1.5 × pay_rate)
        end if
        return gross_pay
    end function
```

```
— ------------------------------------------------------------------------------------------------------------
— calc_med_ins_deduct() – Given the employee's medical insurance status code, this function determines and returns the
— employee's medical insurance deduction based on the status code.
— ------------------------------------------------------------------------------------------------------------
function calc_med_ins_deduct (med_ins_status : int) → double
    define med_ins_deduct ← 0
    if med_ins_status is MED_INS_STATUS_EMPL_ONLY then
        med_ins_deduct ← MED_INS_DEDUCT_EMPL_ONLY
    else if med_ins_status is MED_INS_STATUS_EMPL_ONE then
        med_ins_deduct ← MED_INS_DEDUCT_EMPL_ONE
    else
        med_ins_deduct ← MED_INS_DEDUCT_FAMILY
    end if
    return med_ins_deduct
end function

— ------------------------------------------------------------------------------------------------------------
— calc_tax_fed() – Given the employee's federal taxable gross pay, this function calculates and returns his/her federal tax.
— ------------------------------------------------------------------------------------------------------------
function calc_tax_fed (fed_tax_gross_pay : double) → double
    define tax_fed ← 0
    if fed_tax_gross_pay ≥ $384.62 and fed_tax_gross_pay < $1413.67 then
        tax_fed ← fed_tax_gross_pay × 7.97%
    else if fed_tax_gross_pay ≥ $1413.67 and fed_tax_gross_pay < $2695.43 then
        tax_fed ← fed_tax_gross_pay × 12.75%
    else if fed_tax_gross_pay ≥ $2695.43 then
        tax_fed ← fed_tax_gross_pay × 19.5%
    end if
    return tax_fed
end function

— ------------------------------------------------------------------------------------------------------------
— calc_tax_state() – Given the employee's federal taxable gross pay, this function calculates and returns hizer state tax
— ------------------------------------------------------------------------------------------------------------
function calc_tax_state (fed_tax_gross_pay : double) → double
    define tax_state ← 0
    if fed_tax_gross_pay < $961.54 then
        tax_state ← fed_tax_gross_pay × 1.19%
    else if fed_tax_gross_pay < $2145.66 then
        tax_state ← fed_tax_gross_pay × 3.44%
    else
        tax_state ← fed_tax_gross_pay × 7.74%
    end if
    return tax_state
end function

— ------------------------------------------------------------------------------------------------------------
— open_input_file() – Pseudocode omitted. Copy and paste the code for the same function from Lab Project 6
— ------------------------------------------------------------------------------------------------------------
function open_input_file (fin : ifstream&, filename : string) → nothing
    — body omitted
end function
```

```
— ------------------------------------------------------------------------------------------------
— open_output_file() – Pseudocode omitted. Copy and paste the code for the same function from Lab Project 6
— ------------------------------------------------------------------------------------------------
function open_output_file (fout : ofstream&, filename : string) → nothing
    — body omitted
end function

— ------------------------------------------------------------------------------------------------
— terminate() – Pseudocode omitted. Copy and paste the code for the same function from Lab Project 6
— ------------------------------------------------------------------------------------------------
function terminate (msg : string, exit_code : int) → nothing
    — body omitted
end function

— ------------------------------------------------------------------------------------------------
— main() –
— ------------------------------------------------------------------------------------------------
function main () → int
    — Read the employee's payroll information from "payroll.txt". In an ideal design, I would move this code to a separate
    — function and just call that function from here.
    define ifstream object named fin
    call open_input_file (fin, "payroll.txt")    — will not return if the file could not be opened
    define string objects first_name, last_name
    read the first and last name from fin into first_name and last_name
    define double variables pay_rate, hrs_worked
    read the pay rate and hours worked from fin into pay_rate and hrs_worked
    define int variable med_ins_status
    read the medical insurance status code from fin into med_ins_status
    close the input file

    — Calculate the employee's paycheck data.  In an ideal design, I would move this code to a separate function and just call
    — that function from here.
    gross_pay ← calc_gross_pay (pay_rate, hrs_worked)
    four01k_deduct ← gross_pay × FOUR01K_RATE
    med_ins_deduct ← calc_med_ins_deduct (med_ins_status)
    fed_tax_gross_pay ← gross_pay - med_ins_deduct - four01k_deduct
    tax_fed ← calc_tax_fed (fed_tax_gross_pay)
    tax_oasdi ← fed_tax_gross_pay × OASDI_RATE
    tax_medicare ← fed_tax_gross_pay × MEDICARE_RATE
    tax_state ← calc_tax_state (fed_tax_gross_pay)
    tax_total ← tax_fed + tax_oasdi + tax_medicare + tax_state
    net_pay ← fed_tax_gross_pay - tax_total

    — Sends the employee's paycheck to "paycheck.txt".   In an ideal design, I would move this code to a separate function and
    — just call that function from here.
    define ofstream object named fout
    call open_output_file (fout, "paycheck.txt")   — will not return if the file could not be opened
    configure fout to output real numbers in fixed notation with 2 digits after the decimal point
    configure fout for right-justification mode
    send "-----------------------------", endl to fout
    send "EMPLOYEE: ", last_name, ", ", first_name, endl, endl to fout
    send "PAY RATE:           $" to fout
    send pay_rate, endl to fout displaying the value in a field of width 8
    send "HOURS:                " to fout
    send hrs_worked, endl to fout displaying the value in a field of width 8
    send "GROSS PAY:          $" to fout
    send gross_pay, endl to fout displaying the value in a field of width 8
```

```
        send "MED INS DEDUCT:      $" to fout
        send med_ins_deduct to fout displaying the value in a field of width 8
        send "401K DEDUCT:         $" to fout
        send four01k_deduct, endl to fout displaying the value in a field of width 8
        send "FED TAX GROSS PAY:   $" to fout
        send fed_tax_gross_pay, endl to fout displaying the value in a field of width 8
        send "TAX - FEDERAL:       $"  to fout
        send tax_fed, endl to fout displaying the value in a field of width 8
        send "TAX - OASDI:         $"  to fout
        send tax_oasdi, endl to fout displaying the value in a field of width 8
        send "TAX - MEDICARE:      $"  to fout
        send tax_medicare, endl to fout displaying the value in a field of width 8
        send "TAX - STATE:         $"  to fout
        send tax_state, endl to fout displaying the value in a field of width 8
        send "TAX - TOTAL:         $"  to fout
        send tax_total, endl to fout displaying the value in a field of width 8
        send "NET PAY:             $"  to fout
        send net_pay, endl to fout displaying the value in a field of width 8
        send "-----------------------------", endl to fout
        close the output file

        return 0
    end function main

end file
```

3.5 Prelab Exercise 5: Testing

After uploading *main.cpp* to your Repl.it project, open *main.cpp* in the editor. In the *Testing* section of the header comment block, I have documented one test case, using the employee data for Homer Simpson from §4. For this exercise, you are to design and document **two additional test cases**. I have included a testing spreadsheet file named *cse100-lab07-testing.ods* in the lab project zip archive. This spreadsheet is in Libre Calc format (I do not use Microsoft Office) so you may either install Libre Calc and load the spreadsheet or you may try to convert the spreadsheet from Libre Calc format to Microsoft Office format. Microsoft Office *will* read the Libre Calc spreadsheet file, but the way certain formulas are implemented in Excel is different than the way Calc implements them, so some of the imported formulas from Calc will be messed up in the Excel spreadsheet. I will leave it to you to correct the spreadsheet formulas if you attempt to load the Calc testing spreadsheet into Excel. But regardless of how you do it, you need to create your own input data for two test cases, use your calculator or the spreadsheet to calculate what the output values should be, and document the expected output in the appropriate section of the test case template in the header comment block of *main.cpp*.

I would suggest that your three test cases (the one I gave you and the two you are creating) test all three possible values for the medical insurance status code. For example, for the test case I gave you, I used 2. I would suggest using 0 for your first test case and 1 for your second test case. This will ensure that every true and false clause in the if-elseif-... statement in *calc_med_ins_deduct*() is executed and is correctly implemented.

At least one test case should be for hours worked less than or equal to 80; at least one should be for hours worked greater than 80; and it would be advisable to have a test case where hours worked is equal to 80. This will ensure that both the true and false clauses in *calc_gross_pay*() are executed and are correctly implemented.

Then, for *calc_tax_fed*() and *calc_fed_state*(), you should design your input data such that each true and false clause in the if-elseif-... statements in these functions are executed and correctly documented. The Libre Calc spreadsheet discusses testing all the functions of the program, including suggestions for how to test *calc_tax_fed*() and *calc_tax_state*().

Remember, the entire point of testing is to try to find bugs and this is critical because all but reasonably small programs have bugs in them. Although asking you to create these test cases before you write the code may seem like busy work, it is necessary because after you write the program, you do not know if there are bugs in it, until you thoroughly test the code and for that, you need test cases. Remember that just because your program compiles and runs does not mean it is correct. A reasonably large program can never be proven to be bug-free. The best we can do is to test, test, and test some more to try to find all the bugs we can.

Test coverage is a term which relates to how much of the code is tested by a specific test case and by all of the test cases. If you design your test cases as I described in the Libre Calc testing spreadsheet, then every single statement in this program will be executed, so your test coverage will be 100%, which it should be. If your test coverage is not 100%, it means that there are some statements in your code which were never tested, and untested statements could contain bugs in them. You can bet these untested statements will be executed by the users of your program, and if there are bugs in these statements, then your customers will find them and customers are never happy when the programs they rely on contain bugs. Make your product buggy enough and your customers will simply move to a competitor's product and keep doing this until they find a product that meets their requirements and is stable, i.e., not buggy and crashing all of the time (unless they are forced to use Canvas as we faculty are, and are not given other options).

4 Lab Exercise: Software Requirements[1]

All employees at **FreeMedicalOrgans.com**[2] are paid **biweekly**, i.e., every two weeks. An **hourly** employee's biweekly **gross pay** (denoted by double variable *gross_pay*) is calculated as the number of **hours** he or she worked (denoted by double variable *hrs_worked*) during the pay period multiplied by his or her **hourly pay rate** (denoted by double variable *pay_rate*). In addition, hourly workers are eligible for **overtime pay** if their hours worked exceeds 80 hours during the two week pay period. The **overtime pay rate** is one-and-a-half times the usual pay rate. For example, if Wilma worked $hrs\_worked = 87$ hrs and her pay rate is $pay\_rate = \$19.25$ per hour, then her gross pay would be calculated as $gross\_pay = (80 \times pay\_rate) + (hrs\_worked - 80) \times (1.5 \times pay\_rate) = (80 \times \$19.25) + (87 - 80) \times (1.5 \times \$19.25) = \$1742.13$.

All employees have **taxes** withheld from their paychecks. Taxes that are withheld by the employer are: (1) **federal income tax** (denoted by double variable *tax_fed*); (2) **state income tax** (denoted by double variable *tax_state*); (3) **federal social security tax** (denoted by double variable *tax_oasdi*[3]); and (4) **federal medicare tax** (denoted by double variable *tax_medicare*).

The amounts withheld for this group of four taxes are based on a percentage of the employee's **federal taxable gross pay** (denoted by double variable *fed_tax_gross_pay*). An employee's federal taxable gross pay is equal to his or her gross pay minus deductions made for the mandatory company **401K retirement plan** (denoted by double variable *four01k_deduct*) and **medical insurance** (denoted by double variable *med_ins_deduct*), i.e., $fed\_tax\_gross\_pay = gross\_pay - four01k\_deduct - med\_ins\_deduct$. All employees are required to contribute 6.0% (denoted by double constant $FOUR01K\_RATE = 0.06$) of their gross pay to the company 401K retirement plan, i.e., $four01k\_deduct = FOUR01K\_RATE \times gross\_pay$. The rate an employee pays for medical insurance is based on the following table (where variable *med_ins_status* contains the medical insurance status code of 0, 1, or 2),

Medical Insurance Status	Biweekly Cost to Employee
Employee only, no dependents (code = 0)	\$32.16
Employee + 1 dependent (code = 1)	\$64.97
Family (code = 2)	\$110.13

The federal income tax withheld (variable *tax_fed*) from the employee's paycheck is based his or her federal taxable gross pay and this table,

Federal Taxable Gross Pay	Federal Tax Withholding Percentage
< \$384.62	0.00%
≥ \$384.62 and < \$1413.67	7.97%
≥ \$1413.67 and < \$2695.43	12.75%
≥ \$2695.43	19.5%

So, $tax\_fed = fed\_tax\_gross\_pay \times federal\ tax\ withholding\ percentage$. The amount withheld for state income tax (variable *tax_state*) is based on a percentage of federal taxable gross pay per the following table,

1 I made all of these numbers up. I would suggest that you not use my lab project document to figure your taxes.
2 Where our motto is, "Trust us, you don't want to know where these things come from, but, well, umm, oink."
3 OASDI stands for Old Age, Survivors, and Disability Insurance, although the term "social security" is more commonly used.

Federal Taxable Gross Pay	**State Tax Withholding Percentage**
$< \$961.54$	1.19%
≥ \$961.54 and < \$2145.66	3.44%
≥ \$2145.66	7.74%

So, *tax_ state* = *fed_ tax_ gross_ pay* × *state tax withholding percentage*. The amount withheld for OASDI (double variable *tax_ oasdi*) is 6.2% (denoted by double constant *OASDI_ RATE* = 0.062) of federal taxable gross pay, i.e., *tax_ oasdi* = *fed_ tax_ gross_ pay* × *OASDI_ RATE*. The federal medicare tax is 1.45% (denoted by double constant *MEDICARE_ RATE* = 0.0145) of the employee's federal taxable gross pay, i.e., *tax_ medicare* = *fed_ tax_ gross_ pay* × *MEDICARE_ RATE*.

For this programming exercise, you are to complete the C++ program template in *main.cpp* contained within the Lab 7 zip archive by implemented the pseudocode discussed in Prelab Exercise 4.

The program opens a text file named *payroll.txt* which contains payroll information for one hourly employee of Free MedicalOrgans.com. The format of the data in the file is,

```
lastname firstname
pay_rate hrs_worked
med_ins_status
```

The *med_ ins_ stratus* field is either 0 (for employee only, no dependents), 1 (for employee + one dependent), or 2 (for family); see the Medical Insurance Status Table above. An example hourly employee record would be,

```
Simpson Homer
15.25 84.0
2
```

Here, the employee's first and last names are "Homer" and "Simpson". Homer's pay rate is $15.25 per hour and he worked 84 hours in the last biweekly pay period (4 hours of overtime). Homer's medical insurance status is 2 (for employee + family). The program shall read the information from the file and perform calculations to determine,

1. Biweekly gross pay (stored in a variable named *gross_ pay*)
2. Company-required 401K deduction (stored in a variable named *four01k_ deduct*)
3. Medical insurance deduction (*med_ ins_ deduct*)
4. Federal taxable gross pay (*fed_ tax_ gross_ pay*)
5. Federal income tax (*tax_fed*)
6. OASDI tax (*tax_ oasdi*)
7. Medicare tax (*tax_ medicare*)
8. State income tax (*tax_ state*)
9. Total taxes (*tax_ total*)
10. Net pay (*net_ pay*)

The total taxes (*tax_ total*) is the sum of *tax_fed*, *tax_ oasdi*, *tax_ medicare*, and *tax_ state*. The employee's net pay (*net_ pay*) is his or her federal taxable gross pay minus total taxes, i.e., *net_pay* = *fed_ tax_ gross_ pay* − *tax_ total*.

The program shall then open an output file named *paycheck.txt* for writing and shall output the paycheck in this format,

```
-----------------------------
EMPLOYEE: Simpson, Homer

PAY RATE:           $   15.25
HOURS:                  84.00
GROSS PAY:          $ 1311.50
MED INS DEDUCT:     $  110.13
401K DEDUCT:        $   78.69
FED TAX GROSS PAY:  $ 1122.68
TAX - FEDERAL:      $   89.48
TAX - OASDI:        $   69.61
TAX - MEDICARE:     $   16.28
TAX - STATE:        $   38.62
TAX - TOTAL:        $  213.98
NET PAY:            $  908.70
-----------------------------
```

Note that all real numbers are printed with two digits after the decimal point. This is controlled by using the **fixed** and **setprecision()** stream manipulators. The numbers are all output right-justified in fields of width 8. This is controlled by using the **right** and **setw()** manipulators. Once the paycheck file has been printed, the output file shall be closed and the program shall terminate.

4.1 Additional Programming Requirements

1. Modify the **header comment block** at the top of *main.cpp* source code file so it contains the first author's information in the AUTHOR1: comment. If there is only one author, then delete the AUTHOR2: comment. However, if there are two authors, document the second author's information in the AUTHOR2: comment.
2. Do not forget to document your test cases in the header comment block.
3. Always write your code in a way to **enhance readability**. This includes **writing comments** to explain what the code is doing, **properly indenting** the code, and **using blank lines** to separate the various parts of the program. For guidance, study the example programs in the textbook and the source code the instructor writes in class or posts online.
4. When you complete the program, you will **run it three times** using the **three test cases** documented in the header comment block. **Document** in your testing results (in the header comment block of your source code file) the **Actual Output** from the program, i.e., copy-and-paste the contents of *paycheck.txt* to this section. If the Expected Output matches the Actual Output for all three test cases, then your program is most likely correctly implemented (testing can never prove a program does not have bugs, it can only demonstrate their presence). However, if there is a mismatch then it could be caused by one of two things: (1) the calculations you did in determining the expected output were incorrect and your test case contains incorrect data; or (2) your program has a bug in it and it is not outputting the correct results. In the case of (1) you should recalculate the expected output and then rerun the test. In the case of (2) you should locate the bug(s) and correct them. Keep re-running the program on the test cases until all test cases pass. When your testing is compete, **document** your test cases and testing results at the top of your source code file, **in the comment header block**. Indicate if each of the test cases **passed** or **failed**.

5 What to Submit for Grading and by When

- Note: There will be a 40% project penalty deduction (or -2 points) for not following these instructions and uploading incorrect files and/or incorrectly named files. This is because we will be using a grading script to grade your projects and when you do not name your files as requested, the script will fail and this will then require the grader to manually grade your project. Consider this potential 2 point deduction as a file/folder naming tax that you pay if you do not follow the instructions.
- If you work with a partner, it is imperative that both you and your partner belong to the same group in Canvas. In Canvas, navigate to the People page and click on the Project Groups tab. There are 80 groups named Project Group 1 through Project Group 80. Find an empty group, and add yourself and your partner to the group. You may also remove yourself from a group should you decide to work with a different partner on a future lab project.
- Using the file explorer program of your operating system, create a new empty folder named **lab07-*asurite*** where *asurite* is your ASURITE username that you use to log in to MyASU (e.g., mine is *kburger2* so my folder would be named lab07-kburger2). If you worked with a partner on the project, put both of your ASURITE usernames in the name of the new folder **lab07-*asurite1*-*asurite2***, e.g., Fred Flintstone's and Barney Rubble's project submission archive would be named **lab07-fflint-brubble.zip**.
- Copy your completed *main.cpp* source code file to the folder your just created. This folder should contain only one file named *main.cpp*. Do **not** copy the input file *payroll.txt* or the output file *paycheck.txt* to this folder. We do not need the data files because we will use our own when testing your program.
- Compress the folder creating a **zip archive** named **lab07-*asurite*.zip** or **lab07-*asurite1*-*asurite2*.zip**.

- Submit the zip archive to Canvas by the deadline using the *Lab Project 7* submission link.
- If you worked with a partner, then only one partner of the group needs to submit the solution archive to Canvas. When the project is graded, the grader will encounter the zip archive submission by the student who uploaded the file and will grade the submission. Then, Canvas will automatically assign the other partner of the group the same score.
- If your program does not compile or run correctly, upload what you have completed for grading anyway because you will generally receive some partial credit for effort.
- The submission deadline can be found in the Canvas Lab 7 submission page. It is prior to **11:59:00pm Sat 3 Apr**.
- We allow a 2-day late submission period. Each day the project is late, a 10% penalty per day deduction will be subtracted from your graded score. The lab project is worth 5 pts, so 10% of 5 pts is 0.5 pts, so if you submit the solution on Sun after the Sat deadline, your final score will be reduced by 0.5 pts and if you submit on Mon, your final score will be reduced by 1 pt.
- The Canvas submission link will become unavailable exactly at 11:59:00pm on the Mon following the Sat deadline and you will not be able to submit your project for grading after that time. We do not accept emailed submissions.
- Consult the online syllabus for the late project submission and academic integrity policies.

6 Grading Rubric

We will use several grading variables when grading your project: *doc, build*, and *partial* are all initialized to false. Grading variables *passed* and *naming* will be initialized to 0.

Then, we will set *naming* = 2 pts if the names of the files/folder you submit are not as requested in §5. We will subtract *naming* = 2 points from your lab project score if the files are not named as requested.

Next, we will examine the documentation of your test cases and will set *doc* = true if you properly performed testing and documented the test cases and test case results in the header comment block of your source code file in the requested format. Failure to do so will set *doc* to false.

Next, we will attempt to build your project. If it builds without syntax errors, then we will set grading variable *builds* to true. If your program fails to build due to syntax errors, then we set *builds* to false. To award partial credit for a project which does not build, the grader will examine your code. If it appears you made a good faith effort to complete the majority of the required code, then grading variable *partial* will be set to true. If none to little of the required code was implemented, then grading variable *partial* will be set to false.

We do not share our testing input and output files but next, we will test your program on three different input files. Grading variable *passed* was initialized to 0. For each test case which passes, *passed* will be incremented (a test case passes when the output your program writes to paycheck.txt matches the output in our paycheck.txt file).

Then, the grader will assign *points* per this table:

doc	*build*	*partial*	*passed*	*points*	*remarks*
true	true	n/a	3	5.0	Documented all test cases, built, ran, and passed all test 3 cases
false	true	n/a	3	4.5	Did not doc all test cases, built, ran, and passed all test 3 cases
true	true	n/a	1-2	4.0	Documented all test cases, built, ran, and passed 1-2 but not all 3 test cases
false	true	n/a	1-2	3.5	Did not doc all test cases, built, ran, and passed 1-2 but not all 3 test cases
true	true	n/a	0	3.0	Documented all test cases, built, ran, and failed all test cases
false	true	n/a	0	2.5	Did not doc all test cases, built, ran, and failed all test cases
true	false	true	n/a	2.0	Documented all test cases, syntax errors, good effort writing most of the code
false	false	true	n/a	1.5	Did not doc all test cases, syntax errors, good effort writing most of the code
true	false	false	n/a	1.0	Documented all test cases, syntax errors, poor effort writing little of the code
false	false	false	n/a	0.5	Did not doc all test cases, syntax errors, poor effort writing little of the code
false	false	false	n/a	0.0	No submission or simply submitted the given template file no modifications

And then the **final lab project score** will be *score* = *points* - *naming*, i.e., we will deduct 0 pts if all the files were correctly named as requested in §5 and we will deduct 2 pts if the files were incorrectly named. The grader will enter this score in Canvas and Canvas will automatically deduct 0.5 pts for a 1-day late submission on Sun 4 Apr (-10%) and it will deduct 1 pt for a 2-day submission on Mon 5 Apr (-20%).