

p2-birthdays

EECS 183 Project 2: Birthdays

Project Due Friday, September 30, 2022, 11:59pm Eastern

[Direct autograder link](#)



You will write an application to identify the day of the week on which you were born.

Along the way, you will write functions that compute whether a year is a leap year and which day of the week (e.g. Monday, Tuesday) a particular date falls on.

This project is significantly more difficult than Project 1 and you can expect it to take 2 to 3 times longer to complete.

By completing this project, you will learn to:

- Develop programs that are divided into functions
- Compile programs that are not yet complete by using *stubs*
- Verify the correctness of functions by writing test cases
- Distinguish between invalid and bad input when creating test cases
- Implement functions based on a specification and RME comment
- Create algorithms that use conditionals and loops
- Translate mathematical formulas to code

You will apply the following skills you learned in lecture:

- Lecture 3
 - Call math functions like `floor()` and `ceil()` .
- Lecture 4
 - Call functions with multiple parameters
 - Use `return` statements to provide a result from a function
- Lecture 5
 - Create function declarations to allow calls to a function before its code is provided in a function definition
 - Execute the `cin` algorithm to determine how a given user input will be stored into variables
- Lecture 6
 - Use `if` , `else` , and `else if` to conditionally execute code
 - Interpret RME comments
- Lecture 7
 - Write event-controlled loops using `while`
 - Combine loops with `cin` to continue reading until the user is finished
- Lecture 8
 - Write count-controlled loops using `for`

Getting Started

Starter Files

You can download the starter files [using this link](#).

The IDE setup tutorials for Visual Studio and XCode include a video about how to set up a project using the starter files. You can access the tutorials here:

- [Visual Studio](#)
- [XCode](#)

Make sure there are **3 files** in your project: `birthdays.cpp` , `test.cpp` , and `start.cpp` .

Submission and Grading

Submit your code to the [autograder here](#). You receive 4 submits each day and your best overall submission counts as your score. You have one bonus submission that you can use to get a 5th submission on any one day. You can find where the files you need to submit are on your computer

using these steps from [Project 1](#). You will submit **two files**, which must be called `birthdays.cpp` and `test.cpp`.

Here is a grade breakdown:

- *60 points: correctness.* Implement functions in `birthdays.cpp` and create a birthday calculator. To what extent does your code implement the features required by our specification? To what extent is your code consistent with our specifications and free of bugs?
- *10 points: testing.* To what extent is your code tested? Implement the testing functions and submit them via the `test.cpp` file. See the [Testing](#) section for more details.
- *10 points: style.* To what extent is your code written well? To what extent is your code readable? We will only look at your `birthdays.cpp` when determining your style grade. Consult the [EECS 183 Style Guide](#) and check the [Style Checklist](#) at the end of this project's specification for some tips!

The deadline is Friday, September 30, 2021 at 8PM Eastern, with an automatic extension until 11:59PM. If your last submission is on Wednesday, September 28 by 11:59PM, you will receive a 5% bonus. If your last submission is on Thursday, September 30 by 11:59PM, you will receive a 2.5% bonus.

You have 3 late days that you can use any time during the semester for projects. There are 3 late days total, not 3 per project. To use a late day, submit to the autograder after the deadline. It will prompt you about using one of your late day tokens. There are more details about late days [in the syllabus](#).

Understanding the Distribution Code

birthdays.cpp: Starter code for the application you will write in this project. Holds the definitions of required functions and the implementations of a couple functions. We have stubbed all required functions for you.

test.cpp: Testing functions for your `birthdays.cpp` implementation. Holds the definitions of required testing functions. We have stubbed all required functions for you.

start.cpp: This file contains the `main()` function for your program, which allows you to run either the `birthdays` application or your test suite. You do not need to modify this file or submit it to the autograder.

Stubbing functions means adding the minimal necessary code to make a function compile. For example, some of the functions in `birthdays.cpp` have return types of `bool`. We have added `return false` in those functions so that they will compile even if you have not implemented all

the functions yet. Be sure to remove our stubs when you write your own implementation of the function.

Testing Your Setup

Once you have created a project, you should be able to compile and run the distribution code. We have included a `main()` function in `start.cpp` which will allow you to run either your tests or the birthdays application.

```
1  EECS 183 Project 2 Menu Options
2  -----
3  1) Execute testing functions in test.cpp
4  2) Execute birthdays() function in birthdays.cpp
5  Choice --> 1
6  Now testing function isGregorianCalendar()
7  9/2/2019: Expected: 1, Actual: 0
8  1/31/1001: Expected: 0, Actual: 0
9  ...
```

The first test case currently fails because the `isGregorianCalendar()` function is not fully implemented yet.

How to get help

Most students in EECS 183 need help from staff and faculty multiple times each project. We're here for you! Many more people need help with Project 2 than with Project 1.

If your question is about the specification or about something about the project in general, Piazza is the fastest place to get help. You can also ask about your particular code, but in a private post.

You can get 1-1 help over a video call by signing up for office hours at eecs183.org. You can find instructions at eecs183.org > Office Hours.

Collaboration Policy

We want students to learn from and with each other, and we encourage you to collaborate. We also want to encourage you to reach out and get help when you need it. You are encouraged to:

- Give or receive help in understanding course concepts covered in lecture or lab.
- Practice and study with other students to prepare for assessments or exams.
- Consult with other students to better understand project specifications.

- Discuss general design principles or ideas as they relate to projects.
- Help others understand compiler errors or how to debug parts of their code.

To clarify the last item, you are permitted to look at another student's code to help them understand what is going on with their code. You are not allowed to tell them what to write for their code, and you are not allowed to copy their work to use in your own solution. If you are at all unsure whether your collaboration is allowed, please contact the course staff via the admin form before you do anything. We will help you determine if what you're thinking of doing is in the spirit of collaboration for EECS 183.

The following are considered Honor Code violations:

- Submitting others' work as your own.
- Copying or deriving portions of your code from others' solutions.
- Collaborating to write your code so that your solutions are identifiably similar.
- Sharing your code with others to use as a resource when writing their code.
- Receiving help from others to write your code.
- Sharing test cases with others if they are turned in as part of your solution.
- Sharing your code in any way, including making it publicly available in any form (e.g. a public GitHub repository or personal website).

The full collaboration policy can be [found in the syllabus](#).

Problem Statement

In this project, you will develop an application to calculate information regarding specific dates in the past and future, which will allow you to find out what day any given birthday was on.

Your program will provide a menu, implemented using a loop, to obtain user input and calculate birthdays.

Here is an example of what the execution of your final application will look like:

```
1  EECS 183 Project 2 Menu Options
2  -----
3  1) Execute testing functions in test.cpp
4  2) Execute birthdays() function in birthdays.cpp
5  Choice --> 2
6  *****
7      Birthday Calculator
8  *****
9
```

```

10
11  Menu Options
12  -----
13  1) Determine day of birth
14  2) Print the next 10 leap years
15  3) Finished
16
17  Choice --> 1
18
19  Enter your date of birth
20  format: month / day / year --> 1 / 25 / 2000
21
22  You were born on a: Tuesday
23
24  Have a great birthday!!!
25
26  Menu Options
27  -----
28  1) Determine day of birth
29  2) Print the next 10 leap years
30  3) Finished
31
32  Choice --> 3
33
34  *****
35      Thanks for using the Birthday Calculator
36  *****

```

Development Cycle with Functions

In Project 1, you had to divide the program into pieces so you could test each part individually. In Project 2 and later, the program is already divided into functions in the starter code which you can use as the parts to work on. Functions make it much easier to test your code than it was for Project 1. In this section, we will walk through how to write your program one function at a time by writing tests first.

The functions in a program call each other, and it is easiest to start with the functions that do not call any other functions. For example, in this project, `print10LeapYears()` will call `isLeapYear()`, so it makes sense to complete `isLeapYear()` before `print10LeapYears()`. We will be able to test `isLeapYear()` before we write the code that actually uses it in our program. The order you write the functions will be different than the order they appear in `birthdays.cpp`.

We will follow very similar steps to Project 1 to write the `isLeapYear()` function.

Step 1: Make examples of input and output

There are two places to look for information about each function: first this specification, and second the RME attached to the function declaration. The specification is useful for a high-level overview, and the RME is useful for the very specific details. [Here is a link to the section in this specification](#) about `isLeapYear()`. Here is its RME from `birthdays.cpp`:

```

1  /**
2   * Requires: year is a Gregorian year
3   * Modifies: nothing
4   * Effects: returns 'true' if the year is a leap year
5   *           otherwise returns 'false'
6   */
7  bool isLeapYear(int year);

```

In this case, we are able to find:

In the specification	In the RME
The definition of a leap year	What assumptions you can make about the inputs (that year will always be Gregorian)
Examples of leap years and non-leap years	What the input and output types of the function are

Like we did for Project 1, we'll a table of example inputs and outputs. The examples from the spec are already filled in for you:

Input (int)	Output (bool)
1768	true
1800	false
2000	true
[more examples you think of]	

Can 1740 be a valid input to `isLeapYear()`? The answer is no, because the REQUIRES clause states that year must be a Gregorian year (see `isGregorianCalendar()` [for details about Gregorian years](#)). That means that we do not need to check inside `isLeapYear()` whether year is Gregorian

or not – we can make the assumption the programmer will never give an invalid year as input. There is more discussion about these kinds of assumptions you can make in the [Invalid vs. Bad Input](#) section later in the specification.

Step 2: Write Test Cases

In your IDE, pull up the `test.cpp` file. We have provided a `test_isLeapYear()` function that you can use to write your tests inside. Here's an example of how to do it. Your goal is not just to call the function, but to use `cout` statements that will give you proof the function works properly.

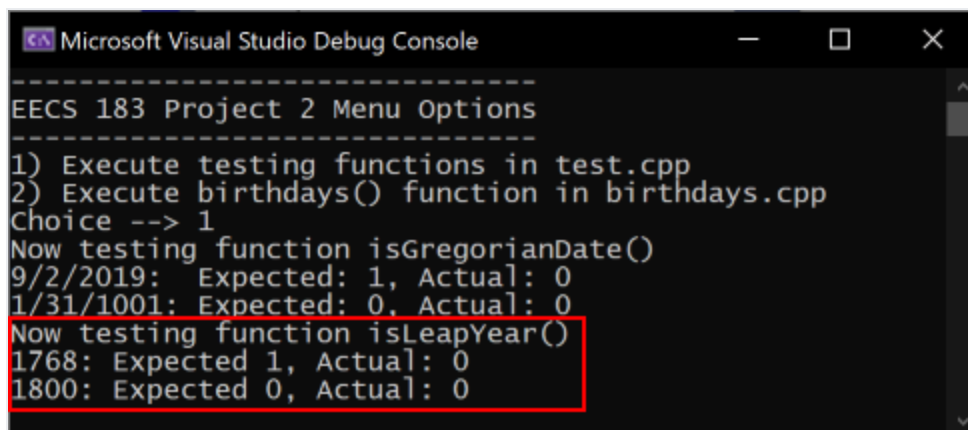
```
1 void test_isLeapYear()
2 {
3
4 }
```

Don't forget to add a call to `test_isLeapYear()` in the `startTests()` function at the top of `tests.cpp` !

The important parts of these test cases are that they:

1. Tell you which input is being tested
2. Show you which input you expect
3. Show you the actual output of the function A test case that does not include all of these parts will not help you find problems in your code.

At this point, you should run your tests. It sounds silly, because you know that they will fail. However, this will help you verify that your tests are actually run and that they do not have compiler errors.



```
Microsoft Visual Studio Debug Console
-----
EECS 183 Project 2 Menu Options
-----
1) Execute testing functions in test.cpp
2) Execute birthdays() function in birthdays.cpp
Choice --> 1
Now testing function isGregorianCalendar()
9/2/2019: Expected: 1, Actual: 0
1/31/1001: Expected: 0, Actual: 0
Now testing function isLeapYear()
1768: Expected 1, Actual: 0
1800: Expected 0, Actual: 0
```

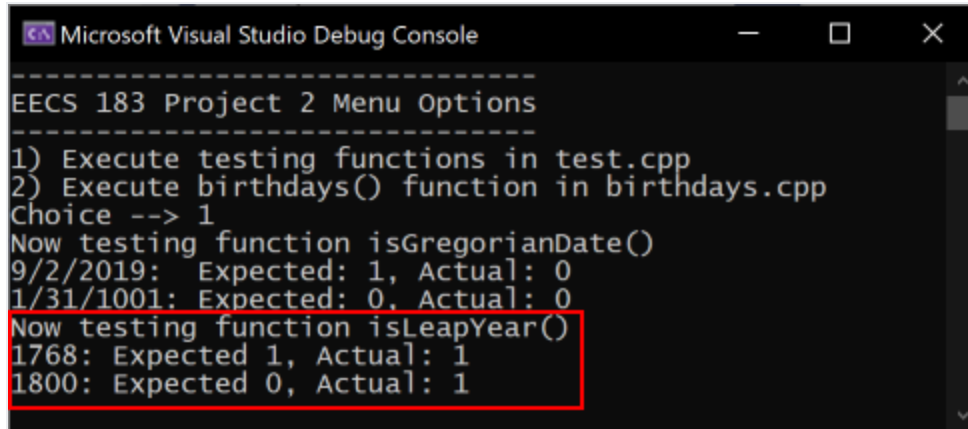
Step 3: Write Code

Now, in `birthdays.cpp`, implement the `isLeapYear()` function. You can use your tests to help you discover the pattern for the algorithm. The tests let you check your work as you go.

For example, this buggy solution:

```
1 bool isLeapYear(int year) {  
2     return (year % 4) == 0;  
3 }
```

gives this output for the test cases:



```
Microsoft Visual Studio Debug Console  
-----  
EECS 183 Project 2 Menu Options  
-----  
1) Execute testing functions in test.cpp  
2) Execute birthdays() function in birthdays.cpp  
Choice --> 1  
Now testing function isGregorianDate()  
9/2/2019: Expected: 1, Actual: 0  
1/31/1001: Expected: 0, Actual: 0  
Now testing function isLeapYear()  
1768: Expected 1, Actual: 1  
1800: Expected 0, Actual: 1
```

Because of this, we know that the problem has to do with something that pertains to 1800 but not 1768, which will help us find the bug.

Step 4: Refine your tests

A significant part of your grade for Project 2 comes from your test cases. We will run your tests against buggy code, and your tests will need to be able to find the bugs. As a rule of thumb, about 10 test cases is often the right amount to find all the bugs in many of the Project 2 functions.

Do not write tests that break the REQUIRES clause of the function you are testing. This can cause you to lose points for your tests. As examples, you should not use 1740 as a test input for `isLeapYear()`, and you should not use May 132, 1700 as an input for `determineDay()`.

Implementation

Overview

Although you will begin work on your project by implementing individual functions, here is how you can expect to string together those functions into your overall program at the end:

- To begin, your program will call `printHeading()` , which will print a heading.
- Next, your program will call `getMenuChoice()` , which will not only print the menu (there is a function that does this) but will be also be used to obtain user input for menu selection.
- Depending upon the value of the user enters for their choice the program will either determine the day of your birthday or print 10 leap years.
- Your program shall continue to calculate birthdays and prompt the user for another choice until a user input of "Finished" is entered.
- When the user input indicates that they've had enough and want out of the Birthday Calculator, your program will call `printCloser()` and the program will exit.

There are many functions that assist in making all of this happen, which you can find in `birthdays.cpp` . You will need to implement the following functions.

All of these functions have RMEs in `birthdays.cpp` with additional details.

getMenuChoice()

This function will print a menu and return the user's selection. You cannot depend upon users to input a number within range, so if the user enters a menu option other than 1, 2, or 3 you need to:

1. Print Invalid menu choice
2. Re-print the menu, and
3. Get another menu choice

Repeat this until a valid menu choice is entered.

isGregorianDate()

`isGregorianDate()` tests whether a date falls within the "Gregorian calendar". The modern rules for dates and leap years started when the "Gregorian calendar" was adopted. Although the Gregorian calendar came into effect in the 1500s, later adjustments to the calendar mean that in this project, any date that falls on or after September 14, 1752 is a Gregorian date, and any date on September 13, 1752 or earlier is not.

For example: 9 / 14 / 1752 and 1 / 10 / 1978 are valid Gregorian dates, while 9 / 10 / 1752 , 9 / 13 / 1752 and 1 / 10 / 1751 are not.

You do **not** need to check in this function whether the month and day represent a valid day of the year.

isLeapYear()

`isLeapYear()` computes whether or not a particular year is a leap year. In the Gregorian calendar, every year evenly divisible by 4 is a leap year, with the exception of the following conditions:

- If the year can be evenly divided by 100, it is NOT a leap year, unless:
- The year is also evenly divisible by 400. Then it is a leap year.

For example: 1768 is a leap year. 1800 is not a leap year. 2000 is a leap year.

isValidDate()

`isValidDate()` verifies that a date is valid, according to the following definition:

1. A valid date is a Gregorian date, according to `isGregorianDate()`.
2. `day` is a valid day in the given month. For example, if month is 4 (April), `day` must be between 1 and 30 inclusive. Note that the number of days in February depends on whether the given year is a leap year.

Month	Days		Month	Days
January	31		July	31
February	29 if leap year, 28 otherwise		August	31
March	31		September	30
April	30		October	31
May	31		November	30
June	30		December	31

Here are some examples of invalid user input to get you started.

- Not a valid month: 13 / 20 / 1980 .
- Not a valid day: 1 / 32 / 1980 .
- Not a valid day: 4 / 31 / 2015 .
- Any day before 9/14/1752: 5 / 23 / 1300 .

In all of these cases, `isValidDate()` should return false.

All dates are entered MONTH / DAY / year and not DAY / MONTH / year. 9 / 12 / 2000 is the 12th of September and not the 9th of December.

determineDay()

`determineDay()` will compute the day of the week on which a date occurs using Zeller's Rule. Using the month, day and year of a date, Zeller's Rule computes the day of the week on which that date occurred/will occur.

$$f = \left(D + \left\lfloor \frac{13(M+1)}{5} \right\rfloor + Y + \left\lfloor \frac{Y}{4} \right\rfloor + \left\lfloor \frac{C}{4} \right\rfloor + 5 \times C \right) \bmod 7$$

- **M** is the number of the month (adjusted, [see below](#)).
- **D** is the day.
- **Y** is the last two digits of the year number (possibly adjusted).
- **C** is the century, i.e. the first two digits of the year number (possibly adjusted).
- $\lfloor x \rfloor$ means "the greatest integer that is smaller than or equal to x ", which is equivalent to the `floor()` function in C++.
- *mod* is the modulus operator (% in C++)

For example:

- If the date is 5/3/2015 (May 3, 2015), then *M* is 5, *D* is 3, *Y* is 15, and *C* is 20.
- If the date is 1/3/2015 (January 3, 2015). then *M* is 13, *D* is 3, *Y* will be 14, and *C* is 20.

Note that none of the constants in Zeller's Formula count as "magic numbers" for the purposes of style grading so long there is a comment explaining that the section of code implements Zeller's Formula.

Calendar Adjustments

Zeller's Rule uses a calendar year beginning in March. To account for this, we count March as month 3, and January and February as months 13 and 14 of the **previous year**. The table below marks conversions.

Our calendar	Zeller's calendar
1/1/2015	13/1/2014
2/1/2015	14/1/2014
3/1/2015	3/1/2015
4/1/2015	4/1/2015
5/1/2015	5/1/2015

Our calendar	Zeller's calendar
6/1/2015	6/1/2015
7/1/2015	7/1/2015
8/1/2015	8/1/2015
9/1/2015	9/1/2015
10/1/2015	10/1/2015
11/1/2015	11/1/2015
12/1/2015	12/1/2015

Converting to day of week

Zeller's rule will return a number f between 0 and 6. This zero-indexed number will correlate to a day of the week in the following manner:

f	Day of the week
0	Saturday
1	Sunday
2	Monday
3	Tuesday
4	Wednesday
5	Thursday
6	Friday

Example Calculations

A detailed example of Zeller's Formula is included below.

Date: January 29, 2064 (i.e., 1 / 29 / 2064)

- $M = 13$ (remember Jan = 13, Feb = 14, March = 3, ...)
- $D = 29$
- $Y = 63$ (by Zeller's calendar, the year is actually 2063)
- $C = 20$

$$\begin{aligned}
 f &= \left(D + \left\lfloor \frac{13(M+1)}{5} \right\rfloor + Y + \left\lfloor \frac{Y}{4} \right\rfloor + \left\lfloor \frac{C}{4} \right\rfloor + 5 \times C \right) \bmod 7 \\
 &= \left(29 + \left\lfloor \frac{13(13+1)}{5} \right\rfloor + 63 + \left\lfloor \frac{63}{4} \right\rfloor + \left\lfloor \frac{20}{4} \right\rfloor + 5 \times 20 \right) \bmod 7 \\
 &= (29 + \lfloor 36.4 \rfloor + 63 + \lfloor 15.75 \rfloor + \lfloor 5 \rfloor + 100) \bmod 7 \\
 &= (29 + 36 + 63 + 15 + 5 + 100) \bmod 7 \\
 &= 248 \bmod 7 \\
 &= 3
 \end{aligned}$$

3 corresponds to a Tuesday, so January 29, 2064 will be a Tuesday.

printDayOfBirth()

`printDayOfBirth()` prints to `cout` the day of the week corresponding to an `f` computed using Zeller's Formula.

- The input `day` is the value calculated using Zeller's rule. Use the table we provided above to determine the day of the week.
- For example, if `day` is 1, `printDayOfBirth()` will print

```
1 Sunday
```

determineDayOfBirth()

`determineDayOfBirth()` reads a date from the user and prints the day of the week corresponding to that date. This function will:

- Prompt for a date
- Check whether that date is valid
 - If the date isn't valid, it will print the error message `Invalid date`.
 - If it is, it will
 - print the day of the week you were born on.
 - tell you to `Have a great birthday!!!`

This function will contain the following prompts, which print depending on program's execution:

```
1 Enter your date of birth
2 format: month / day / year -->
3 Invalid date
4 You were born on a:
5 Have a great birthday!!!
```

Make sure you use these exact prompts, letter-for-letter.

This function will call the following functions:

- `isValidDate()`
- `determineDay()`
- `printDayOfBirth()`

You can assume dates will always be entered with spaces around the `/` . For example, we may test the input `12 / 20 / 1980` on the autograder, but we will never test `12/20/1980` .

Even if the user enters a birthday that has not yet taken place, your program must still use the past tense `You were born on a: .`

`print10LeapYears()`

`print10LeapYears()` prompts the user for a Gregorian year and prints the first 10 leap years occurring after (**not including**) the input year. If the year is invalid, it prints nothing. The first Gregorian year was 1752.

This function will contain the following prompts, which print depending on program's execution:

```
1  Enter year -->
2  Leap year is
```

Putting it Together

You will need to reference both this specification and the RMEs when implementing these functions.

Once you have written and tested each of the above functions, it is time to combine everything in `birthdays()` and do further testing with your new debugging skills. Be sure that your program behaves as illustrated in the [Sample Output](#).

`birthdays()` will call the following function(s):

- `printHeading()`
- `getMenuChoice()`
- `determineDayOfBirth()`
- `print10LeapYears()`
- `printCloser()`

It is important to note that implementing `birthdays()` might not be as simple as calling the above functions in the given order. You will have to consider the intended behavior of the program once `birthdays()` is called from `main()` in `start.cpp`. Do you notice that based on input choices by the user, some of the above functions might somehow be called more than just once? Take a look at all of the [Sample Output](#) and consider how you might get `birthdays()` to behave like this.

Function Table

Function	Other functions it should call
<code>getMenuChoice</code>	<code>printMenu</code>
<code>isGregorianDate</code>	Does not utilize any other functions
<code>isLeapYear</code>	Does not utilize any other functions
<code>isValidDate</code>	<code>isGregorianDate</code> , <code>isLeapYear</code>
<code>determineDay</code>	Does not utilize any other functions
<code>printDayOfBirth</code>	Does not utilize any other functions
<code>determineDayOfBirth</code>	<code>isValidDate</code> , <code>determineDay</code> , <code>printDayOfBirth</code>
<code>print10LeapYears</code>	<code>isLeapYear</code>
<code>birthdays</code>	<code>printHeading</code> , <code>getMenuChoice</code> , <code>determineDayOfBirth</code> , <code>print10Lea</code>

Testing

As part of this project, you will also submit code in `test.cpp` that tests the functions you implement in `birthdays.cpp`. The autograder will run your tests against buggy programs in order to see if your tests can expose the bugs. Because you will not know the exact nature of the bugs on the autograder, you will need to build comprehensive tests.

As a rule of thumb, every function needs at least 5-10 test cases to find a reasonable set of bugs.

Invalid vs. Bad Input

You will testing your program and functions for **invalid input** rather than **bad input**. Invalid input has the correct form but invalid values. Bad input has an incorrect form. For example, for the `determineDayOfBirth()` function, here are examples of invalid input and bad input:

Invalid (test these)	Bad (don't test)
12 / 40 / 2000	1/1/2019
40 / 10 / 2104	January / 10 / 2104
0 / 1 / 1910	1 1 2019
1 / 1 / 1200	asdoiw8032hfg80r

The specification and RMEs for a function will define invalid input for a specific function.

Something that breaks the REQUIRES clause is always bad input, and should not be tested.

getMenuChoice()

`getMenuChoice()` is a good function to test early, since you do not need any other functions implemented for it to work. To test `getMenuChoice()`, you need to check it against good input and invalid input.

We have provided two initial test cases for `getMenuChoice()`. Because it reads from `cin`, you need to type input as part of the test.

isValidDate()

Because `isValidDate()` does not read from `cin`, it can be tested automatically, without any user input. To test `isValidDate()`, write your own tests inside the function called `test_isValidDate()`. Here are some ideas for test cases:

- Values for valid and invalid months.
- Values for valid and invalid days.
- Days that are valid for Jan but not for Feb, etc.
- Days that are valid for March but not for April, etc.
- Values for valid and invalid Gregorian years.
- Values for February 29 if it is a leap year and if it isn't a leap year.

Be sure to call your `test_isValidDate()` function within `runTests()`.

isLeapYear()

Testing `isLeapYear()` will be similar to `isValidDate()`. Note, however, that `isLeapYear()` includes "year must be a Gregorian year" in its REQUIRES clause. All your tests must respect this

REQUIRES clause. This means that even though it is legal C++ to write the following test, it should not be used as a test case:

```
1 // invalid test case because 1750 is not a Gregorian year
2 cout << "1750: Expected: 1, actual: " << isLeapYear(1750) << endl;
```


Other Functions


Test functions for the other functions in `birthday.cpp` have been stubbed for you in `test.cpp`. These functions will be tested in similar ways to `getMenuChoice()`, `isValidDate()`, and `isLeapYear()`.

Bugs To Expose

After you submit your test suite to the autograder, you might see output that looks like this:

Mutation Testing Suites






Suite Name	Student Tests	Score
Student Birthdays Tests		7.50/10

Compile: 


Return Code: 0

Setup Error Output: No Output

Bugs Exposed: 5

-  CHECK_GREG_DAY
-  CHECK_GREG_BASE
-  CHECK_GREG_BASE2
-  CONVERT_MONTHS
-  CHECK_VALID_DATE

Valid test cases you submitted:

-  test.cpp

That means that your test suite exposed 5 out of 8 bugs in the staff's "buggy" implementations of `birthdays.cpp` and your score for the test suite is 7.5 out of 10 points.

There are a total of 8 unique bugs to find in our implementations. Your tests do not need to expose all of the bugs to receive full points for the project. The autograder will tell you the names of the bugs that you have exposed, from the following set:

- CHECK_GREG_MONTH
- CHECK_GREG_DAY
- CHECK_GREG_BASE

- CHECK_GREG_BASE2
- CHECK_LEAP_YEAR
- VALID_LEAP_YEAR
- CONVERT_MONTHS
- CHECK_VALID_DATE

Sample Output

Here are a few examples of the way your program output should look, where red underlined text represents some user's input. Make sure to use [a diffchecker](#) when comparing your program's output to these runs.

Sample Run 1

```
1  -----
2  EECS 183 Project 2 Menu Options
3  -----
4  1) Execute testing functions in test.cpp
5  2) Execute birthdays() function in birthdays.cpp
6  Choice --> 2
7  *****
8      Birthday Calculator
9  *****
10
11
12  Menu Options
13  -----
14  1) Determine day of birth
15  2) Print the next 10 leap years
16  3) Finished
17
18  Choice --> 1
19
20  Enter your date of birth
21  format: month / day / year --> 9 / 31 / 1980
22
23  Invalid date
24
25  Menu Options
26  -----
27  1) Determine day of birth
```

```

28  2) Print the next 10 leap years
29  3) Finished
30
31  Choice --> 1
32
33  Enter your date of birth
34  format: month / day / year --> 1 / 25 / 1956
35
36  You were born on a: Wednesday
37
38  Have a great birthday!!!
39
40  Menu Options
41  -----
42  1) Determine day of birth
43  2) Print the next 10 leap years
44  3) Finished
45
46  Choice --> 3
47
48  *****
49      Thanks for using the Birthday Calculator
50  *****

```

Sample Run 2

```

1  -----
2  EECS 183 Project 2 Menu Options
3  -----
4  1) Execute testing functions in test.cpp
5  2) Execute birthdays() function in birthdays.cpp
6  Choice --> 2
7  *****
8      Birthday Calculator
9  *****
10
11
12  Menu Options
13  -----
14  1) Determine day of birth
15  2) Print the next 10 leap years
16  3) Finished
17

```

```
18  Choice --> 5
19
20  Invalid menu choice
21
22  Menu Options
23  -----
24  1) Determine day of birth
25  2) Print the next 10 leap years
26  3) Finished
27
28  Choice --> 3
29
30  *****
31      Thanks for using the Birthday Calculator
32  *****
```

Sample Run 3

```
1  -----
2  EECS 183 Project 2 Menu Options
3  -----
4  1) Execute testing functions in test.cpp
5  2) Execute birthdays() function in birthdays.cpp
6  Choice --> 2
7  *****
8      Birthday Calculator
9  *****
10
11
12  Menu Options
13  -----
14  1) Determine day of birth
15  2) Print the next 10 leap years
16  3) Finished
17
18  Choice --> 1
19
20  Enter your date of birth
21  format: month / day / year --> 9 / 13 / 1752
22
23  Invalid date
24
25  Menu Options
```

```

26  -----
27  1) Determine day of birth
28  2) Print the next 10 leap years
29  3) Finished
30
31  Choice --> 1
32
33  Enter your date of birth
34  format: month / day / year --> 19 / 13 / 1982
35
36  Invalid date
37
38  Menu Options
39  -----
40  1) Determine day of birth
41  2) Print the next 10 leap years
42  3) Finished
43
44  Choice --> 1
45
46  Enter your date of birth
47  format: month / day / year --> 9 / 13 / 1982
48
49  You were born on a: Monday
50
51  Have a great birthday!!!
52
53  Menu Options
54  -----
55  1) Determine day of birth
56  2) Print the next 10 leap years
57  3) Finished
58
59  Choice --> 3
60
61  *****
62      Thanks for using the Birthday Calculator
63  *****

```

Sample Run 4

```

1  -----
2  EECS 183 Project 2 Menu Options

```

```
3 -----
4 1) Execute testing functions in test.cpp
5 2) Execute birthdays() function in birthdays.cpp
6 Choice --> 2
7 *****
8     Birthday Calculator
9 *****
10
11
12 Menu Options
13 -----
14 1) Determine day of birth
15 2) Print the next 10 leap years
16 3) Finished
17
18 Choice --> 1
19
20 Enter your date of birth
21 format: month / day / year --> 9 / 24 / 1980
22
23 You were born on a: Wednesday
24
25 Have a great birthday!!!
26
27 Menu Options
28 -----
29 1) Determine day of birth
30 2) Print the next 10 leap years
31 3) Finished
32
33 Choice --> 5
34
35 Invalid menu choice
36
37 Menu Options
38 -----
39 1) Determine day of birth
40 2) Print the next 10 leap years
41 3) Finished
42
43 Choice --> -2
44
45 Invalid menu choice
46
```

```

47  Menu Options
48  -----
49  1) Determine day of birth
50  2) Print the next 10 leap years
51  3) Finished
52
53  Choice --> 2
54
55  Enter year --> 1972
56
57  Leap year is 1976
58  Leap year is 1980
59  Leap year is 1984
60  Leap year is 1988
61  Leap year is 1992
62  Leap year is 1996
63  Leap year is 2000
64  Leap year is 2004
65  Leap year is 2008
66  Leap year is 2012
67
68  Menu Options
69  -----
70  1) Determine day of birth
71  2) Print the next 10 leap years
72  3) Finished
73
74  Choice --> 3
75
76  *****
77      Thanks for using the Birthday Calculator
78  *****

```

Sample Run 5

```

1  -----
2  EECS 183 Project 2 Menu Options
3  -----
4  1) Execute testing functions in test.cpp
5  2) Execute birthdays() function in birthdays.cpp
6  Choice --> 2
7  *****
8      Birthday Calculator

```



```
9  *****
10
11
12  Menu Options
13  -----
14  1) Determine day of birth
15  2) Print the next 10 leap years
16  3) Finished
17
18  Choice --> 2
19
20  Enter year --> 1842
21
22  Leap year is 1844
23  Leap year is 1848
24  Leap year is 1852
25  Leap year is 1856
26  Leap year is 1860
27  Leap year is 1864
28  Leap year is 1868
29  Leap year is 1872
30  Leap year is 1876
31  Leap year is 1880
32
33  Menu Options
34  -----
35  1) Determine day of birth
36  2) Print the next 10 leap years
37  3) Finished
38
39  Choice --> 2
40
41  Enter year --> 1600
42
43
44  Menu Options
45  -----
46  1) Determine day of birth
47  2) Print the next 10 leap years
48  3) Finished
49
50  Choice --> 2
51
52  Enter year --> 2013
```

```
53
54  Leap year is 2016
55  Leap year is 2020
56  Leap year is 2024
57  Leap year is 2028
58  Leap year is 2032
59  Leap year is 2036
60  Leap year is 2040
61  Leap year is 2044
62  Leap year is 2048
63  Leap year is 2052
64
65  Menu Options
66  -----
67  1) Determine day of birth
68  2) Print the next 10 leap years
69  3) Finished
70
71  Choice --> 3
72
73  *****
74      Thanks for using the Birthday Calculator
75  *****
```

Roadmap and Timeline

Completing this project essentially boils down to implementing the stubbed functions we have given you in `birthdays.cpp` , testing your implementation of those functions in `test.cpp` , and then putting it all together in `birthdays()` .

This additional element of testing adds a new layer to your projects, and is a critical component of ensuring that you can complete this project smoothly. It is important that you follow the suggestions given in [Development Cycle with Functions](#) for one function at a time rather than writing all functions first and then testing them all after. Many students find that if `function a` has a bug and that function is called by `function b` , it could appear that `function b` does not work properly when in reality there is nothing wrong with the implementation. Attacking the project one function at a time and then testing it before moving on can help you complete this project more smoothly. Having this done before writing `birthdays()` is critical for the same reason.

Remember that you can submit to the autograder even if you only have some parts of the project completed.

Timeline

As an approximate timeline, you are on track if by:

- September 21: You have completed testing and implementing `getMenuChoice()` and `isGregorianCalendar()`.
- September 23: You have completed testing and implementing `isLeapYear()`, `isValidDate()`, and `determineDay()`.
- September 27: You have completed testing and implementing `printDayOfBirth()`, `determineDayOfBirth()`, and `print10LeapYears()`.
- September 28: You have written `birthdays()` and are completing your final debugging.
- September 29: You have made your last submission to the autograder for 5% extra credit.

Style

Your code must follow the [EECS 183 style guide](#). Keep in mind that only `birthdays.cpp` is graded for style. Your `test.cpp` will not be style graded in any way.

Style Rubric

Top Comment

Must have name, username, program name, and project description at the top of the file.

If all or part of the top comment is missing, take 1 point off.

Readability

-1 point for each of the following categories:

Indentations

- Not using a consistent number of spaces for each level of code indentation
 - This includes using tabs on some lines and spaces on others
- Not indenting lines at all
- Failing to indent the blocks of code inside curly braces

Spacing

- Not putting a space around operators (e.g., `5*7` instead of `5 * 7` or `count=0;` instead of `count = 0;`)
- Not putting a space before and after the stream extraction operator (e.g., `cin>>month>>slash;` instead of `cin >> month >> slash;`)
- Not putting a space between `if` (or `while`) and condition to be evaluated

```
1 // good
2 if (x == 1)
3 // bad
4 if(x == 1)
```

- Spaces around parentheses: Function headers and calls should not have a space between the function name and opening parenthesis.

```
1 // good
2 printDayOfBirth(2);
3 // bad
4 printDayOfBirth (2);
```

Bracing

- Using a mix of Egyptian-style and hanging braces
 - Egyptian-style: `'{'` at the end of a statement
 - Hanging: `'{'` on its own line
- Braces should always be used for conditionals, loops, and functions. Examples:

```
1 // good
2 if (x == 1) {
3     return false;
4 }
5 if (x == 2)
6 {
7     return true;
8 }
9 // bad
10 if (x == 1) return false;
11 if (x == 2)
12     return true;
```

Variables

- Variable names not meaningful
 - do NOT take off for `M`, `C`, `D`, `Y` in Zeller's formula in `determineDays` .

- Inconsistent variable naming style (camelCase vs. snake_case)
 - Excluding const variables, which are always SNAKE_CASE
- Mixing camelCase and snake_case in names of tester functions is OK. (e.g. test_isLeapYear)
- Not using all uppercase SNAKE_CASE for const variable names
- Using variable types that do not make sense in context (e.g. using double instead of int for M , Y , or C)

Line Limit and Statements

- Going over 80 characters on a line (includes lines of comments and lines of code)
- More than one statement on a single line.
 - A statement ends in a semicolon.
 - Do not count off for multiple statements as part of a for loop declaration line

Comments

- Commenting on the end of a line of code

```
1 // A comment should be placed before a line of code
2 int count = 0; // not on the same line as the code
```

- Insufficient comments or excessive comments
 - Code should be thoroughly commented such that lines' functionality is apparent from comments alone or from quickly glancing at code
 - Example of appropriate comment:

```
1 // convert cups of flour to bags of flour
2 int bagFlour = ceil((CUPS_FLOUR * numBatches) /
3                     CUPS_IN_LB_FLOUR);
```

- Example of excessive comments:

```
1 // declare variable
2 int bagFlour;
3 // calculate bags of flour
4 bagFlour = ceil((CUPS_FLOUR * numBatches) /
5                 CUPS_IN_LB_FLOUR);
```

- Unneeded comments left in the code, such as:

```
1 // your code goes here
```

- Commented out code, such as:

```
1 // int numBatches = people / 12;  
2 int numBatches = ceil(people / NUM_IN_BATCH);
```

RMEs

- Missing RMEs for any of the defined functions (exception: main). This includes functions from the distribution code and any functions created by the student.
 - Not having RMEs for tester functions is OK.
- Repeating RMEs for the same function.

bools

- Returning 0 and 1 instead of true and false for a bool function

Coding Quality

-2 for each of the following categories:

Global Variables

- Global variables not declared as const

Magic Numbers

- Do not take off points for using 13, 5, 4, and 7 in Zeller's formula
 - Should have a comment stating the purpose of Zeller's formula
- Do not take off points for using 1752, 14, and 9 in isGergorian
 - Should have a comment stating the purpose of these numbers
- Do not take off points for using 0-6 with the days of the week, 1...-12 with months, and 1...-31 for days even if comments are absent
 - Using 13 and 14 for months should have a comment
 - Using 4, 100, and 400 should have a comment

Egregious code

- Logic that is clearly too involved
 - e.g. instead of basing numbers on conversions, writing:

```
1 if(year >= 1700 && year < 1800) {  
2     century = 17;  
3 } else if (year >= 1800 && year < 1900) {  
4     century = 18;  
5 }
```

and so on

Function Misuse

- Not calling helper functions where appropriate
 - e.g. not calling `isLeapYear()` inside `isValidDate()`
 - not calling `isGregorianCalendar()` inside `isValidDate()`

Checklist

To maximize your style points, be sure to follow this non-exhaustive checklist:

- Review the [EECS 183 Style Guide](#). While all sections are important and relevant for this project, pay particular attention to functions, conditionals, and loops.
- Review the style grading rubric above.
- Be sure you've included your name, your username and a small description of the program in the header comments
- Be sure that your code is well-commented. It might at first seem that the algorithm is straight-forward and self-explanatory, but would you remember all the details a month from now? And it's much easier for the staff to help you with your code if you have comments!
- Be sure to use `camelCase` for names of variables and functions in this project. We used `camelCase` for the functions we ask you to implement, and it is important that you stay consistent.
- Be sure that your variable names are descriptive. Don't use names like `d` , `Y` or `C` . Instead, use `day` , `year` , `century` or the like.
- Use functions that we've asked you to implement and/or your own helper functions to simplify logic and to minimize repetition of similar code.
- Don't use global variables. However, global constants are OK.
- When defining and calling functions, make sure not to have spaces around `(` and `)` . For example:

```
1 cout << ceil(numberOfCookies);
```

However, when using `if/else` statements and loops, be sure to put a space after `if` , `else` , `for` and `while` , as this example demonstrates:

```
1  if (numberOfCookies > numberOfPeople) {  
2      // do something  
3  }
```

- Remember that all lines must be 80 characters or less.

Optional Warm-Up Questions

- What does the `!` operator do?
- How can you determine if a number is evenly divisible by another? In other words, how can you check if the remainder is zero?
- How can you get the last digit of a number?
- How can you get the first digit of a number between 100 and 999?
- What's the purpose of a function, such as `sqrt` or `getline`?
- What's the difference between a function's declaration (aka prototype) and a definition (aka implementation).
- Suppose you have this code that defines a function called `half` :

```
1  double half(int number) {  
2      double answer = (double) number / 2;  
3      return answer;  
4  }
```

- What's `int number` on line 1?
 - What does `(double)` (the one inside parentheses) do on line 2?
 - How would you use this function in `main`?
- Write a function that accepts one argument, an `int`, and checks if it's even by returning `true` if it is and `false` otherwise.
- Write the line of code that checks if 7 is an even number using the above function.
- Head over to the [EECS 183 Style Guide](https://web.archive.org/web/20221002153640/https://eecs183.github.io/p2-birthdays/) and read the section about Functions, Conditions and Loops.
- What's the purpose of the `Requires` clause?
- How are while loops different from for loops? When would you use each?

- What's a breakpoint?
- What's the difference between step over and step into commands?
- How do you determine the value a variable has on a specific line of code?

