Assignment 2
C/C++ Programming II

C2A2 General Information

Assignment 2 consists of FOUR (4) exercises:

C2A2E1 C2A2E2 C2A2E3 C2A2E4

All requirements are in this document.

Get a Consolidated Assignment 2 Report (optional)

If you would like to receive a consolidated report containing the results of the most recent version of each exercise submitted for this assignment:

`Send an empty-body email to the assignment checker with the subject line **C2A2_177752_U09845800** and no attachments.

Inspect the report carefully since it is what I will be grading. You may resubmit exercises and report requests as many times as you wish before the assignment deadline.

C2A2 General Information, continued

How many bits are in a *byte*?

Contrary to what many people mistakenly think, the number of bits in a byte is not necessarily 8. Instead, a byte is more accurately defined in the language standards as an "addressable unit of data storage large enough to hold any member of the basic character set of the execution environment". Specifically, this means that the number of bits in a byte is dictated by and is equal to the number of bits in type **char**. While on the vast majority of implementations the number of bits in such an "addressable unit" is 8, there have been implementations in which this has not been true and has instead been 6 bits, 9 bits, or some other value. To maintain compatibility with all standards-conforming implementations the macro `CHAR_BIT` has been defined in standard header file `limits.h` (`climits` in C++) to represent the number of bits in a byte on the implementation hosting that file. The implication of this is that no portable program will ever assume any particular number of bits per byte but will instead use `CHAR_BIT` in code whenever the actual number is needed. This ensures that the code will remain valid even if moved to an implementation having a different number of bits per byte. It is important to note that the data type of `CHAR_BIT` is always **int**.

How many bits are in an arbitrary data type?

The **sizeof** operator produces a count of the number of bytes of storage required to hold an object of the data type of its operand (note 2.12). Except for type **char**, however, not all of the bits used for the storage of an object are necessarily used to represent its value. Instead, some bits may simply be unused "padding" needed only to enforce memory alignment requirements. As a result, simply multiplying the number of bits in a **char** (byte) by the number of bytes in an arbitrary data type does not necessarily produce the number of bits used to represent that data type's value. Instead, the actual number of "active" bits must be determined in some other way.

C2A2E1 *(3 points – C Program)*

Exclude any existing source code files that may already be in your IDE project and add two new ones, naming them **C2A2E1_CountBitsM.h** and **C2A2E1_CountIntBitsF.c**. Also add instructor-supplied source code file **C2A2E1_main-Driver.c**. Do not write a `main` function! `main` already exists in the instructor-supplied file and it will use the code you write.

File **C2A2E1_CountBitsM.h** must contain a macro named `CountBitsM`.
`CountBitsM` syntax:
This macro has one parameter and produces a value of type **int**. There is no prototype (since macros are never prototyped).
Parameters:
`objectOrType` – any expression with an object data type (24, temp, printf("Hello"), etc.), or the literal name of any object data type (**int**, **float**, **double**, etc.)
Synopsis:
Determines the number of bits of storage used for the data type of `objectOrType` on any machine on which it is run. This is an extremely trivial macro.
Return:
the number of bits of storage used for the data type of `objectOrType`

File **C2A2E1_CountIntBitsF.c** must contain function `CountIntBitsF` and no `#define` or `#include`.
`CountIntBitsF` syntax:

```
int CountIntBitsF(void);
```

Parameters:
none
Synopsis:
Determines the number of bits used to represent a type **int** value on any machine on which it is run.
Return:
the number of bits used to represent a type **int** value

`CountBitsM` and `CountIntBitsF` must:

1. not assume a **char**/byte contains 8 or any other specific number of bits.
2. not call any function.
3. not use any external variables.
4. not perform any right-shifts.
5. not display anything.

`CountBitsM` must:

1. not use any variables.
2. use a macro from header file `limits.h`

`CountIntBitsF` must:

1. use a **for** statement.
2. not use more than two variables, an **if** statement, a **break** statement, or a **continue** statement.
3. not perform any multiplications or divisions.
4. not use a macro or anything from a header file.
5. not be in a header file.

Assignment checker warnings regarding instructor-supplied file **C2A2E1_main-Driver.c** are always due to problems in your `CountBitsM` macro.

Questions: (Place your answers as comments in the "Title Block" of file **C2A2E1_CountIntBitsF.c**.)
If run on the same implementation, could the value produced by `CountBitsM` for type **int** be different from the value produced by `CountIntBitsF`? Why or why not? The correct answers have nothing to do with the value CHAR_BIT.

Submitting your solution

`Send an empty-body email to the assignment checker with the subject line **C2A2E1_177752_U09845800** and with all three source code files attached.

See the course document titled "How to Prepare and Submit Assignments" for additional exercise formatting, submission, and assignment checker requirements.

Hints:

In macro `CountBitsM` multiply the number of bytes in the data type of its argument by the implementation-dependent number of bits in a byte.

In function `CountIntBitsF` start with a value of 1 in a type **unsigned int** variable and left-shift it one bit at a time, keeping count of number of shifts, until the variable's value becomes 0. If you use a plain **int** (which is always signed) for this purpose you have made a portability mistake.

C2A2E2 *(5 points – C++ Program)*

Exclude any existing source code files that may already be in your IDE project and add two new ones, naming them **C2A2E2_CountIntBitsF.cpp** and **C2A2E2_Rotate.cpp**. Also add instructor-supplied source code file **C2A2E2_main-Driver.cpp**. Do not write a `main` function! `main` already exists in the instructor-supplied file and it will use the code you write.

File **C2A2E2_CountIntBitsF.cpp** must contain an exact copy of the `CountIntBitsF` function you wrote for the previous exercise, except omit the keyword **`void`** from its parameter list.

File **C2A2E2_Rotate.cpp** must contain function `Rotate` and no `#define` or `#include`.
`Rotate` syntax:

```
unsigned Rotate(unsigned object, int count);
```

Parameters:

`object` – the value to rotate

`count` – the number of bit positions & direction to rotate: negative=>left and positive=>right

Synopsis:

Produces the value of parameter `object` as if it had been rotated by the number of positions and in the direction specified by `count`.

Return:

the rotated value

The `Rotate` function must:

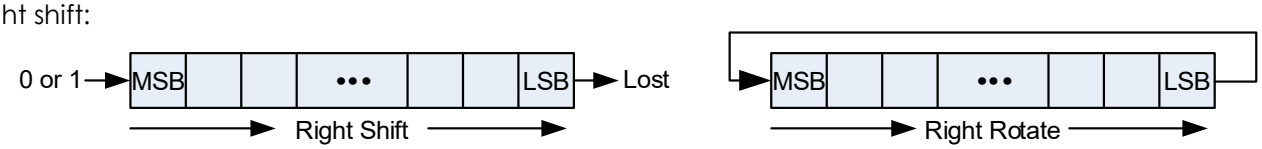
1. Call `CountIntBitsF` once and only once.
2. not call any function other than `CountIntBitsF`.
3. not assume a **char**/byte contains 8 or any other specific number of bits.
4. not use loops, recursion, **sizeof**, macros, or anything from any header file.
5. not implement a special case for handling a `count` value of 0.
6. not display anything.

Here are some examples for a 32-bit type **unsigned int**:

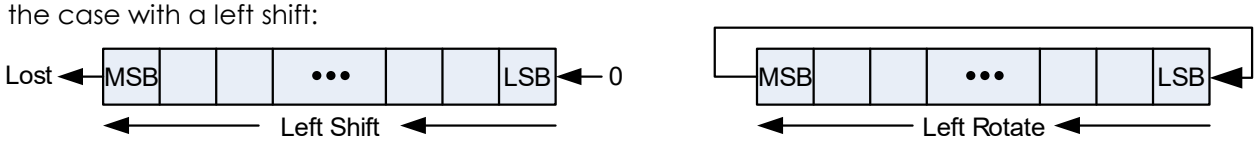
Function Call (right rotate)	Return Value		Function Call (left rotate)	Return Value
`Rotate(0xa701, 1)`	`0x80005380`		`Rotate(0xa701, -1)`	`0x14e02`
`Rotate(0xa701, 225)`	`0x80005380`		`Rotate(0xa701, -225)`	`0x14e02`
`Rotate(0xa701, 1697)`	`0x80005380`		`Rotate(0xa701, -1697)`	`0x14e02`
`Rotate(0xdefacebd, 2)`	`0x77beb3af`		`Rotate(0xdefacebd, -2)`	`0x7beb3af7`
`Rotate(0xdefacebd, 194)`	`0x77beb3af`		`Rotate(0xdefacebd, -194)`	`0x7beb3af7`
`Rotate(0xdefacebd, 5378)`	`0x77beb3af`		`Rotate(0xdefacebd, -5378)`	`0x7beb3af7`

Explanation

When a pattern is "shifted" each bit shifted off the end is simply lost. In "rotation", however, the end bits are treated as if they are connected. That is, when a pattern is right-rotated the LSB (least significant bit) is placed into the MSB (most significant bit MSB) rather than being lost, as would be the case with a right shift:



Conversely, when a pattern is left-rotated the MSB is placed into the LSB rather than being lost, as would be the case with a left shift:



Submitting your solution

`Send an empty-body email to the assignment checker with the subject line **C2A2E2_177752_U09845800** and with all three source code files attached.

See the course document titled "How to Prepare and Submit Assignments" for additional exercise formatting, submission, and assignment checker requirements.

Hints:

1. The number of bits in the signed and unsigned versions of the same basic integer type (int vs. unsigned int, long vs. unsigned long, etc.) is always the same.

2. Bit patterns for values greater than 9 must always be represented in hexadecimal; representing them in decimal is cryptic and inappropriate. Shift/rotation counts should always be represented in decimal.

3. For shifting operations, the language standards state that if the value of the right operand (the shift count) is negative or is greater than or equal to the width of the promoted left operand the behavior is undefined.

4. Although rotation can be achieved one bit at a time using a loop or recursion, it is extremely inefficient and should never be done. Instead, a rotation of any size can always be accomplished with only two shift operations. To demonstrate this, assume an 8-bit data type contains a pattern that must be rotated right by 5 positions. If done bit at a time with a loop the following would occur:

```
00100001   <- arbitrary original pattern
10010000   <- rotated right by 1 bit
01001000   <- rotated right by 2 bits
00100100   <- rotated right by 3 bits
00010010   <- rotated right by 4 bits
00001001   <- done - rotated right by 5 bits
```

   However, if done the efficient way the unaltered original pattern will be shifted right by 5 bits, the unaltered original pattern will be shifted left by the total number of bits it contains minus 5, and the results of the two shifts will be bitwise-OR'd together to produce the final result as follows:

```
00100001   <- original pattern from above
00000001   <- original pattern shifted right by 5 bits
00001000   <- original pattern shifted left by 8 minus 5 bits
00001001   <- done - bitwise-OR of the two shifted patterns
```

   If you understand what is going on in the right rotate example above you should be able to easily develop a left rotate version too.

C2A2E3 *(6 points – Table only – No program required)*

Figure 8 of note 12.4D and figure 5 of note12.6C in the course book illustrate the final state of call stacks before any function returns have begun. Your task is to create a similar diagram in instructor-supplied text file **C2A2E3_StackDiagram.txt** to represent the operation of the program below.

Assume:

- Any necessary header files and prototypes are present.
- "Pascal" calling convention
- **short**: 6 bytes, **int**: 8 bytes, **long**: 9 bytes, addresses(pointers): 4 bytes

```
int main()  <-------  The "startup" function calls main.
{
    int result = ready();
    return 0;
}

long ready()
{
    int temp = gcd(64, 48);
    return temp;
}

short gcd(long x, short y)
{
    if (y == 0)
        return x;
    return gcd(y, x % y);
}
```

Function `main`	
Operation	Instruction Address
assignment to `result`	25A*h*

Function `ready`	
Operation	Instruction Address
assignment to `temp`	23A*h*

Function `gcd`	
Operation	Instruction Address
2nd **return** in `gcd`	5BA*h*

Procedure and Requirements:

Please refer to file **C2A2E3_StackDiagram.txt** while reading the following requirements. It contains:

1. A title block.
2. An empty stack diagram for your stack information.
3. A sample stack diagram (for a different program) that illustrates the required format and syntax.

IMPORTANT: The assignment checker analyzes your diagram and provides an error report. If you do not use the required format and syntax, the report may be inaccurate and you could lose significant credit.

1. Modify the **C2A2E3_StackDiagram.txt** title block to contain your own information and add the following two-item "startup" stack frame to the empty stack diagram:

```
BP+4h    E59h  1011h   Function Return Address   (4 bytes)
BP       E55h     0h   Previous Frame Address    (4 bytes)
```

2. After reading the "Optional Suggestion" on the next page, complete the stack diagram by pushing items for any additional frames:
   a. There must be a dividing line before each frame that indicates the name of the function it represents. If the function is being used recursively, that name must be followed by a space and a number indicating the frame's recursive level.
   b. If a stack item represents a variable, place its name in the "Description" column.

Continued on the next page…

…C2A2E3 requirements continued

3. Numeric values must be decimal or hexadecimal according to the guidelines below. Decimal values do not have a suffix while hexadecimal values must have an ***h*** suffix.
    a. The values of variables **x** and **y** and the number of bytes in each item must be decimal.
    b. All addresses must be hexadecimal.
    c. The **BP** offset values of relative addresses may be decimal or hexadecimal.

4. Use a double question mark for the values of:
    a. All return objects.
    b. Variables that are assigned values after a function returns.
    c. Values for which insufficient information is provided.

5. General formatting:
    a. "Hard" tab characters are not allowed.
    b. Columns must be aligned and there must be nothing between them but spaces.
    c. Do not number lines or add additional information.
    d. No line may exceed 80 columns.

6. Before submitting your modified **C2A2E3_StackDiagram.txt** file to the assignment checker, delete everything in it except your modified title block and completed stack diagram.

Waypoints:
Verifying the following can help you determine if there might be a problem:
1. There should be exactly **25** items on the stack.
2. There should be exactly **8** double question mark stack values.
3. The **gcd** function should have **3** stack frames, each containing **5** items.
4. The absolute address of the last item in the last stack frame should be **DCD*h***.

Optional Suggestion:
As each item is pushed onto the stack, I suggest only populating its "Description" and "Item Size" fields. Once this is complete, go back through the items again and populate just the "Abs Adr" field, then do it twice more for the "Rel Adr" and "Stack Value" fields. I recommend this order because the "Rel Adr" **BP** offset values depend upon values in other fields. The only downside of this approach is that slightly more effort is needed to align columns.

Submitting your solution

`Send an empty-body email to the assignment checker with the subject line **C2A2E3_177752_U09845800** and with your modified **C2A2E3_StackDiagram.txt** text file attached.

See the course document titled "How to Prepare and Submit Assignments" for additional exercise formatting, submission, and assignment checker requirements.

C2A2E4 *(6 points – C++ Program)*

Exclude any existing source code files that may already be in your IDE project and add two new ones, naming them **C2A2E4_OpenFile.cpp** and **C2A2E4_Reverse.cpp**. Also add instructor-supplied source code file **C2A2E4_main-Driver.cpp**. Do not write a `main` function! `main` already exists in the instructor-supplied file and it will use the code you write.

File **C2A2E4_OpenFile.cpp** must contain a function named `OpenFile`.
`OpenFile` syntax:

```
void OpenFile(const char *fileName, ifstream &inFile);
```

Parameters:
`fileName` – a pointer to the name of a file to be opened
`inFile` – a reference to the `ifstream` object to be used to open the file
Synopsis:
Opens the file named in `fileName` in the read-only text mode using the `inFile` object. If the open fails an error message is output to `cerr` and the program is terminated with an error exit code. The error message must mention the name of the failing file.
Return:
void if the open succeeds; otherwise, the function does not return.

File **C2A2E4_Reverse.cpp** must contain a function named `Reverse`.
`Reverse` syntax:

```
int Reverse(ifstream &inFile, const int level);
```

Parameters:
`inFile` – a reference to an `ifstream` object representing a text file open in a readable text mode.
`level` – recursive level of this function call: 1 => 1st call, 2 => 2nd call, etc.
Synopsis:
Recursively reads one character at a time from the text file in `inFile` until a separator is read. Then any non-separator characters that were read are displayed in reverse order with the last and next to next to last characters capitalized. Finally, the separator is returned to the calling function. `Reverse` neither reverses nor prints separators but merely returns them. The code in the instructor-supplied driver file is responsible for printing each returned separator.
Definition of separator:
any whitespace (as defined by the standard library `isspace` function), a period, a question mark, an exclamation point, a comma, a colon, a semicolon, or the end of the file.
Return:
the current separator

The `Reverse` function must:

1. Implement a recursive solution and be able to display words of any length.
2. Be tested with instructor-supplied data file **TestFile2.txt**, which must be placed in the program's "working directory".
3. not declare more than two non-`const` variables other than its two parameters.
4. not use arrays, **static** objects, external objects, dynamic memory allocation, or the `peek` function.
5. not use anything from `<cstring>`, `<list>`, `<sstream>`, `<string>`, or `<vector>`.
6. not use any variable names or comments that specifically state a capitalization position (such as last, 3rd to last, next to next to last, etc .).

Example

If the text file contains:

```
What!  Another useless, stupid, and unnecessary program?
Yes;  What else?:  Try input redirection.  /[.]/  /}.!?,::=+#/
```

and `Reverse` is called using:

```
while ((thisSeparator = Reverse(inFile, 1)) != EOF)
     cout.put(thisSeparator)
```


the following is displayed: tAhW! rehtOnA sselEsU, dipUtS, DnA yrasseceNnU margOrP?
SeY; tAhW eSlE?: YrT tuPnI noitceriDeR. [/./] }/.!?,::/#+=

Submitting your solution

`Send an empty-body email to the assignment checker with the subject line **C2A2E4_177752_U09845800** and with your three source code files attached.

See the course document titled "How to Prepare and Submit Assignments" for additional exercise formatting, submission, and assignment checker requirements.

Hints and Recommendations:

1. See course book notes 12.7 and 12.8 for recursive examples.

2. Recursive functions should use the fewest variables necessary and should not use external or static variables.

3. When calling `Reverse` always pass the expression `level + 1` as its second argument.

4. Writing an **`inline`** function that determines if its parameter is a separator and returns type **`bool`** is cleaner than directly testing for separators in the `Reverse` function. Note the following:
   a. Whitespace is not just the *space* character itself but is every character defined as whitespace by the **`isspace`** function.
   b. The **`ispunct`** function is not suitable for this exercise because it detects more than just the punctuator characters used as separators.

5. Because the value of macro `EOF` is type **`int`**, the implementation-dependent negative value it represents cannot be reliably represented by type **`char`**. Casting a type **`char`** expression to type **`int`** does not eliminate the problem because the extra bits needed to represent `EOF` will have already been lost. Also, if a type **`int`** expression contains the value of `EOF`, casting it to type **`char`** results in a value that cannot represent `EOF`.

6. The step-by-step algorithm I recommend is shown below, although you are not required to use it. Regardless, remember that separators must never be displayed or reversed by the `Reverse` function but must merely be returned by it. The instructor-supplied "driver" file is responsible for displaying returned separators.

   Algorithm: Each time function `Reverse` is called it must do the following:
   A. Read the next input character and store it in a type **`int`** automatic variable named `thisChar`.
   B. IF the character in `thisChar` is a separator
      Return the character to the caller.
      ELSE
      1) Call `Reverse` and store its return value in a type **`int`** automatic variable named `thisSeparator`.
      2) IF the current recursive level is a capitalization level
         Display the character in `thisChar` as capitalized.
         ELSE
         Display the character in `thisChar` unaltered.
      3) Return the character in `thisSeparator` to the caller.