Assignment 2
C/C++ Programming II

General Information

Instructor-Supplied Source Code and Data Files

For all programming exercises in this course starting with this assignment the instructor has supplied one or more source code "driver" files. In order to produce a working program these files must be added to your projects along with any source code files you create as directed in each exercise. These driver files will always contain a `main` function that will use/test the code you wrote, so never write your own `main` function or you will get a "multiply defined" linker error when you try to build your program. When sending source code files to the assignment checker always send those from the instructor as well as any you wrote.

In addition to the instructor-supplied driver files several data files (`.txt`, `.bin`) have also been supplied. These will be needed for testing various exercises as indicated in those exercises and they must be copied into the appropriate "working directory" (see the next section) in order for the code in the instructor-supplied driver files to find them. These data files are for your own local testing purposes and must never be sent to the assignment checker.

Your Source Code Files

In addition to the source code files supplied by the instructor most exercises require you to create one or more new files to contain the various items called out in those requirements. Note, however, that you may also place anything else you desire in your new files (functions, macros, etc.) as long as it is appropriate and not forbidden by good programming practice or the exercise requirements.

Where Does a Program Look for Files When Attempting to Open Them?
Where Does a Program Create New Files?
Where Should You Put Instructor-Supplied Files?

If a program is trying to open an existing file or create a new file and a full path to that file is provided, that path is used. However, specifying a full path is not usually advisable since paths can differ on different systems. For this reason full paths are never acceptable in this course for files your programs need to open or create, but instead only the file name itself with no path must be used. In such cases programs typically assume their location to be in the current "working directory".

If you are executing your program from within an IDE the "working directory" will usually be the same directory where the various "project configuration" files were automatically created for you by the IDE when you initially created your project. If you don't know which directory that is you can find it by searching your computer for one of those files. For example, in Visual Studio the primary project configuration file has a `.vcxproj` extension whereas in the Code::Blocks IDE it has a `.cbp` extension. If you are instead executing your program outside an IDE the "working directory" will usually be the same directory that contains your executable file. On most systems this executable file will have the same name as one of your implementation files but with a `.exe` extension.

If an exercise calls for the use of an instructor-supplied source code file (`.c`, `.cpp`, `.h`, etc.), place it in the same directory where you create your own source code files. If an exercise calls for the use of an instructor-supplied data file (`.txt` or `.bin`) place it in the appropriate typical working directory and verify that your program can find it. If not you can resort to the brute force method for determining the working directory. In this method you simply write a program to create a file with a unique name and then search your computer to see where it was created.

© 1992-2015 Ray Mitchell

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 C language standard 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.

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.

© 1992-2015 Ray Mitchell

Exercise 1 *(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 is not a prototype – never actually prototype a macro)*:

```
int CountBitsM(objectOrType)
```

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 a function named `CountIntBitsF`.
`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`:

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

`CountBitsM`:

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

`CountIntBitsF`:

1. Shall not use any macro;
2. Shall not use anything from any header file;
3. Shall not be in a header file.
4. Shall not perform any multiplications or divisions;

If you get an Assignment Checker warning regarding instructor-supplied file **C2A2E1_main-Driver.c** the problem is actually in your macro.

Could the value produced by `CountBitsM` for type **int** be different than the value produced by `CountIntBitsF`? If so, why? If not, why not? Place the answers to these questions as comments in one of the "Title Blocks".

Submitting your solution

Send all three source code files to the Assignment Checker with the subject line **C2A2E1_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "Preparing and Submitting Your 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 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.

Exercise 2 *(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 a copy of the `CountIntBitsF` function you wrote for the previous exercise, modified if necessary for this exercise.

File **C2A2E2_Rotate.cpp** must contain a function named `Rotate`.
`Rotate` syntax:

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

Parameters:
`object` – the object to rotate
`count` – the number of bit positions & direction to rotate: negative=>left and positive=>right
Synopsis:
Rotates all bits in `object` by the number of bit positions and direction specified by `count`.
Return:
the value of the rotated object

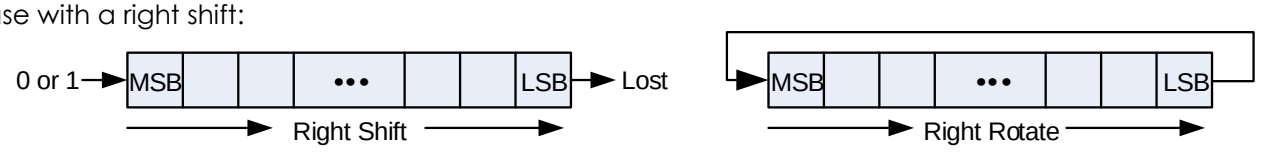
The `Rotate` function must:

1. call `CountIntBitsF` if the number of bits in type **unsigned** is needed;
2. not call `CountIntBitsF` more than once or in a loop;
3. not make any assumptions about the number of bits the data type of parameter *object*;
4. not make any assumptions about the number of bits in a **char**/byte (such as 8);
5. not use `CHAR_BIT` or **sizeof** or call any function or macro that does;
6. not display anything.

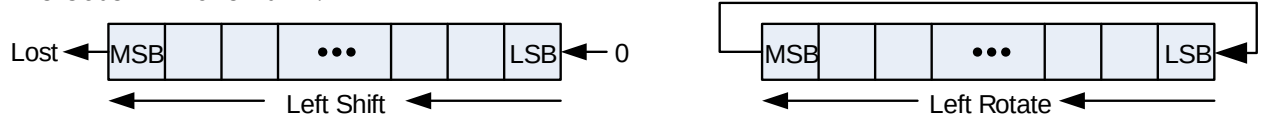
Here are some typical results:

Function Call	Return values for a 16-bit object	Return values for a 32-bit object
`Rotate(0xA701, 1)`	`0xD380`	`0x80005380`
`Rotate(0xA701, 256)`	`0xA701`	`0x0000A701`
`Rotate(0x000C, 2)`	`0x0003`	`0x00000003`
`Rotate(0x8000, -1)`	`0x0001`	`0x00010000`
`Rotate(0x3000, -2)`	`0xC000`	`0x0000C000`

Explanation
When a pattern is "shifted" each bit shifted off the end is simply lost. In "rotation", however, the end bits (the least significant bit LSB and the most significant bit MSB) are treated as if they are adjacent. That is, when a pattern is right-rotated the LSB is placed into the 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 all three source code files to the Assignment Checker with the subject line **C2A2E2_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

Hints:

To avoid needless loss of credit be sure you understand lines 14 & 15 of note 11.7 of the course book. Displaying bit patterns in decimal is meaningless; bit patterns must always be displayed in hexadecimal.

Exercise 3 *(6 points – Drawing only – No program required)*

Create a stack frame illustration using the format shown on the next page and place it in a PDF file named **C2A2E3_StackFrames.pdf**. Although using Excel, Word, Visio, etc. to create such an illustration and the required PDF file is easy, you may instead do it the hard way by drawing it by hand and scanning it in if you wish as long as it is neat and easily readable. Your illustration must start with the following "startup" stack frame:

Memory Addresses		Stack Values	Description	
Relative	Absolute			
BP+??	??	??	??	***startup* Stack Frame**
BP+??	FA9*h*	??	??	

The remaining stack frames must be based upon the data type sizes and code shown below. Do not show stack frames for library functions. This is a theoretical exercise only and should not be compared to any values obtained from actually running the program. Assume all appropriate headers and prototypes are present and that the "C calling convention" is being used:

Assume: type **int** is 3 bytes; type **long** is 4 bytes; all addresses (pointers) are 5 bytes.

```
int main(void)◄──────  "startup function" calls main
{                      and main returns to it.
    long val = Ready();

    printf("Return from main: val = %ld\n", val);
    return(EXIT_SUCCESS);
}

long Ready(void)
{
    long res = gcd(128L, 96L);

    printf("Return from Ready: res = %ld\n", res);
    return res;
}

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

Function `main`	
Operation	Instruction Address
assignment to `val`	AB4*h*

Function `Ready`	
Operation	Instruction Address
assignment to `res`	108*h*

Function `gcd`	
Operation	Instruction Address
the **return** on line 42	7C0*h*

Waypoints:
To help you determine if you might have a problem, note the following:

1. There will be 3 stack frames for the `gcd` function, each containing 5 items;
2. The absolute address of the final item in the final stack frame of your drawing will be **F44*h***.

Submitting your solution

Send your PDF file to the Assignment Checker with the subject line **C2A2E3_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

Hint:

The illustration below demonstrates the required general format for this exercise and was taken directly from Note 12.6C of the course book. Create only one diagram. It must represent the finished stack with all stack frames on it. Since the value of a return object is not known until a function returns, use question marks to represent the values of such objects.

Figure 5

Stack View -- Final State

Memory Addresses		Stack Values	Description	
Relative	Absolute			
BP+??	??	??	??	*startup* Stack Frame
BP+??	FA9*h*	??	??	
BP+6*h*	FA7*h*	??	Return Object (int)	
BP+3*h*	FA4*h*	??	Function Return Address	*main* Stack Frame
BP	FA1*h*	??	Previous Frame Address	
BP-2	F9F*h*	??	x	
BP+3	F9C*h*	200*h*	Function Return Address	*Ready* Stack Frame
BP	F99*h*	FA1*h*	Previous Frame Address	
BP+6*h*	F97*h*	397	value	
BP+3*h*	F94*h*	116*h*	Function Return Address	*Recur* Stack Frame 1
BP	F91*h*	F99*h*	Previous Frame Address	
BP+6*h*	F8F*h*	39	value	
BP+3*h*	F8C*h*	7BE*h*	Function Return Address	*Recur* Stack Frame 2
BP	F89*h*	F91*h*	Previous Frame Address	
BP+6*h*	F87*h*	3	value	
BP+3*h*	F84*h*	7BE*h*	Function Return Address	*Recur* Stack Frame 3
BP	F81*h*	F89*h*	Previous Frame Address	

BP F81*h*
SP F81*h*

Exercise 4 *(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.
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 encountered. Those non-separator characters are then displayed in reverse order, with the last character displayed being capitalized. Finally, the separator is returned to the calling function. Separators are not reversed and are not printed by `Reverse`, but are instead merely returned. The code in the instructor-supplied driver file is responsible for printing the separators.
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 EOF
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**. If you have placed it in the appropriate directory the instructor-supplied source code file will use it automatically.
3. not declare more than two variables other than the function's 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>`.

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 the three source code files to the Assignment Checker with the subject line **C2A2E4_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

Hints:

See course book notes 12.7 and 12.8. Write an inline function to check if a character is a separator, noting that whitespace is not just the *space* character itself but is every character defined as whitespace by the `isspace` function. Recursive functions should have as few automatic variables as is practical, but should also not use any external or static variables. As each recursive level of function `Reverse` is entered a new character is read and stored in a local variable I'll call `thisChar`. If the character is a separator it is then returned to the caller. If it is not a separator the `Reverse` function is called again and its return value is stored in a variable I'll call `thisSeparator`. After each return the character in `thisChar` is displayed and, if at recursive level 1, is also capitalized. Variable `thisSeparator` is then returned to the caller. Separators are never printed by `Reverse` but are instead merely returned by it. My driver is responsible for printing the separators returned to it by `Reverse`.

Get a Consolidated Assignment 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 email to the assignment checker with the subject line **C2A2_*ID***, where ***ID*** is your 9-character UCSD student ID. 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.

© 1992-2015 Ray Mitchell