CMSC 216 Project #1 Fall 2012

C Function Puzzles **Due: Tuesday Sep 25, 2012, 6:00PM**

This project is designed to give you practice with some of the operations that are different from what you've used in Java. You will be implementing several functions that have, in many cases, very simple solutions. However, you will not be allowed to implement the simple solution – instead, your assignment is to implement these functions using only certain operators and language constructs, the set of which will vary from function to function. Your goal is to implement the given functions with as few operators as possible, which will require a solid knowledge of how to use the operators that you are allowed to use.

1 Procedure

1.1 Obtain the project files

We have provided a header file, `puzzles.h`, which you will need to include in your code. You can obtain this file by logging onto one of the Grace machines and executing the following command.

```
tar -C ~/216 -xzvf ~/216public/project1/project1.tgz
```

The tarball contains both the header file and a `.submit` file that you will need to submit your project. Extracting the files with the `tar` command given above will also place these files in a new directory, `~/216/project1`, where you should do your work for this project.

1.2 Implement the functions

After extracting the project files, `cd` into the `project1` subdirectory and create a `puzzles.c` file. This file will contain the implementations of the functions you are being asked to write, and should `#include` the `puzzles.h` file. Descriptions of the various functions, and the restrictions you must obey for each one, are provided below.

1.3 Test your implementations

As for all projects in this class, the small suite of public tests we release will not be comprehensive, and you will need to do testing on your own to ensure the correctness of your functions. The results of the secret tests for this project, and for subsequent projects, will form the majority of your grade.

Although we have not formally covered testing yet in the course, you should test your functions using a driver program. For example, say you are asked to write a function `bit_or(a, b)` that will return the result of the bitwise OR operation on `a` or `b`, but without using the `|` operator. In this case, you can write a function – for testing purposes only – that uses the `|` operator, and check your `bit_or()` function's return value against the return value of this testing-only function for various inputs. An example program, `tester.c`, is shown below.

```
#include <stdio.h>
#include "puzzles.h"

static int correct_bit_or(int a, int b);
static void test_case(int a, int b);

int main() {
  test_case(0, 0);
  test_case(1, 2);
  test_case(0x29, 0x983);
  return 0;
}
```

```
static int correct_bit_or(int a, int b) {
  return a | b;
}

static void test_case(int a, int b) {
  int correct_result, my_result;

  printf("Testing bit_or(%d, %d): ", a, b);
  correct_result = correct_bit_or(a, b);
  my_result = bit_or(a, b);
  if (my_result == correct_result) {
    printf("PASSED!\n");
  }
  else {
    printf("FAILED.  (Got %d, should be %d.)\n", my_result, correct_result);
  }
}
```

To compile this into a program (assuming you have written the `bit_or()` function into file `puzzles.c`), you would use the following command:

```
gcc -g -Wall -O0 -Werror -Wshadow -Wwrite-strings -o tester tester.c puzzles.c
```

Then running the `tester` program will print out something like this:

```
Testing bit_or(0, 0): PASSED!
Testing bit_or(1, 2): PASSED!
Testing bit_or(41, 2435): FAILED.  (Got 2499, should be 2475.)
```

2 Specifications

The prototypes for the functions you will implement are listed both here and are also in the `puzzles.h` file. For each function, a list of allowed operators, constants, and C language constructs is given. When writing your functions, you may only use the functions' formal parameters along with the list of allowed operators, constants, and constructs. Specifically, you **may not** assign to variables (i.e., use the = operator or any compound assignment operator like +=), call other functions, or use `if`, `switch`, `break`, `continue`, or loop statements **unless** the function specification explicitly allows one or more of those constructs. In all functions, you **may** create variables, use the `return` keyword, and use parentheses; if assignment is allowed, you may also use any compound assignment operator if the corresponding operator is allowed (e.g., if `&` and = are allowed, `&=` is allowed). Note that `!=` is **NOT** a compound assignment operator, but an equality test operator.

2.1 The functions

1. `int bit_or(int a, int b)`:
   Allowed operators: `+ & ~ ^ << ! >>`

   This function should return the bitwise OR of the `a` and `b` parameters (`a | b`).

2. `int is_nonzero(int x)`:
   Allowed operators: `+ & | ~ ^ << ! >>`

   Returns 1 if `x` is not zero, 0 otherwise.

3. `unsigned int times7(unsigned int x)`:
   Allowed operators: `+ & ~ | ^ << ! >>`
   Allowed constants: `1 2 4 8 16`

   Returns 7 times `x`. You may assume that the value of `7 * x` can be stored in an `unsigned int` without any overflow.

4. `int floor_log8(unsigned int x):`
   Allowed operators: `++ -- = & | ~ ^ << ! >>`
   Allowed constants: `1 2 4 8 16`
   Allowed constructs: `for` loops

   Returns the floor of the logarithm base eight of `x` (i.e., $\lfloor \log_8 x \rfloor$). If `x` is 0, the function should return -1.

5. `size_t sizeof_long():`
   Allowed operators: `++ -- = & | ~ ^ << ! >>`
   Allowed constants: `1 2 4 8 16`
   Allowed constructs: `for` loops

   Returns the size of a `long`, in bytes, without using the `sizeof` operator. The return value should be equivalent to the expression `sizeof(long)` . Note that your function must be written to work on any system, so you may not make any assumptions about the size of a `long` based on what you know of the Grace systems.

   (Note: `size_t` is a type required by the C Standard that is used by many C library functions. It is defined by a system as an unsigned integer type capable of representing the size of the largest possible object in the computer.)

6. `unsigned int reverse_bytes(unsigned int x):`
   Allowed operators: `+ - = & | ~ ^ << ! >>`
   Allowed constants: `1 2 4 8 16`

   Reverse the bytes of `x` (you may assume that `int`s are 4 bytes in size for this function). For example, `reverse_bytes(0x12345678) = 0x78563412`.

7. `unsigned int hex_c0c0c0c0():`
   Allowed operators: `+ - = & | ~ << ! >>`
   Allowed constants: `1 2 4 8 16`

   Return `0xc0c0c0c0`.

8. `int pop_count(unsigned int x):`
   Allowed operators: `++ -- = & ~ | ^ << ! >>`
   Allowed constants: `1 2 4 8 16`
   Allowed constructs: `while` loops

   Return the population count (the number of bits set to 1) of `x`. For example, `pop_count(0x05000001) = 3`.

9. `unsigned int get_byte(unsigned int x, int n):`
   Allowed operators: `+ - = & | ~ ^ << ! >>`
   Allowed constants: `1 2 4 8 16`

   Return the `n`th-most significant byte of `x` (with `n == 0` denoting the most significant byte). For example, `get_byte(0x0a0b0c0d, 2) = 0x0c`. You may assume `n` is one of $\{0, 1, 2, 3\}$.

10. `int equal(int a, int b):`
    Allowed operators: `+ & | ~ ^ << ! >>`
    Allowed constants: `1 2 4 8 16`

    Return 1 if `a == b`, 0 otherwise.

11. `int is_address_in_prefix(uint32_t address, uint32_t prefix, uint32_t prefix_len):`
    Allowed operators: `<< - & == =`
    Allowed constants: `32`

    An IP address prefix is a 32 bit number with some fixed number of significant bits at the beginning. 0/0 is the prefix that includes all addresses because no bits are significant. 128.8.128.8/32 is a prefix that includes only one address because all bits are significant. (128.8.128.8 is another way of writing

the integer 0x80088008). 128.8.0.0/16 is a prefix that includes 32,000 addresses, all those that start with 0x8008.

Return 1 if the provided address is in the specified prefix and 0 otherwise.

12. `int index_of_first_bit_different(uint32_t address1, uint32_t address2):`
Allowed operators: `^ < && & >> == ++ ?: + =`
Allowed constants: `-1 0 32`
Allowed constructs: `for` loops

Knowing which bit is different between two addresses helps decide whether they're in the same prefix.

Return 1 if the first bit is different and -1 if no bits are different.

3 Important Points

1. Your functions must work with all possible inputs (except those that violate the assumptions you are explicitly told you may make).
2. The list of allowed constants, operators and C constructs changes from function to function. Be sure you are only using items from a given function's list when implementing that function.
3. Even though you may have a working implementation, you should keep examining it closely to try to use as few operations as possible to obtain the best possible project grade.

4 Grading Criteria

Your project grade will be determined with the following weights:

Results of public tests	15%
Results of secret tests	60%
Code style grading	15%
Solution quality	10%

The public tests will be made available shortly after the project is released. You can find out your results on these tests by checking the submit server a few minutes after submitting your project. Secret tests, and their results, will not be released until after the project's late deadline has passed.

4.1 Public tests

The public tests are C programs, much in the same style as the `tester` program described above. They will be compiled and run in a similar manner on the submit server, and you can also compile and run them yourself. Along with the C programs, corresponding expected output will be provided: for example, a public test program named `public00.c` will have an output file named `public00.output`. To verify your code passes a public test, you can use a command similar to the following:

```
./public00 | diff - public00.output
```

This will send the output of the `public00` command into the `diff` command, which will then check the output of `public00` against the file `public00.output` – if there are any differences, they will be printed out. If no differences exist, there will be no output, and you have passed the test.

4.2 Style grading

For this project, your code is expected to conform to the following style guidelines:

- Your code must have a comment at the beginning with your name, university ID number, and UMD Directory ID (i.e., your username on Grace).

- No lines longer than 80 columns are allowed (tab stops count as 8 spaces for the purposes of this rule). You can check your code's line lengths using the `linecheck` program in the `~/216public/bin` directory, which should be in your path. Just run "`linecheck filename.c`" and it will report any lines that are too long.
- Use reasonable and consistent indentation. If you like, you may use the `indent` command to help format your code, but if you do use `indent` we request that you use it with the `-kr` switch – the default indentation style is somewhat less than beautiful.
- Use descriptive and meaningful variable names, using the naming convention described in lecture.
- Do not use global variables.
- You will not need to use symbolic constants for this project, as you are restricted to a small set of available constants.
- As function calls are not an allowed construct for any of the puzzles, you may not use helper functions for this project.
- Each function must have, at a minimum, a comment describing its purpose and operation. If you use a complicated algorithm to implement a function, you definitely need an extra comment explaining the complicated steps of your algorithm.
- Use whitespace appropriately, especially horizontal whitespace between operators and operands. As this project naturally lends itself to dense code, it is **imperative** that you keep your code readable by not writing code like this:

  ```
  f=(x*x)|(y*z)&&(n^(n|x))==y+y+y;
  ```

  but rather, code like one of these examples:

  ```
  f = (x * x) | (y * z) && (n ^ (n | x)) == y + y + y;
  f = (x*x) | (y*z) && (n ^ (n|x)) == y + y + y;
  f = (x << 1)       /* 2x */
      + (
          (!! y)     /* forces y into {0, 1} */
          ^ (!! z)   /* same for z */
        );
  ```

  Blatant disregard of this requirement will be penalized harshly.
- Code must compile with the `-g -Wall -O0 -Werror -Wshadow -Wwrite-strings` flags passed to `gcc`.

4.3 Solution quality

When writing your solutions, you should try to make your functions as short and simple as possible while preserving clarity. Thus, for this project, we will be checking your solutions to see how many operations you have used in each function – the lower the number of operations, the higher your quality score will be.

For the purposes of calculating the number of operations, all of the following count as one operation each time they are used in a function:

- Any operator that is not a compound assignment operator (e.g., `+=`, `<<=`).
- The guard of any conditional or loop statement (because the program must check the truth of the expression).
- Returning from a function.

Compound assignment operators count as two operations. Statements that are part of a loop that are executed multiple times only count once toward the operation count.

To help clarify this, here are some examples:

```
int example1(int x) {
  int i, j = 0;
  for (i = 0; i < x; i++)
    j++;
  return i + j;
}
```

```
int example2(int x) {
  int i;
  i = 0;
  while (x--)
    i++;
  return i << 1;
}
```

```
int example3(int x) {
  return 2 * x;
}
```

`example1()` is considered to have 8 operations (2 assignments, 1 comparison, 1 loop guard, 2 increments, 1 addition, and 1 return). `example2()` is considered to have 6 operations (1 assignment, 1 decrement, 1 loop guard, 1 increment, 1 left shift, and 1 return). `example3()` is considered to have 2 instructions (1 multiplication and 1 return), and would receive the highest quality grade of these example functions.

We will also check during style grading to ensure that your solutions do not violate the rules described in Section 2. If we find solutions that use disallowed operators, constants, or constructs, we will make note of it and deduct points from your public and/or secret tests results to compensate for the use of disallowed functionality. If you have any questions at all about whether something is or is not allowed for a given function, please talk to a member of the instructional staff so that we can clarify the matter for you and for the rest of the class.

5 Submission

5.1 Deliverables

For this project, the only file that we will grade is `puzzles.c` (which **must** be the name of your source file). We will compile our tests with the version of `puzzles.h` that we supplied, so do not make any changes to that file.

5.2 Procedure

Before you submit your project, you should first check to make sure you have passed all the public tests for the project. Refer to Section 4.1 for details on how to test for yourself that you have passed a test. If the public tests have not yet been posted, you cannot be sure you have passed the public tests, and so we do not recommend submitting.

Once you believe you have passed the public tests, you can submit your project by executing, in your project directory, the command:

```
submit
```

This will prompt you for your UMD Directory ID and password, and if all goes well, inform you of a successful submission. You should then log onto the submit server (there is a link on the course website) and check your public test results to be sure that things worked as you expected.

6 Other Notes

6.1 Academic Integrity

As mentioned in the syllabus, any evidence of cheating will be referred to the Student Honor Council and may result in a grade of XF in this course. Submissions will be checked with an automated source code comparison tool to look for evidence of cheating. This tool has already proven itself to be very effective, so we advise you to remember that the course syllabus (§9) forbids cooperation between students on projects, including either copying **or sharing** source code. You are also further advised to refrain from copying code from external resources. Your projects are to be **your own work**, and **only your work**.

Your instructors are quite serious about this, and will not hesitate to refer cases to the Honor Council should we have sufficient evidence to do so.

6.2 Deadlines

Submission deadlines are strictly enforced by the submit server, and we will not extend them for things such as network outages. Extensions may be given on a case-by-case basis, but will likely only be granted in emergency cases. Therefore, you should start work on this project early, as last-minute technical problems are not an excuse for missing either the on-time or late deadline.