# CS 246 Winter 2016 – Assignment 2
# Instructors: Peter Buhr and Rob Schluntz
# Due Date: Monday, February 22, 2016 at 22:00

February 7, 2016

This assignment introduces basic C++ programming, control flow, internal representation, and how to use a Makefile. Use it to become familiar with these facilities, and ensure you use the specified concepts in your assignment solution, **i.e., writing a C-style solution for questions is unacceptable, and will receive little or no marks**. (You may freely use the code from these example programs.)

Use the regress script from assignment 1 to compare output from given programs with your programs to ensure identical behaviour.

1. Given the C++ program in Figure 1, compile the program with and without preprocessor variable RECURSION defined.

   ```
   $ g++ -DRECURSION recursion.cc
   $ g++ recursion.cc
   ```

   Compare the two versions of the program with respect to performance by doing the following for each version:

   - Run the program and time the execution using the time command:

     ```
     $ /usr/bin/time -f "%Uu %Ss %E" ./a.out
     3.21u 0.02s 0:03.32
     ```

     (Output from time differs depending on the shell, so use the system time command.) Compare the *user* time (3.21u) only, which is the CPU time consumed solely by the execution of user code (versus system and real time).

     If the program segment faults (signal 11), increase the stack-size limit using a shell command. For sh shells use:

     ```
     $ ulimit -S -s unlimited
     ```

     for csh shells use

     ```
     % limit stacksize unlimited
     ```

   - Use the program command-line argument (if necessary) to adjust the number of times the experiment is performed to get user times approximately in the range 0.1 to 100 seconds. (Timing results below 0.1 seconds are inaccurate.) Use the same command-line value for all experiments.
   - Run both the experiments again after recompiling the programs with compiler optimization turned on (i.e., compiler flag -O2).

     ```
     $ g++ -O2 -DRECURSION recursion.cc
     $ g++ -O2 recursion.cc
     ```

   - Include 4 timing results to validate the experiments.
   - Explain the relative differences in the timing results with respect to looping versus recursion for iteration.
   - State the performance difference when compiler optimization is used.

2. (a) Transform routine do_work in Figure 2, p. 3 so it preserves the same control flow but removes the **for** and **goto** statements, and replaces them with **ONLY** expressions (including & and |), **if**/**else** and **while** statements. The statements **switch**, **for**, **do**, **break**, **continue**, **goto**, **throw** or **return**, and the operators &&, || or ? are not allowed. In addition, setting a loop index to its maximum value, to force the loop to stop is not allowed. Finally, copying significant amounts of code or creating subroutines is not allowed, i.e., no transformation where the code grows exponentially with the number of nested loops. **Zero marks**

```
#include <cstdlib>                     // atoi

void iterate( int i ) {
    volatile int bef = 0, aft = 0;     // prevent loop/recursion elimination
#if ! defined( RECURSION )
    for ( ;; ) {
        bef += 1;
      if ( i == 0 ) break;
        i -= 1;
        aft += 1;
    } // for
#else
    bef += 1;
  if ( i == 0 ) return;
    iterate( i - 1 );
    aft += 1;                          // prevent tail recursion optimization
#endif
} // iterate

int main( int argc, char *argv[] ) {
    unsigned int times = 10000000;
    switch ( argc ) {
      case 2:
        times = atoi( argv[1] );
    } // switch

    iterate( times );
} // main
```

Figure 1: Looping versus Recursion

**will be given to a transformation violating any of these restrictions.** New variables may be created to accomplish the transformation. Output from the transformed program must be identical to the original program.

(b) Compare the original and transformed program with respect to performance by doing the following:

- Remove (comment out) *all* the print (cout) statements in the original and transformed version.
- Time the execution using the time command:

  ```
  $ /usr/bin/time -f "%Uu %Ss %E" ./a.out
  3.21u 0.02s 0:03.32
  ```

  (Output from time differs depending on the shell, so use the system time command.) Compare the *user* time (3.21u) only, which is the CPU time consumed solely by the execution of user code (versus system and real time).
- Use the program command-line argument (if necessary) to adjust the number of times the experiment is performed to get user times in the range 5 to 20 seconds. (Timing results below 1 second are inaccurate.) Use the same command-line value for all experiments.
- Run both the experiments again after recompiling the programs with compiler optimization turned on (i.e., compiler flag -O2).
- Include 4 timing results to validate the experiments.
- Explain the relative differences in the timing results with respect to the original and transformed program.
- State the performance difference when compiler optimization is used?

```
#include <cstdlib>                      // atoi
#include <iostream>
using namespace std;

// volatile prevents dead-code removal
void do_work( int C1, int C2, int C3, int L1, int L2, volatile int L3 ) {
    for ( int i = 0; i < L1; i += 1 ) {
        cout << "S1 i:" << i << endl;
        for ( int j = 0; j < L2; j += 1 ) {
            cout << "S2 i:" << i << " j:" << j << endl;
            for ( int k = 0; k < L3; k += 1 ) {
                cout << "S3 i:" << i << " j:" << j << " k:" << k << " : ";
      if ( C1 ) goto EXIT1;
                cout << "S4 i:" << i << " j:" << j << " k:" << k << " : ";
          if ( C2 ) goto EXIT2;
                cout << "S5 i:" << i << " j:" << j << " k:" << k << " : ";
              if ( C3 ) goto EXIT3;
                cout << "S6 i:" << i << " j:" << j << " k:" << k << " : ";
            } // for
          EXIT3:;
            cout << "S7 i:" << i << " j:" << j << endl;
        } // for
      EXIT2:;
        cout << "S8 i:" << i << endl;
    } // for
  EXIT1:;
} // do_work

int main( int argc, char *argv[] ) {
    int times = 1, L1 = 10, L2 = 10, L3 = 10;
    switch ( argc ) {
      case 5:
        L3 = atoi( argv[4] );
        L2 = atoi( argv[3] );
        L1 = atoi( argv[2] );
        times = atoi( argv[1] );
        break;
      default:
        cerr << "Usage: " << argv[0] << " times L1 L2 L3" << endl;
        exit( EXIT_FAILURE );
    } // switch

    for ( int i = 0; i < times; i += 1 ) {
        for ( int C1 = 0; C1 < 2; C1 += 1 ) {
            for ( int C2 = 0; C2 < 2; C2 += 1 ) {
                for ( int C3 = 0; C3 < 2; C3 += 1 ) {
                    do_work( C1, C2, C3, L1, L2, L3 );
                    cout << endl;
                } // for
            } // for
        } // for
    } // for
} // main
```

Figure 2: Static Multi-level Exit

3. Write a program that reads a portable pixmap image (PPM) from standard input, applies image transformations, and writes the transformed image to standard output. PPM is a simple image encoding format. To simplify the question, only one of the 7 formats is supported: P6. A PPM file starts with 4 header values separated by whitespace: P6, *width*, *height*, *colour maximum*, where *width* and *height* are numbers representing the size of the image, and *colour maximum* is the maximum size of a colour value. After the header, and separated from it by a single whitespace character, are *width* $\times$ *height* pixels, where a pixel is represented by 3 values (red, green, and blue (RGB)) for that spot in the image, e.g.:

```
P6 3 4 255
0 0 0         128 128 128     255 255 255
255 0 0       255 0 0         255 0 0
0 255 0       0 255 0         0 255 0
0 0 255       0 0 255         0 0 255
```

where the header values are ASCII text and the pixels are internal 1-byte values for each colour (meaning the maximum colour value is 255). For example, the actual image file printed as bytes in decimal is:

```
$ od –width=12 -t u1 image
0000000   80   54  32   51   32   32   52   32   50   53   53  10
0000014    0    0   0  128  128  128  255  255  255  255    0   0
0000030  255    0   0  255    0    0    0  255    0    0  255   0
0000044    0  255   0    0    0  255  255    0    0  255    0   0
```

where the first line is the ASCII characters (printed in decimal), and the remaining lines are bytes representing the colour values (printed in decimal). Hence, there is a combination of formatted and unformatted I/O to read the image. Use formatted I/O to read/write the header and unformatted I/O to read/write each row of pixels, i.e., a single read/write per row. Use exceptions to handle all file errors and end-of-file.

Store the PPM image in a structure with the pixels in a matrix:

```
struct PpmImage {
    struct Pixel {
        unsigned char r, g, b;                  // value for red, green, blue
    };
    unsigned int width, height, colourMax;    // image configuration
    Pixel **pixels;                             // matrix of pixels
};
```

Create the following image transformations:

**vertical flip:** flip the image from top to bottom by interchanging the rows of the pixel matrix around the center of the image.

**transpose:** transpose the image along its major diagonal.

**grey-scale:** remove the colour from the image by setting each of the 3 colours for a pixel to the average of the 3 colours.

All transformation routines have the following interface:

```
void transformation( PpmImage & );
```

**You may NOT change, add, or remove fields, parameters or returns for any of the GIVEN code.**

The shell interface to the ppm program is as follows:

```
ppm [ -v | -t | -g ]* < infile > outfile
```

(Square brackets indicate optional command line parameters.) The flag -v means a vertical transform; the flag -t means a transpose transform; the flag -g means a grey-scale transform. There can be 0 or more flags, which may appear in any order and any number of times. If there are no flags, the image remains unchanged and is copied from input to output. Assume only one image in a PPM file. Do not assume the image is well formed, i.e., the image file may contain errors; terminate the program with exit code 1 when an error is found in the image file.

There are many command to view a PPM file, such as the UNIX command `display file.ppm`. To view a PPM image using a command on the undergrad environment requires X11 forwarding.

4. Write a C++ program include that performs a simplified version of the file inclusion aspect of the C/C++ pre-processor. The include program transforms include directives found in an input file into the contents of the file specified in the directive. The include program does *not* change the contents of an input file, except for replacing the include directives by their specified file content. Only include directives of the form:

```
#include "filename"
```

are considered. This form searches for a file in the directory containing the current file from the last include location for a relative file pathname, and at a specific file for an absolute pathname.

```
#include "/usr/include/stdio.h" // absolute pathname (starts with '/')
#include "myfile.h"            // relative pathname (search current directory)
#include "data/myfile.h"       // nested relative-pathname (search directory “data”)
```

For simplicity, assume an include directive appears at the beginning of a line with no space or tab between the # and include. The file name must be preceded by at least one space or tab character, be enclosed in double quotes ("), and not followed by any other characters. Assume an include directive has no syntax errors, so it is unnecessary to check for any errors during parsing of the directive. However, you must check for runtime errors, such as an including a file that does not exist or cannot be read. **Do not implement the other form of file inclusion with file names in chevrons <>.**

Includes may be nested; that is, included files may include other files, which may in turn include other files. Moreover, the same file can be included several times. However, recursive includes are not allowed. For example, if file1 includes file2, and file2 in turn includes file1, it is an error, since attempting to process such an include directive produces an unbounded amount of output.

For nested relative-pathnames, there is the notion of a current directory that must be maintained and concatenated onto file names with respect to the location in the file structure where compilation is occurring. For example, if compilation starts in a particular directory with file start.cc containing:

```
#include "data/myfile.h"       // nested relative-pathname (search directory “data”)
```

then the current directory becomes `"data/"`. Now if file `"data/myfile.h"` contains **#include** `"y.h"`, then the file that is accessed is `"data/y.h"` not `"y.h"` in the directory where the compilation started. If file `"data/myfile.h"` contains **#include** `"data/y.h"`, then the current directory becomes `"data/data/"`, and if the file `"data/data/y.h"` contains **#include** `"z.h"`, the file accessed is `"data/data/z.h"` not `"z.h"` in the directory where the compilation started.

For an absolute pathname, the current directory is set to the directory path of the file name and then processing continues as for a relative pathname. For example, if start.cc contains:

```
#include "/u/jfdoe/cs246/a3/data/myfile.h"
```

The current directory becomes /u/jfdoe/cs246/a3/data and all relative pathnames start from this point in the file structure.

Write a recursive routine with the following prototype:

```
void processOneFile( istream &in, ostream &out, Node *flist, string prefixDir );
```

that does all the include processing. **Routine processOneFile may not dynamically allocate any storage, i.e., it may not use new/malloc or call any other routine *you write* that uses dynamic allocation. Dynamic allocation in only allowed to handle command-line arguments.** To detect recursive include directives, keep a stack of the file names being read. Use the following data structure for the stack elements:

```
struct Node {
    string fileName;
    Node *link;
};
```

and chain the elements into a stack using parameter flist. **You may NOT use any *container* data-structures from the C++ standard library.** When an include directive is encountered, check to see if the file name is already in the stack; if it is, a recursive include is being attempted. In this case, issue an error message and terminate the program with a return code of -1 (see man exit). Otherwise, push the file name on the stack, open it, and begin processing it.

Because a file can be named in multiple ways, e.g., file.cc, ./file.cc, ../jfdoe/file.cc, /home/jfdoe/file.cc, it is difficult to determine if a file is being included recursively solely by name. Therefore, convert all relative-pathnames to absolute because absolute pathnames are unique. The C library-routine realpath (see man -s 3 realpath) converts a relative file-name to an absolute file-name. Note, realpath works with C-style strings not C++ strings, so create an output character array of size FILENAME_MAX (constant defined in include file cstdlib) to hold the largest possible absolute file-name.

When the end of an included file is reached, pop the file name from the stack, close the file, and resume processing the file that included it. To handle the current directory for nested relative-pathnames, use the parameter prefixDir to pass the current directory to the next level of inclusion. If the next level has a nested relative-pathname, add its directory portion to the current directory to form the new current directory.

The program must be written to handle any depth of nested includes. However, in UNIX there is a limit on the number of files that can be open at one time, which has two consequences. First, it is important to close files when done with them. Second, if include files are nested too deeply, it is impossible to open any more files. Treat this case exactly like the case where a file does not exist or is not readable/writable. This means that it is sufficient to simply attempt to open a file; if the open fails for any reason, issue an error message and terminate the program with a return code of -1.

**You may NOT change, add, or remove fields, parameters or returns for any of the GIVEN code.**

The shell interface to the include program is as follows:

```
include [ infile [ outfile ] ]
```

(Square brackets indicate optional command line parameters.) If the output file is missing, direct output to cout. If the input and output file are both missing, input from cin and output to cout. The program must handle lines of arbitrary length in the input files using type string. The only error checking is for a file that cannot be opened and recursive include directives. Use exceptions to handle all file errors and end-of-file. No other error checking is to appear in the program. (Too much error checking makes assignment programs difficult to mark and does not increase what you learn but uses up your valuable time.)

Verify the correctness of your program by comparing the output with the C/C++ preprocessor:

```
$ include xxx.cc            # both generate the same output
$ cpp -P -C -traditional-cpp xxx.cc
```

Both commands generate the same output, modulo restrictions that apply for program include and handling of recursive includes.

## Submission Guidelines

Please follow these guidelines carefully. **Review the Assignment Guidelines and C++ Coding Guidelines *before* starting each assignment. Each text file, i.e., *.*txt file, must be ASCII text and not exceed 500 lines in length, where a line is a maximum of 120 characters.** Name your submitted files as follows:

1. recursion.txt – contains the information required by question 1, p. 1.
2. nostaticexits.{cc,C,cpp} – code for question 2a, p. 1. **Output for this question is checked via a marking program, so it must match exactly with the given program.**
3. nostaticexits.txt – contains the information required by question 2b, p. 2.
4. ppm.{cc,C,cpp} – code for question 3, p. 4. **Program documentation must be present in your submitted code. Output for this question is checked via a marking program, so it must match exactly with the given program.**

5. include.{cc,C,cpp} – code for question 4, p. 5. **Program documentation must be present in your submitted code. Output for this question is checked via a marking program, so it must match exactly with the given program.**
6. include.testtxt – test documentation for question 4, p. 5, which includes the input and output of your tests. Write a brief description for each test explaining what aspects of the program it is testing and how you decided if the program passed the test.

Use the following Makefile to compile the programs for questions 3, p. 4 and 4, p. 5 (**do not submit this file**):

```
CXX = g++-4.9                                # compiler
CXXFLAGS = -g -Wall -Werror -std=c++11 -MMD # compiler flags
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}} # makefile name

OBJECTS1 = ppm.o                             # object files forming executable
EXEC1 = ppm                                  # executable name

OBJECTS2 = include.o                         # object files forming executable
EXEC2 = include                              # executable name

OBJECTS = ${OBJECTS1} ${OBJECTS2}
EXECS = ${EXEC1} ${EXEC2}
DEPENDS = ${OBJECTS:.o=.d}                   # substitute ".o" with ".d"

.PHONY : all clean

all : ${EXECS}

${EXEC1} : ${OBJECTS1}                       # link step
	${CXX} $^ -o $@

${EXEC2} : ${OBJECTS2}                       # link step
	${CXX} $^ -o $@

${OBJECTS} : ${MAKEFILE_NAME}                # OPTIONAL : changes to this file => recompile

-include ${DEPENDS}                          # include *.d files containing program dependences

clean :                                      # remove files that can be regenerated
	rm -f ${DEPENDS} ${OBJECTS} ${EXECS}
```

Put this Makefile in the directory with your programs, name your source files ppm.{cc,C,cpp} and include.{cc,C,cpp}, and then execute shell command make ppm or make include in the directory to compile a program (make without an argument compiles both programs). This Makefile is used by Marmoset to build programs, so make sure your programs compiles with it. *Do not make any changes to the* Makefile.

**Follow these guidelines. Your grade depends on it!**