Programming Requirements

Ming-Hwa Wang, Ph.D.
Department of Computer Engineering
Santa Clara University

General Requirements

- Programs should be coded in the language required by the class. Run on school's Linux workstations using your own account.
- Submit the electronic version of source code by using Submit. Programs **must** pass Autotest (except cross platform client/server programs) before submitting. Due to limited disk space, only the last submitted program will be saved and used for grading. You should not submit too often. The order of emails arriving may not the same as your sending order. If you send too often without enough time in between - roughly 15 minutes - the last one received may not your correct version. In this case, it is your responsibility if you get the wrong grade.
- Your program should **input from stdin** (i.e., keyboard), and **output to stdout** (i.e., monitor) except requested explicitly. You need to print out all input and output to demonstrate the requested functionality in the program specifications. You can test your program and input either by typing from keyboard manually (and type Ctrl-D for end of input) or redirected (by "<") from a test file (as Autotest does it for you automatically.) Be careful about the EOF handling to make manual testing exact the same as Autotest.
- Your program should follow the **whitespace free format convention**.
- Sample input data are under /home/mwang2/test/coen<courseDigits>. You are responsible to construct your own test data according to the requirements specified for each programming assignment. Your test data should cover all the possibilities because your program will be tested using the test data prepared by grader after due date. You can't put any limitation on the input size/length (using realloc if necessary or use dynamic data structures) except specified explicitly.
- Programming is not typing. You need to debug your program and give correct result to get good grade. Typing only worth 30% of the score.

Whitespace Free Format Convention

Whitespaces include spaces, tabs, comments, and new lines. When you writing programs in any modern language, they all follow this convention, e.g., you can write the code either like:

```
if (b==0) {
   cout << b << endl;
}
```

Or equivalently you can simply write like:

```
if ( b == 0 ) { cout << b << endl; }
```

Re-grading Policy

If you have any doubt regarding the grading of program assignments. You should do the following:

1. Try re-run your program with the real test files the grader provided:
   `$ Autotest <num> -t /home/<grader>/test`
   where <num> is the program assignment number.
2. Only if your program can generate correct results for grader's test cases, you can ask regarding. If you need manual help to run your program (e.g., use special format input), then you need to pay 5 points for special service. But if the grader made mistake, you can get your points back without service charge.
3. You have to ask re-grading in 1 week after you get your grade of the program. After 1 week, the grader needs to clean up his limited disk space for new program assignments.

Protection

- protect your source code been copied by others:
  `$ cd; cd ..`
  `$ chmod go-rwx <yourLoginName>`

Programming Languages

- Use C++ and compiled with g++ (include –lstdc++ automatically) or CC (not include)
- Use C compiled with cc or ANSI C compiled with gcc
  - function prototypes
    ```
    #ifdef ANSI
    void vFoo(int i) {
    #else
    vFoo(i) int i; {
    #endif
    ```
  - comments
    use /* and */ pair instead of //
- Use Java compiled with javac and run with java
  - setup jdk
  - javac foo.java
  - java foo
- Use Python (prefer python 3.x)
- Use debugging tool
  Compiled with –g option and using dbx for cc, gdb for gcc and CC/g++, use jdb for java, and use python3 –m pdb for Python

- C/C++ include file search path using –I<path> option
- C/C++ library
  - C/C++ library search path using –L<path> option
  - C/C++ uses -lm for math library, e.g.,

```
$ gcc –o foo foo.cpp –lm
```

Makefile

- a Makefile contains rules and optional definitions, please read /home/mwang2/P0/Makefile as an example
- a definition (or macro) contains names and their actual values
- a rule contains a tartget, dependant source files, and actions to generate the target, the first rule in a Makefile is the goal to achieve, e.g.,

```
all: <file>.c <file>.h
    g++ –o Pn <file>.c
```

- **Note**: make file actions must start with a tab (not 8 spaces), and you must have a clean target to remove all object files, executables, core file, or temporary files to save bandwidth and space and prevent filtering out by spam filters
- You can use P0/Makefile for other programming assignments by changing all P0 to P*n*. If you use C/C++, changing info.cpp and info.hpp in the P*n*/Makefile to the source files for P*n.*

Coding Standards

- use meaningful id instead of short and brief names
- always use curly braces for if-else etc. statements
- graceful/meaningful error handling to avoid core dump
- no hard coded numbers
- using #if for static condition instead of "if" statement for speed
- program/function/multiple-line/single-line documents
- using blank line and tab (indentation) properly
- using top-down modular design, watch reusability
- function size (150 lines including documentation)
- use module prefix for functions and global variables
- defensive programming by pre-/post-conditions and assertions
- use while and if-then-else to get rid of goto
- allocate and free an object in same function

Common Bug-prone Coding Examples

- access field of a record from the record pointer without checking if the pointer is null
- free objects but not nullify the pointers
- using static variables in recursive calls

Test Data

The test data are put under /home/mwang2/test/coen<num>.

```
$ cd /home/mwang2/test/coen<num>
$ ls
t10.dat t11.dat t19.dat
```

An example test file name is t10.dat, where the first digit 1 for programming assignment #1, and the second digit is serial number from 0. Generally the sample test files use big serial numbers are error input files (e.g., t19.dat), and your program need handle it gracefully (i.e., defensive programming.) However, for real test cases, you can't tell if the input is legal or not merely by checking the serial numbers.

Directory Organization for Automation

- Directory organization:

```
$ cd; mkdir Pn
```

  where *n* can be 0, 1, 2, 3, 4, 5, etc.
- Files: put all source files related to program *n* under P*n*
  - README file (optional)
  - C/C++ Makefile (**note:** action should be lead by a tab)

```
Pn : <file>.c <file>.h
    CC –o Pn –DANSI <file>.c
clean:
    rm –rf *.o Pn
```

  - Java Makefile (**note:** action should be lead by a tab)

```
# NOTE: need to run setup jdk first, and run gmake
DIR = .
SRCS = $(wildcard $(DIR)/*.java)
OBJS = $(SRCS:.java=.class)
all:    $(OBJS)
clean: FORCE
    rm –f *.class core *~
.SUFFIXES: .java .class
.java.class:
    javac $<
FORCE:
```

  - header files and C/C++ files, or Java files
  - test input and output files (optional)
- ***Manual testing***: either type input through keyboard, or redirect input from a file as:

```
./Pn < /home/mwang2/test/coen<num>/txx.dat
```

  where <num> is the course number, e.g., 233, 210, etc.
- Manual submit (***note:*** don't do this, this just shows your why automation is needed):
  - Tar the whole directory

```
$ make clean
$ tar cvf Pn.tar Pn
```

```
$ compress Pn.tar
$ uuencode Pn.tar.Z Pn.tar.Z > Pn.tar.Z.uu
```

- Email P*n*.tar.Z.uu to me and grader:
  ```
  $ mailx -s Pn mwang2@scudc.scu.edu grader < Pn.tar.Z.uu
  ```
- To recover the files
  save the email as P*n*.tar.Z.uu and strip heading
  ```
  $ uudecode Pn.tar.Z.uu
  $ uncompress Pn.tar.Z
  $ tar xvf Pn.tar
  ```

Automation

Always use scripts to do submission and testing. To get the most current Perl scripts:

```
$ cd
$ ln –s /home/mwang2/bin/Autotest<num> Autotest
$ ln –s /home/mwang2/bin/Submit<num> Submit
```

where <num> is the course number, e.g., 233, 210, etc.

Submit

1. trial submission to yourself:
   ```
   $ Submit <your_login_name> /home/<your_login_name>/ Pn
   ```
2. test correctness by Autotest:
   // those 4 steps can be skipped except the grader
   ```
   $ cd; cd AutoTestDir
   $ mkdir <your_login_name>
   $ cd <your_login_name>
   $ mail
   ```
   (save the email as P*n*.m, or save attachment P*n*.tar.gz)
   // end of skipped steps
   ```
   $ ~/Autotest n
   $ cat out
   ```
   (check if your out is correct, **note:** If no, debug your program and then go to step 1.)
3. formal submission:
   ```
   $ cd; Submit Pn
   ```

Autotest

- You can provide parameters to make Autotest more flexible. The way to call Autotest is:
  ```
  $ Autotest <num> [-k] [-t <testdir>] [<name list>]
  ```
  Things in [ ] are optional, <num> should be 0-5, <test dir> should be a full directory path name, and <name list> is a list of login names separated by space. –k option is used to keep run directory.
- For testing before submit, use
  ```
  $ Autotest <num>
  ```
- For testing using your own test cases under <test dir>, do
  ```
  $ Autotest <num> -t <test dir>
  ```
- For grading and only run some of the student's programs, do
  ```
  $ Autotest <num> -k <name list>
  ```

Input Requirements and Example Code

When you code your program, all input should be from stdin, or keyboard. For example, if your program input a number and prints out the square of the number as output.

- If you are a C programmer, your C program should read an integer from keyboard as:
  ```
  // C Program : square.c
  # include <stdio.h>
  main ( ) {
      int i;
      while ( scanf("%d", &i) != EOF ) {
          printf("The square of %d is %d.\n", i, i * i);
      }
  }
  ```
  To compile using gcc:
  ```
  $ gcc square.c
  ```
  Or to compile using cc:
  ```
  $ cc square.c
  ```
- If you are a C++ programmer, your C++ program should be:
  ```
  // C++ Program : square.cpp
  #include <iostream.h>
  main () {
      int i, i2;
      while ( cin >> i ) {
          // Noted that the >> operator automatically
          // suppress white spaces. if you don't want
          // suppress white spaces, use cin.get(ch)
          i2 = i * i;
          cout << "The square of " << i << " is " << i2 << "."
          << endl;
      }
  }
  ```
  To compile using CC:
  ```
  $ CC square.cpp
  ```
- To run the program
  ```
  $ a.out
  25 (you type in 25 and hit return)
  The square of 25 is 625.
  3 (you type in 3 and hit return)
  ```

```
The square of 3 is 9.
^D (you type ctl-D as EOF and hit return)
```

- In order to automate our process, the UNIX redirect is used. Where the input can be from a file, call t0.dat:

```
$ cat t0.dat
25
3
$ a.out < t0.dat
The square of 25 is 625.
The square of 3 is 9.
```

- If you use Java:

```
/** StdIo.java */
import java.io.*;
public class StdIo {
    public static void main(String[ ] args) throws
IOException {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in)
        );
        String s;
        while ( (s = in.readLine( )) != null ) {
            System.out.println(s);
        }
    }
}
```

To run using javac and java:

```
$ javac StdIo.java
$ java StdIo < t0.dat
```

- If you use Python:

```
#!/opt/python-3.4/linux/bin/python3
# or you need to use your machine's python3 path,
#   e.g, /usr/bin/python3
import sys
for line in sys.stdin:
    print(line.rstrip())
```

To run:

```
$ ./stdIo.py < t0.dat
```

Output Requirements and Example Code

When you code your program, all output should go to stdout or stderr. By default, both stdout and stderr will be displayed on the terminal CRT.

- Your C program tee.c can print messages to stdout and stderr:

```
// C Program tee.c
#include <stdio.h>
main ( ) {
   fprintf(stdout, "Output to stdout\n");
   fprintf(stderr, "Output to stderr\n");
}
```

Compile it using gcc:

```
$ gcc tee.c
```

To compile using cc:

```
$ cc tee.c
```

- And here is your C++ program:

```
// C++ Program tee.cpp
#include <iostream.h>
main ( ) {
   cout << "Output to stdout" << endl;
   cerr << "Output to stderr" << endl;
}
```

Compile it using CC:

```
$CC tee.cpp
```

- To run:

```
$ a.out
Output to stdout
Output to stderr
```

- We can redirect only the output into a file, call t0.out, by either UNIX redirect ">":

```
$ a.out > t0.out
Output to stderr
$ cat t0.out
Output to stdout
```

- or the "tee" command:

```
$ a.out | tee t0.out
Output to stdout
Output to stderr
$ cat t0.out
Output to stdout
```

- We can redirect both stdout and stderr to a file by either UNIX redirect ">&", "2>&1", or "|&":

```
$ a.out >& t0.out
$ cat t0.out
Output to stdout
Output to stderr
```

- or the "script" command:

```
$ script junk
$ a.out
Output to stdout
Output to stderr
$ exit
$ cat junk
a.out
Output to stdout
Output to stderr
exit
```

File Input and Output

Autotest requires you to do input from stdin and output to stdout or stderr. When a special requirement is asked to do file input/output, the following are the rules:

- To make Autotest works for file input, all input file should put in full path, e.g., "/home/mwang2/test/foo".
- You should save your file a level up by using an absolute full path file name or a file name like "../foo" instead of "foo", i.e., parallel to the P*n* directory. Note that the Autotest will automatically remove your P*n* directory after it is done (in order to save disk space).

Note that file input and output make Autotest less flexible, thus only can be used when specially specified in program specifications.

Binary File Input and Output

To handle binary file input/output, you use the same way as text input/output. All binary file has to be some multiple of bytes. To read the content of a binary file, use the 'od' command. To know more about the 'od' command, use 'man od'.

Run Time Comparison

Autotest not only run your program many times, each for a test case, but also uses the 'time' command to record the run time of your executable. The time command will display the real (or elapsed) time, the system time, and the user time. To know more about the time command, use 'man time'. The user time is used for run time comparison for sequential programs, the elapsed time for threaded programs.

A Reminder

Since the penalty is 20% per day, you rather start early than postponed until the near the due date. After you finished coding, you only got about 30% done. You need time to debug your program, which may be the longest time you need to make it work; you need time to run the program, especially the NP-hard problem may takes you very long time to run; you need time to do big-Oh calculation and/or documentation; and you also need to optimize in order to get high score on speed.