

The University of Hong Kong
Department of Computer Science and Information Systems
Programming Methodology and Object Oriented Programming
for Software Engineering and Double Degree students (CSIS0396A)

Assignment 1

Assigned: Wednesday Feb 21, 2001

Deadline: Wednesday Mar 7, 2001, 5:00pm.

This is a written assignment. Submit your type-written assignment to assignment box B3 before the deadline. For assignments that are late for at most 48 hours, the marks will be multiplied by 0.8. Assignments are not accepted after that.

1. **(Iostream error handling.)** In the program of lecture notes page 6.5, we have a program that repeatedly read in three numbers and call `do_work()`. The program recovers in case of errors. However, the input is not processed in a strictly line-oriented way. If a line contains more than 3 numbers, the extra ones are considered as in the next line. If a line contains less than 3 numbers, then the numbers in the next line is used as the second or third number.

Modify the program to perform strictly line-oriented input. In case if a line contains only one or two numbers, a warning message “Line *x*: Error: Not enough arguments” should be written to the **cerr** stream, and **do_work** should not be called. In case if a line contains some non-space characters after the third number, a warning message “Line *x*: Warning: extra characters ignored” should be written to the **cerr** stream, and **do_work** should still be called. Lines containing only white-spaces should be ignored. Make sure that the program still handles all the other errors.

2. **(Exceptions.)** (Adapted from The C++ Programming Language 3rd edition, 8.5[5]) Consider the following program.

```
#include <iostream>
using namespace std;
void f(int), g(int);

void f(int n) {
    if (n > 10000 || n ≤ 2)
        throw 0;
    g(2*n-1);
}

void g(int n) {
    if (n > 10000)
        throw 0;
    f(3*n-5);
}

int main() {
    int a;
    cin >> a;
    f(a);
}
```

The program consists of two functions **f** and **g** calling each other. At some point an exception (0) will be thrown. Modify the program so that it prints out which function is initially throwing exception and the number of times that function is called recursively before throwing the exception. For example, if we type 1, the program should print `f throws at depth 1`. **You are only allowed to make the following changes:**

- Adding new classes for exception throwing,
- Modifying the throw statement to throw objects of these classes,
- Adding exception handlers.

Don't change the program in any other way.

3. **(Inlining.)** Consider the following program:

```
#include <iostream>
using namespace std;

int f(int x, int y) {
    return x*x + y*y;
}

int main() {
    int a, b;
    cin >> a >> b;
    int c = f(a, b);
    cout << c << endl;
    return 0;
}
```

- a. Compile the program with the compilation flag **-O2 -g** under Linux, and disassemble `f` and `main` within the debugger. What is the size of code needed to call the function `f`? What is the size of code needed for the function `f` itself? (These number are highly dependent on how you interpret the code. Any reasonable interpretation is accepted. Remember to describe how you get the numbers.)
- b. Change the function `f` to an inline function, and repeat the above step. What is the size of code needed to embed (i.e., inline) the function into the caller?
- c. Let x denote the number of times the program calls `f`, and assume that the function `f` need not be generated if we inline the function. For what range of x will the size of program not increased by inlining `f`?
- d. In practice, sometimes the size of the program will not increase even if the number of calls is outside the range you answered above. Why?
- e. Explain why the designer of C++ think that putting a function in a class signifies that the library writer wants the code to be inlined.

4. **(Dependencies in makefiles.)**

In practice, the dependency generation scheme in page 9.10 of the lecture notes is seldom used, since it generates too many files in the directory. But there is one more fundamental problem with the scheme: the dependency is not correctly updated when header files change.

- a. Use one example to illustrate the problem. In the example, show the change made to the header file and the change of dependencies. Explain why the dependency is not updated correctly.
 - b. Most real-world projects have instead a Makefile rule, typically called **depend**, that creates or update a file (e.g., `.depend`) containing all the dependencies. So whenever the developer changes the header file, they run **make depend** before **make**. Rewrite the Makefile in page 9.10 to use this scheme.
5. (CVS.) By reading the CVS manual and trying the commands out, explain how to do the following tasks. Write down in sequence all commands that are needed. In all cases, assume that the repository is already made, you have already set the variable `CVSROOT`, the repository has a directory called `xyproj`, you already have a working directory with the same name, the working directory is checked out at the latest version in the repository, and you are at the root of the working directory.
- a. Create a branch of the most updated version, called **trimmed**, which has the file **test.in** of the project removed. Switch back to the main trunk after that.
 - b. Find the differences between the copy of the file **gui.cc** in the current directory and the version with tag `rel-0-7`.
 - c. Revert the file **gui.cc**, which is currently at version 1.6, to the copy at version 1.4. Check in the file as version 1.7.
 - d. Update the whole directory with modifications made from other developers, creating new directories if other developers have add them to the repository.