

CSIS0396A Programming Methodology and Object Oriented Programming (Class A) Assignment 1

Tutor: yycau@csis.hku.hk, Deadline Feb 19, 2002, 11:59pm.

Create a CVS repository and do the following three mini-projects in it, using the project name as the CVS directory name. Each mini-project should have a Makefile, which must be created at the beginning of the mini-project. We will read the history recorded in CVS, so make sure that you start using CVS at the beginning of the project. Keep meaningful log message throughout. Handin your CVS repository via the hand-in system of our department.

Hint: it is easy to forget to check in some changes or add some files before handing in the repository. Receiving an assignment that can't be compiled, our tutor will have no choice but to give a very low mark. Be reminded to create an empty directory, check-out all the files from the repository and check whether you can make the programs there before you hand-in.

1. (*Project ttt: Basic usage of separated compilation and make. 20 marks*)

From the web page, download the source file `ttt.cc`, which is a program that plays Tic-tac-toe. It consists of a part describing the rules of the game, a generic game engine and some user interface code. These three parts are clearly marked in the source code. You don't need to understand how the program plays the game. Instead, simply import it into the repository with name `ttt` (the project name). Then split the file into 3 source files, one for each of the 3 parts above. Write a Makefile for it, and make the necessary header files.

Note that the structure named lines should be considered private to the source file defining the rules of the game, and should not be available in other source files.

2. (*Project strings: Advanced usage of make. 20 marks*)

Suppose your program (e.g., a compiler) consists of a part that requires looking up a big sorted array of fixed strings to find the index. Being fixed, you want the strings to be directly available in the program, rather than requiring the program to read a separate file containing them as data. But, because of the sheer size of the array, you don't want having to manually convert the word list into a form that can be compiled as part of a program. Instead, you'd like to write a program that convert the data into code, to be linked into the main program. This program should be extremely simple: it just output some standard text (like `#include <iostream>`, or `const char *wordlist[]={`) and then read the input. For each line read (e.g., `if`), output a line like `"if",`. Finally, print some string (e.g., `};`) to make up a compilable source file. The word list and the representation of the array can change between versions, so you want to include the conversion in the build process rather than run it and forget the original word list.

Write all programs and the Makefile involved. As a test drive, the main program simply needs to read a string from the input, lookup the table using `bsearch` and output the index of the string within the array (or `-1` if not found). The list of strings is available as a file called `wordlist`, which contains one word per line and nothing else. You can assume that the `wordlist` file is already sorted. The Makefile should coordinate the converter program to be compiled and executed to generate the C++ source file that contains the array, and subsequently the object file, and link it with the object file of the main program.

As a test (and to avoid using up all the space of the course account when you hand-in), use

a short word list that contains only the following words: asm, bool, char, const, do, double, else, for, goto, if, int, volatile, while. E.g., if the final program is executed and we type “int”, it should print 10 and exit.

3. (*Project integrate: Plugins. 20 marks*)

A very simple numeric integrator can be made by approximating the area under the given function. The idea is very simple: if we want to integrate a function $f()$ between a and b , we break up the interval between a and b into n equally-sized intervals (where n depends on the accuracy required, e.g., 10000). Then evaluate f at the middle of each interval, add them together, and multiply the sum by the length of each interval. This gives us an approximation of the integration of f between a and b .

It is easy to ask the user for each of the required variables, except the function f . Plugins can help here. Write a program to perform the following tasks:

- Read the name of a plug-in, and load it. Use that name as a function name for f as well. The function should take a **double** as argument and return a **double** typed number. If there is any error (e.g., can't find the plugin, or the plugin does not contain the specified object), report it and terminate.
- Ask for the values of a , b and n . Then calculate and output the numeric integration. You may assume that $b > a$.

Also, write a testing plugin which contains a function calculating x^2 , and test it against the numeric integrator you write.

Instruction to hand-in the repository: change directory to the repository (not working directory!), and type “`tar czvf ~/a1.tgz *`”. Then hand-in the file `a1.tgz` in your home directory.