Laboratory Exercises, C++ Programming

General information:

- The course has four compulsory laboratory exercises.
- You shall work in groups of two students. Sign up for the labs at `sam.cs.lth.se/Labs`.
- The labs require homework. Before each lab session, you must have done as much as you can of the assignments (A1, A2, ...) in the lab, written and tested the programs, and so on. Use the scheduled lab sessions for getting answers to your questions.
- Extra labs are organized only for students who cannot attend a lab because of illness. Notify Sven Gestegård Robertz (`Sven.Robertz@cs.lth.se`) if you fall ill, *before* the lab.

The labs are about:

1. Basic C++ programming, tools, compiling, linking, and finding errors.
2. Introduction to the standard library.
3. Debugging
4. Strings and streams. Function templates.

Practical information:

- You will use many half-written "program skeletons" during the labs. You must download the necessary files before you start working on the lab assignments. The code skeletons are available in a git repository. To get the files, you clone the repository from the command line. First move to the directory where you want your lab code

  ```
  cd my_cpp_dir
  ```

  (where `my_cpp_dir` is the name of your directory) and then clone the repository with

  ```
  git clone https://gitlab.com/cs-lth-cpp/labs-edaf30.git
  ```

  which creates a directory, `labs-edaf30`, containing the lab skeletons, in the current directory. The created directory is itself a git repository so you can now use git to manage your source code. Using version control has many benefits and you are encouraged to use git during the course, but it is not a requirement and it will require some effort. The following sections give a very brief introduction to git, and some pointers to further reading, to get you started.

  If you choose to use git for the course work, and use one of the publicly available services to collaborate with your lab partner, make sure that you make your repositories private. Do not make your solutions publically available on the web.

Good sources of information about C++:

- `http://www.cppreference.com`
- `http://www.cplusplus.com`

Version control using git

This section gives a brief introduction to version control using *git*[1], which is a distributed version control system used by many (most?) projects today. If you want to use a git server to collaborate with your lab partner, or to sync your code between different computers, you need to create a personal copy of the project. If you only work on one computer you can skip the parts about creating a project and collaboration.

Tracking changes with git

The basic commands that are used for tracking changes are

git add *stages* the current state of files or directories for inclusion in the next commit. Please note that if you do `git add` on a directory, it will include *all* files in that directory. Only source code should be put in git; make sure that you don't accidentally add any generated files. If you want to unstage a file (e.g., `foo.cc`), do `git restore --staged foo.cc`.

git commit record the set of staged changes in a new *commit* (a snapshot of the entire repository).

git status lists the (tracked) files that have been changed but not yet committed.

git diff prints a diff between the latest commit and the current files in the workspace, or between two commits. By default `git diff` shows only unstaged changes. Use `git diff --cached` to show already staged changes.

git checkout moves to another (previously committed) version of the repository. Can also be used to revert a file in the working directory to the last commit, discarding any changes.

A common misconception, and a difference to other version control systems, is that in git `git add` does not add "a file" per se, it stages *the current state of a file*. If you make more changes before committing they will not be included in the commit unless you do another `git add`.

That also explains why, in git, the same file can have some changes that are staged to be committed and some that are not, and thus show up twice in the output of `git status`. This is a feature that allows you to put unrelated changes to the same file in separate commits, but the details of how to do that are out of scope here.

Creating your own personal project

If you want to use git to collaborate with your lab partner you can create your own project at gitlab.com (or some other service like github). If you don't have an account you need to create one, then, on gitlab.com, you create a new project with the following steps.

1. From the start page, or under the '+' menu at the top, select *New project*
2. From the *Create new project* page, select *Import project*, choose *Repo by URL* and enter:
    - *Git repository URL*: `https://gitlab.com/cs-lth-cpp/labs-edaf30.git`
    - Leave *Username* and *Password* blank
    - Under *Project name* enter the name you want for your personal project
    - Leave *Mirror repository* unchecked
    - Don't change the automatically generated *Project URL* and *Project slug*
    - Make sure that *Visibility level* is **private**.

Then click *Create project*. After a few seconds, your project will be created.

[1] https://git-scm.com/

3. To collaborate, you must make the project visible to your lab partner. Under *Project information* in the menu on the left, select *Members* and invite your lab partner. The *role* should be *Developer* or *Maintainer* to allow them to push code to the repo.

   The typical setup is that one of the students creates a project on gitlab.com and invites the other. As git is distributed, you will both have local clones of the repository, so ownership of the gitlab project is not important — the gitlab project is only used for synchronizing your local clones.

Cloning your project

Clone your project with the following steps

1. Log into gitlab.com, and go to the project you just created (or was invited to)
2. click on the *Clone* button and copy the URL under *Clone with HTTPS*.
3. open a terminal and go to the directory where you want your code
4. do `git clone URL` (where URL is the URL you copied in the previous step)
5. When using git over https it will ask you for your credentials. Enter the username and password for your account on gitlab.com.

Using git over https will ask you for your credentials each time you do an operation against the remote repository. To avoid that, one can register a ssh key with the git server, and run git over ssh (with the *Clone with SSH* URL) instead. In gitlab, from the menu at the top right corner, select *Edit profile* and then select *SSH keys* from the menu on the left, and follow the instructions.

The details of setting up git over ssh are out of scope for this introduction, and may be complicated if you have not used ssh before. One common problem is that in some cases you need to add a *Host* entry to your `ssh` config file to make ssh use the correct key for gitlab.com.

Collaborating using git

To use a common *remote* repository for collaboration, you need to have a workflow that ensures that you don't cause unnecessary merge conflicts (which happen when both of you have made changes to the same parts in the code). A detailed discussion of that is also out of scope for this course, but a very simple approach is to make sure that you

- never work concurrently (on the same branch), and
- always work on the latest commit.

One way of doing this is to

- pair program,
- always get the latest version of the repo (with `git pull`) at the start of each session,
- create frequent commits (with `git add` and `git commit`) when working, and
- always upload the latest version (with `git push`) at least at the end of each session.

Handling merge conflicts

If you want to work concurrently, a common strategy is to create local *branches* to work in, and then merge (with `git merge`) your changes back to the main branch when you sit together. If you get a *merge conflict*, git will complain and insert both of the conflicting changes into the files, on a format like this:

```
<<<<<<< HEAD
I made a change.
=======
Change in upstream
>>>>>>> c641c9ed5b1dcd5d2082e8758e790fd0143aa391
```

and the conflicts will be listed in the output from `git status`.
To resolve the conflict you must edit the source file, removing the conflict markers, so that the code looks the way you want, and do `git add FILENAME` and `git commit` to resolve the conflict.

Further reading

This section gave a very brief introduction to git. If you want to use git effectively you will need to put some effort into understanding and learning what git is, and how to use it.

- There is a book and a reference manual at `https://git-scm.com/doc`
- There are a set of good tutorials and articles about git (most of them are not specific to Bitbucket) at `https://www.atlassian.com/git/tutorials`
- You can access the manual pages for git with the command

  ```
  git help
  ```

  which gives an overview of help topics and commands, or you can get help for a particular command with e.g.,

  ```
  git help add
  ```

  which gives the manual page for the subcommand `add`.

1 Basic C++ Programming, Compiling, Linking, Debugging

Objective: to introduce C++ programming in a Unix environment.

Read:

- Book: basic C++, variables and types including expressions, statements, functions, simple classes, `ifstream`, `ofstream`.
- GCC manual: `http://gcc.gnu.org/onlinedocs/`
- GNU make: `http://www.gnu.org/software/make/manual/`

1 Introduction

Different C++ compilers are available in a Unix environment, for example `g++` from GNU (see `http://gcc.gnu.org/`) and `clang++` from the Clang project (see `http://clang.llvm.org/`). The GNU Compiler Collection, GCC, includes compilers for many languages, the Clang collection only for "C-style" languages. `g++` and `clang++` are mostly compatible and used in the same way (same compiler options, etc.). In the remainder of the lab we mention only `g++`, but everything holds for `clang++` as well.

Actually, `g++` is not a compiler but a "driver" that invokes other programs:

Preprocessor (`cpp`): takes a C++ source file and handles preprocessor directives (`#include` files, `#define` macros, conditional compilation with `#if` and `#ifdef`).

Compiler: the actual compiler that translates the input file into assembly language.

Assembler (`as`): translates the assembly code into machine code, which is stored in object files.

Linker (`ld`): collects object files into an executable file.

A C++ source code file is recognized by its extension. The two commonly used extensions are *.cc* (recommended by GNU) and *.cpp*. The source files contain definitions. To enable separate compilation, declarations are collected in header files with the extension *.h*. To distinguish C++ headers from C headers the extensions *.hpp* or *.hh* are sometimes used. We will use *.h*.

A C++ program normally consists of many classes that are defined in separate files. It must be possible to compile the files separately. The program source code is usually organized like this (a main program that uses a class `Editor`):

- Define the editor class in a file *editor.h:*

```
#ifndef EDITOR_H  // include guard
#define EDITOR_H

#include <string> // include necessary headers here
class Editor {
public:
    /* Creates a text editor containing the text t */
    Editor(const std::string& t) : text(t) {}
    size_type find_left_par(size_type pos) const;

     // ... functions to edit the text (insert and delete characters)
private:
    std::string text;
};
#endif
```

- Define the class member functions in a file *editor.cc*

```
#include "editor.h"
// include other necessary headers

size_type Editor::find_left_par(size_type pos) const { ... }
...
```

- Define the main function in a file *test_editor.cc*:

```
#include "editor.h"
// include other necessary headers

...
int main() {
    Editor ed( ... );
    test_equals( ed.find_left_par(15), 11);
    ...
}
```

The include guard is necessary to prevent multiple definitions of names. Do *not* write function definitions in a header file (except inline functions and function templates).

Compiling and running a program: When compiling from the command line, the g++ command line looks like this:

```
g++ [options] [-o outfile] infile1 [infile2 ...]
```

The *.cc* files are compiled separately. The resulting object files (*.o* files) are linked into an executable file *test_editor*, which is then executed:

```
g++ -std=c++11 -c editor.cc
g++ -std=c++11 -c test_editor.cc
g++ -o test_editor test_editor.o editor.o
./test_editor
```

The `-c` option directs the driver to stop before the linking phase and produce an object file, named as the source file but with the extension *.o* instead of *.cc*.

In order to reduce compilation times, separate compilation as in the above example is usually desired, but gcc also supports giving multiple source files in the same command. Then, that means to both compile and link. The three calls to g++ above can therefore also be written

```
g++ -std=c++11 -o test_editor test_editor.cc editor.cc
```

Please note that the header files are not included in the g++ command line – they are inserted by the preprocessor `#include` directives.

A1. Write a "Hello, world!" program in a file *hello.cc*, compile and test it.

The next exercise illustrates separate compilation, and the difference between compiler errors and linker errors. We have a program consisting of two source files, `separate_main.cc` containing the main function, and `separate_fn.cc` containing a function that is used in `main()`. In addition to that, the function is declared in `separate_fn.h`.

A2. First, compile and link the program with the command

```
g++ -std=c++11 -o separate_main separate_main.cc separate_fn.cc
```

Verify that an executable is built and that it works as expected.

A3. Then compile the source files separately, and link the produced object files with the commands

```
g++ -std=c++11 -c separate_main.cc
g++ -std=c++11 -c separate_fn.cc
```

The option `-c` instructs the compiler to just compile, but not link. Here, that creates the two *object files* separate_main.o and `separate_fn.o`. Then, link the program with the command

```
g++ -std=c++11 -o separate_main separate_main.o separate_fn.o
```

Again, verify that the executable is built and that it works.

A4. Try to compile and link just the file containing `main()`, with the command

```
g++ -std=c++11 -o separate_main separate_main.cc
```

Make sure that you understand the error message. Is it a compiler error or a linker error? What does it mean, and what causes it?

2 Compiler options and messages

There are more options to the g++ command than were mentioned in section 1. Your source files must compile correctly using the following command line:

```
g++ -c -O2 -Wall -Wextra -pedantic-errors -Wold-style-cast -std=c++11 file.cc
```

Short explanations (you can read more about these and other options in the `gcc` manual):

`-c`	just compile (i.e., produce object code), do not link
`-O2`	optimize the object code (perform nearly all supported optimizations)
`-Wall`	print most warnings
`-Wextra`	print extra warnings
`-pedantic-errors`	produce errors for using non-standard language extensions
`-Wold-style-cast`	warn for old-style casts, e.g., `(int)` instead of `static_cast<int>`
`-std=c++11`	follow the C++-11 standard
`-stdlib=libc++`	Clang only — use Clang's own standard library instead of GNU's `libstdc++`

Do not disregard warning messages. Even though the compiler chooses to "only" issue warnings, your program is most likely erroneous or at least questionable. It is strongly recommended that you add the option `-Werror`, which treats warnings as errors.

Some of the warning messages are produced by the optimizer and will therefore not be output if the `-O2` flag is not used. But you must be aware that optimization takes time, and on a slow machine you may wish to remove this flag during development to save compilation time. Another reason for compiling without optimization is to facilitate debugging, see section 7.

It is important that you become used to reading and understanding the GCC error messages. The messages are sometimes long and may be difficult to understand, especially when the errors involve the standard library class templates (or any other complex class templates).

3 Introduction to make

You have to type a lot in order to compile and link C++ programs — the command lines are long, and it is easy to forget an option or two. You also have to remember to recompile all files that depend on a file that you have modified.

There are tools that make it easier to compile and link, "build", programs. These may be integrated development environments (Eclipse, Visual Studio, . . .) or separate command line tools. In Unix, *make* is the most important tool. Make works like this:

- it reads a "makefile" when it is invoked. Usually, the makefile is named *Makefile*[2].
- The makefile contains a description of dependencies between files (which files that must be recompiled/relinked if a file is updated).
- The makefile also contains a description of how to perform the compilation/linking.

As an example, we take the program from section 1. The files *editor.cc* and *test_editor.cc* must be compiled and then linked. Instead of typing the command lines, you just enter the command `make`. Make reads the makefile and executes the necessary commands.

A minimal makefile, without all the compiler options, looks like this:

```
# The following rule means: "if test_editor is older than test_editor.o
# or editor.o, then link test_editor".
test_editor: test_editor.o editor.o
        g++ -o test_editor test_editor.o editor.o

# Rules to create the object files.
test_editor.o: test_editor.cc editor.h
        g++ -std=c++11 -c test_editor.cc

editor_o: editor.cc editor.h
        g++ -std=c++11 -c editor.cc
```

A rule specifies how a file (the *target*), which is to be generated, depends on other files (the *prerequisites*). The line following the rule contains a shell command, a *recipe*, that generates the target. The recipe is executed if any of the prerequisites are older than the target. It must be preceded by a TAB character, *not* eight spaces.

A5. The file *Makefile* in the *lab1* directory contains the makefile described above. The files *editor.h*, *editor.cc*, and *test_editor.cc* are in the same directory. Experiment:

Run make. Run make again. Delete the executable program and run make again. Change one or more of the source files (it is sufficient to `touch` them) and see what happens. Run `make test_editor.o`. Run `make notarget`. Read the manual [3]and try other options.

3.1 Invoking make

Make has many command line options (see the manual for details). A few quite useful ones are

-f filename specify the name of the makefile to use, e.g. `make -f MyMakefile`

-B unconditionally make all targets

-C dir change to directory *dir* and run make there. Useful to recurse into subdirectories.

-n "dry run": just print the commands that would be executed but do not execute them

[2] If no filename is given `make` looks for `Makefile` or `makefile` in the current directory.
[3] See section 9 (*How to run make*) of the manual. The options are summarized in section 9.7.

4 More Advanced Makefiles

4.1 Implicit Rules

Make has *implicit rules* for many common tasks, for example producing *.o*-files from *.cc*-files. The recipe for this task is: `$(CXX) $(CPPFLAGS) $(CXXFLAGS) -c -o $@ $<`
`CXX`, `CPPFLAGS`, and `CXXFLAGS` are variables that the user can define. The expression `$(VARIABLE)` evaluates a variable, returning its value. `CXX` is the name of the C++ compiler, `CPPFLAGS` are the options to the preprocessor, `CXXFLAGS` are the options to the compiler. `$@` expands to the name of the target, `$<` expands to the first of the prerequisites.

There is also an implicit rule for linking, where the recipe (after some variable expansions) looks like this: `$(CC) $(LDFLAGS) $^ $(LOADLIBES) $(LDLIBS) -o $@`
`LDFLAGS` are options to the linker, such as `-Ldirectory` which makes the linker look for library files in *directory*. `LOADLIBES` and `LDLIBS`[4] are variables intended to contain libraries, such as `-llab1` or `mylibrary.a`. The variable `$^` expands to all prerequisites. So this is a good rule, except for one thing: it uses `$(CC)` to link, and `CC` is by default the C compiler `gcc`, not `g++`. But if you change the definition of `CC`, the implicit rule works also for C++:

```
# Define the linker
CC = $(CXX)
```

4.2 Phony Targets

When invoked without arguments, make builds the first target that it finds in the makefile. By convention, the first target should be named *all*, and it should make all the targets. But suppose that a file named *all* exists in the directory that contains the makefile. If that file is newer than the *test_editor* file, a make invocation will do nothing but say `make: Nothing to be done for 'all'.`, which is not the desired behavior. The solution is to specify the target *all* as a *phony target* (a target that is not an actual file), like this:

```
all: test_editor
.PHONY: all
```

Another common phony target is *clean*. Its purpose is to remove intermediate files, such as object files, and it has no prerequisites. It typically looks like this:

```
.PHONY: clean
clean:
        rm -f *.o test_editor
```

4.3 Generating Prerequisites Automatically

While you're working with a project the prerequisites are often changed. New `#include` directives are added and others are removed. In order for make to have correct information about the dependencies, the makefile must be modified accordingly. This is a tedious task, and it is easy to forget a dependency.

The C++ preprocessor can be used to generate prerequisites automatically. The option `-MMD`[5] makes the preprocessor look at all `#include` directives and produce a file with the extension *.d* which contains the corresponding prerequisite. Suppose the file *test_editor.cc* contains the following `#include` directive:

```
#include "editor.h"
```

The compiler produces a file *test_editor.d* with the following contents:

```
test_editor.o: test_editor.cc editor.h
```

The *.d* files are included in the makefile, so it is equivalent to writing the dependencies manually.

[4] There doesn't seem to be any difference between `LOADLIBES` and `LDLIBS` — they always appear together and are concatenated. Use `LDLIBS`.

[5] The option `-MMD` generates prerequisites as a side effect of compilation. If you only want the preprocessing but no actual compilation, `-MM` can be used.

4.4 Putting It All Together

The makefile below can be used as a template for makefiles in many (small) projects. To add a new target you must:

1. add the name of the executable to the definition of PROGS,
2. add a rule which specifies the object files that are necessary to produce the executable.

```
# Define the compiler and the linker. The linker must be defined since
# the implicit rule for linking uses CC as the linker. g++ can be
# changed to clang++.
CXX = g++
CC  = $(CXX)

# Generate dependencies in *.d files
DEPFLAGS = -MT $@ -MMD -MP -MF $*.d

# Define preprocessor, compiler, and linker flags. Uncomment the # lines
# if you use clang++ and wish to use libc++ instead of GNU's libstdc++.
# -g is for debugging.
CPPFLAGS =  -std=c++11 -I.
CXXFLAGS =  -O2 -Wall -Wextra -pedantic-errors -Wold-style-cast
CXXFLAGS += -std=c++11
CXXFLAGS += -g
CXXFLAGS += $(DEPFLAGS)
LDFLAGS =   -g
#CPPFLAGS += -stdlib=libc++
#CXXFLAGS += -stdlib=libc++
#LDFLAGS +=  -stdlib=libc++

# Targets
PROGS = test_editor print_argv

all: $(PROGS)

# Targets rely on implicit rules for compiling and linking
test_editor: test_editor.o editor.o
print_argv: print_argv.o

# Phony targets
.PHONY: all clean

# Standard clean
clean:
	rm -f *.o $(PROGS)


# Include the *.d files
SRC = $(wildcard *.cc)
include $(SRC:.cc=.d)
```

A6. The makefile with automatic dependencies is in the file *MakefileWithDeps*. Rename this file to *Makefile*, and experiment. The compiler will warn about unused parameters. These warnings will disappear when you implement the member functions.
Look at the generated *.d* files. Use this makefile to build your “Hello world!” program.

5 The terminal and I/O streams

The programs you will write in the labs use the terminal for input and output, so you need to know the basics of how terminal I/O works. When a program is started from the terminal, there are streams connected to the program: *standard in* (available as the variable std::cin in C++, stdin in C, and System.in in Java) is an input stream where the text typed on the keyboard can be read, *standard out* (std::cout / stdout / System.out) is an output stream used for writing text to the terminal. There is also a second output stream, *standard error* (std::cerr / stderr / System.err) which is used to print error messages. Both standard out and standard error are by default printed to the terminal, but can be independently redirected.

To illustrate this, we have the programs read-words.cc and example-out.cc in the directory stream-examples. Build the programs with the Makefile in that directory.

How to terminate input in the terminal

read-words.cc reads words from standard in and prints them on separate lines to standard out. The program will read until the end of the stream, but with standard input connected to a terminal the stream will not reach the end until the terminal is closed. To make the program finish without closing the terminal you can send an end-of-file character by pressing <Ctrl-d> (i.e., hold Ctrl and press d) as the first character on a new line. Note also that the terminal buffers input, so nothing is passed to the program until you hit <Return>.

To kill the program running in the terminal, press <Ctrl>-c.

How to redirect input and output streams

If you run ./read-words, it reads from and writes the output to the terminal, but you can *redirect* the streams so that the program reads or writes a file instead. To redirect standard out, use > and a filename[6]. For instance, if you give the command line

```
./read-words > out.txt
```

The output will be written to a file out.txt instead of to the terminal. That means that if you now type

```
Testing testing 1 2 3
<Ctrl-d>
```

nothing will be printed to the terminal, but the file out.txt will contain

```
Testing
testing
1
2
3
```

To print the contents of that file to the terminal, you can use the utility cat:

```
cat out.txt
```

In the same way, standard in can be redirected to read from a file instead of the terminal. For instance, to pass its own source code to read-words, give the command

```
./read-words < read-words.cc
```

Output streams can also be redirected to standard in of another program with a "pipe" (|). For example, to send that output to a *pager* so that you can scroll or search in the output, give the command

```
./read-words < read-words.cc | less
```

Then you can scroll with the arrow keys. Hit h to see available commands, or q to quit.

The program example-out.cc writes text to the two output streams. If run with

```
./example-out
```

all of its output will be written to the terminal. The two streams can be redirected independently.

[6] Redirecting with > file will overwrite file, to append, use >> file. Note that > will overwrite an existing file without asking.

The invocation

```
./example-out > out.txt
```

will redirect stdout to a file `out.txt`, so only

```
And this is written to stderr
And some more to stderr
```

is printed in the terminal. In the same way, stderr can be redirected with

```
./example-out 2> err.txt
```

It is also possible to redirect both streams to separate files with

```
./example-out > out.txt 2> err.txt
```

or errors to a file and standard out to a pager with

```
./example-out 2> err.txt | less
```

To redirect both stdout and stderr to the same file, use

```
./example-out &> both.txt
```

6 Writing small programs

A7. Editors for program text usually help with matching of parentheses: when you type a right parenthesis, `)`, `]` or `}` the editor highlights the corresponding left parenthesis.

Example with matching parentheses marked (the dots ... represent a sequence of characters except parentheses):

```
...(...(...[...]...)...)...{...}...
           |---|           |---|
       |-----------|
   |-------------------|
```

Implement the function `find_left_par` in the class Editor, test with editortest.cc.

A8. Implement two functions for encoding and decoding:

```
/* For any character c, encode(c) is a character different from c */
unsigned char encode(unsigned char c);

/* For any character c, decode(encode(c)) == c */
unsigned char decode(unsigned char c);
```

Use a simple method for coding and decoding. Test your encoding and decoding routines with `test_coding.cc`.

Then write a program, `encode`, that reads a text file[7], encodes it, and writes the encoded text to another file. The program can ask for a filename as in the following execution

```
./encode
enter filename.
myfile
```

and write the encoded contents to `myfile.enc`.

Alternatively, you can give the file name on the command line, like this:

```
./encode myfile
```

Command-line parameters are passed to `main` in an array of C-strings. The prototype to use is `int main(int argc, const char** argv)`. See `print_argv.cc` for an example of how to use command line arguments.

A third option is to read from `std::cin`, write to `std::cout`, and use stream redirection as outlined in Section 5. Then you have to invoke the program like:

```
./encode < myfile > myfile.enc
```

[7] Note that you cannot use `while (infile >> ch)` to read all characters in `infile`, since >> skips whitespace (why?). Use `infile.get(ch)` instead. Output with `outfile << ch` should be ok, but `outfile.put(ch)` looks more symmetric.

Write another program, `decode`, that reads an encoded file, decodes it, and writes the decoded text to another file *FILENAME.dec*. Add rules to the makefile for building the programs.

Test your programs and check that a file that is first encoded and then decoded is identical to the original. For this, you can use the Unix `diff` command:

```
diff myfile myfile.enc.dec
```

Note: the `encode` and `decode` programs will work also for files that are UTF-8 encoded. In UTF-8, characters outside the “ASCII range” are encoded in multiple bytes, and the `encode` and `decode` functions will be called on each byte in such a character. While some characters in the encoded file may not be valid UTF-8, it whould still hold that decoding such an encoded file will produce a file equal to the original, as `decode(encode(c)) == c` should hold for each byte in the original file.

7 Finding Errors

With a debugger, such as `lldb` or `gdb`[8], you can control a running program (step through the program, set breakpoints, inspect variable values, etc.). This section presents two common debuggers, `lldb` from the LLVM project and the GNU debugger `gdb`. Which one you choose to use is up to you, based on personal preference and availability.

To make the source code available to the debugger, you must instruct the compiler and linker to generate debug information with the option `-g`. Preferably you should also turn off optimization. The optimizer may reorder statements which makes it confusing to single-step through the code. Variables may also be optimized away (i.e., not stored in memory) so that they are not visible to the debugger. Optimization is turned off with the `-O0` option. From g++ version 4.8 there is a new option `-Og`, which turns on some optimizations that do not tend to conflict with debugging.

7.1 LLDB

A program is executed under control of `lldb` like this:

```
lldb ./program
```

or, if `program` takes command line arguments

```
lldb -- ./program test testing
```

where `--` ensures that the arguments to `program` are not interpreted as arguments to `lldb`.

You can also start lldb without arguments and load the program to debug from the prompt. For instance, with the `print_argv` example program you can start lldb without arguments:

```
lldb
```

and then load the file at the lldb prompt:

```
(lldb) target create print_argv
Current executable set to './print_argv' (x86_64)
(lldb) process launch -- arg1 arg2 testing testing --foo -t
```

which will run the program and print

```
Process 97951 launched: './print_argv' (x86_64)
argc=7
[./print_argv] [arg1] [arg2] [testing] [testing] [--foo] [-t]
Process 97951 exited with status = 0 (0x00000000)
```

For gdb compatibility, `run` and `r` are aliases for `process launch`.

If you want to launch a process without running it, so that you can set breakpoints or otherwise examine the program, you can give the `-s` option to `process launch`:

```
(lldb) process launch -s  -- arg1 arg2 testing testing --foo -t
Process 98059 stopped
* thread #1, stop reason = signal SIGSTOP
    frame #0: 0x000000010000419c dyld`_dyld_start
dyld`_dyld_start:
->  0x10000419c <+0>: popq   %rdi
    0x10000419d <+1>: pushq  $0x0
    0x10000419f <+3>: movq   %rsp, %rbp
    0x1000041a2 <+6>: andq   $-0x10, %rsp
Target 1: (print_argv) stopped.
Process 98059 launched: './print_argv' (x86_64)
```

To continue executing the program, give the command `continue` (which can be abbreviated `c`):

```
(lldb) c
Process 98059 resuming
argc=7
[./print_argv] [arg1] [arg2] [testing] [testing] [--foo] [-t]
Process 98059 exited with status = 0 (0x00000000)
```

[8] On MacOS, gdb is not installed by default, and is a bit complicated to install as the system requires the executable of the debugger to be signed. The default debugger on MacOS is `lldb`. Both are very competent debuggers, so if you don't already know `gdb`, learning `lldb` may be a better option. `lldb` is also available on linux.

Another useful variant of that is to set a breakpoint in `main()` and then run the program:

```
(lldb) b main
Breakpoint 1: where = print_argv‘main + 16 at print_argv.cc:20, address = 0x0000000100000cbd
(lldb) r
Process 98110 launched: ’./print_argv’ (x86_64)
Process 98110 stopped
* thread #1, queue = ’com.apple.main-thread’, stop reason = breakpoint 1.1
    frame #0: 0x0000000100000cbd print_argv‘main(argc=7, argv=0x00007ffeefbff268)
    at print_argv.cc:20
   17
   18   int main(int argc, char** argv)
   19   {
-> 20       cout << "argc=" << argc << endl;
   21
   22       for(int i=0; i != argc; ++i){
   23           std::string arg{argv[i]}; // create a std::string for the argument
Target 1: (print_argv) stopped.
```

Then you can continue, single-step, or otherwise inspect the program.
Some useful commands:

`help [command]`	Get help about debugger commands.
`run [args...]`	Run the program (with arguments).
`continue (c)`	Continue execution.
`next (n)`	Step to the next line *over* function calls.
`step (s)`	Step to the next line *into* function calls.
`finish`	Continue until just after the current function returns.
`thread until <line>`	Continue until line, or until the current function returns.
`bt`	Print the call stack ("backtrace").
`up [count]`	move up the call stack (one step if no argument given)
`down [count]`	move down the call stack
`list [nbr | function]`	List 10 lines around the current line or around line number `nbr` or start of `function`. list without arguments shows the next 10 lines.
`breakpoint`	Set a breakpoint. See the help for details, and below for examples.
`expression`	Evaluate an expression. Can be used to inspect or change variables. For compatibility with gdb, `print` and `p` are aliases for `expression`
`display expr`	Print `expr` every time the program stops.
`watchpoint var`	Set a watchpoint, i.e., watch all accesses to a variable. Can be very slow but can be the best solution to find some bugs.

To reduce typing in both lldb and gdb, just hitting `<enter>` at the prompt repeats the last command, and the prompts have tab completion. Command names can also be abbreviated to the shortest unambiguous string. For instance, `breakpoint set` can be abbreviated to `br se`.

Breakpoints

A breakpoint lets you stop the program at a certain position. Breakpoints are set with the command `break` (see `help break` for details).

A breakpoint is created with `break set` and then typically either `-n function name` or `-l line number`. To specify which source file to break in, `-f filename` can be added. E.g.:

```
(lldb) break set -n print_string
```

will set a breakpoint at the beginning of the function `print_string`, and

```
(lldb) break set -l 22 -f print_argv.cc
```

will set a breakpoint on line 22 in the file `print_argv.cc`

Two other useful features of breakpoints are *conditions*, which lets you control when a breakpoint actually stops the program, and *commands* that are executed when a breakpoint is hit.

For instance, if we want to break in the function `print_string` but only when the length of the string `str` is one character, we could instead do

```
break set -n print_string -c "str.size() == 1"
```

If we later wanted to break for strings shorter than four characters, we could do `break list` to find the number of the breakpoint (here: 5) and then modify that breakpoint with

```
break modify -c "str.size() < 4" 5
```

If we want to just print `str` and continue executing, we can add a command with

```
break command add -o "print str" 5
```

and make it auto-continue after running its commands with

```
break modify -G true 5
```

Breakpoints is a very useful tool for some debugging tasks, and learning the basics is strongly recommended. Some useful breakpoint commands are

break list lists breakpoints

break enable/disable enables/disables breakpoints

break modify modify a breakpoint

break command add debugger commands to run when the breakpoint is hit

See the built-in help for details.

GUI mode

`lldb` also has a "GUI" mode which splits the window into panes for the source code, backtrace, variables, and registers. This mode is entered with the command `gui`, and exited by hitting `<Escape>` twice or selecting `Exit` from the menu. In GUI mode, some commands like `step`, `next`, `continue` and running to a selected line are available. See the built-in help for the active pane (which you get by hitting `h`) for details. To switch panes, hit `<Tab>`.

Redirecting output

If your program generates a lot of output to the terminal, that output is generated in the same window where the debugger is running, wich may make it hard to read. One solution to this is to redirect the output to another terminal.

Open another terminal window and find the name of the terminal device:

```
echo $TTY
```

which should output something like

```
/dev/ttys079
```

Then, in lldb, use that device name to redirect the output with the options `-o` (for stdout) and `-e` (for stderr), e.g:

```
process launch -o /dev/ttys079 -- foo bar 1 2 3
```

which should run the program and send the output (of stdout, in this example) to the other terminal. Note that this also allows you to send the stdout and stderr to different terminals.

7.2 GDB

Another very popular and powerful debugger is the GNU debugger, gdb. A program is executed under control of `gdb` like this:

```
gdb ./program
```

or

```
gdb --args ./program some command line arguments
```

Gdb and lldb have very similar sets of commands, and the basic ones are the same (i.e., lldb has aliases that match the gdb command names), but there are differences.

For instance, gdb has a command `start` which sets a temporary breakpoint in `main` and runs there. Also, the breakpoint commands are different. For setting breakpoints, gdb has the single command `break` (abbreviated `b`) which can take either a function name or a line number (see the help for details). Breakpoints are listed with the command `info break`, enabled, disabled or deleted with the commands `enable`, `disable`, and `delete`. Breakpoint conditions are in gdb set with the command `cond`, and breakpoint commands are set with `commands`. See `help breakpoints` for all commands related to breakpoints.
`gdb` also has a mode (called "text user interface") which splits the window into one window for the source code and one for the command prompt. This mode is toggled with the key sequence `ctrl-x a` (i.e., first type `ctrl-x` and then type an `a`). In the code window, the arrow keys scroll the source code, in the command window the arrow keys move in the command history. The key sequence `ctrl-x o` jumps to the other window. See also the commands `tui` and `layout`.

A9. Run the test programs under control of `lldb` or `gdb`, try the commands.

8 Memory-Related Errors

In Java, many errors are caught by the compiler (use of uninitialized variables) or by the runtime system (addressing outside array bounds, dereferencing null pointers, etc.). In C++, many such errors are not caught but have *undefined behaviour* (UB) and may result in erroneous results or faults during program execution. Furthermore, you get no information about where in the program the error occurred. Since deallocation of dynamic memory in C++ is manual, you also have a whole new class of errors (dangling pointers, double delete, memory leaks).

8.1 Google sanitizers

For finding errors related to memory management and undefined behaviour, both gcc and clang can use the google sanitizers. This is a library that instruments the code with a set of runtime checks, and is enabled by compiling and linking with `-fsanitize=<SANITIZER>` (where the possible values for `<SANITIZER>` include `address`, `leak`, or `undefined`). Note that you must compile and link with the same sanitizer. See `https://github.com/google/sanitizers` for more information.

A10. Study and run the examples in the directory *buggy_programs*. Build and run each program both without and with sanitizers. The file `README.txt` in that directory contains brief instructions. Compare the results with that of valgrind (run valgrind on the versions built without sanitizers).

A11. For the programs that crash when built without sanitizers, run the non-sanitizer executable in the debugger and see if you can use the debugger to get a stack-trace of the crashing program. Compare to the output from the debugger to that of the sanitizer.

You may not understand the details of some of the buggy program examples yet as some concepts have not yet been explained in the course. For now, focus on using the tools. Also, the sanitizers produce a lot of output, and some of it is very low-level and only meaningful to experts (who also only read all the details when they really need to). The important parts are what kind of error has been detected, where in the code it is, and how the program got there.

Sometimes the optimizer can make a program run as intended despite having undefined behaviour. It can therefore be a good idea to turn off optimization (with `-O0`) for the sanitizer to find the error. To get more readable messages compile the program with debug symbols (`-g`).

8.2 Valgrind

Valgrind is a tool (available under Linux (including Windows WSL) and Mac OS X (although it takes some time for it to be ported to a new version of the OS) that helps you find memory-related errors at the precise locations at which they occur. It is essentially a virtual machine that adds a set of run-time checks around the code. This results in slower program execution, but this is more than compensated for by the reduced time spent in searching for bugs.

Valgrind is easy to use. Compile and link as usual (without sanitizers), then execute like this:

```
valgrind ./program
```

When an error occurs, you get an error message and a stack trace (and a lot of other information). At the end of execution, valgrind prints a "leak summary" which indicates the amount of dynamic memory that hasn't been properly freed.

A12. Go to the directory *buggy_programs* and build the programs using *Makefile*. Then run each program under valgrind [9] and see what problems, if any, it finds. Make sure you understand the messages printed by valgrind.

To get more detailed information, you can give the option `--leak-check=full` to valgrind. E.g.,

```
valgrind --leak-check=full ./program
```

Another thing to be aware of is that valgrind can give false positives in some cases due to, among other things, implementation details – or problems – in the standard library. To supress such errors, you can create a *suppressions file*, containing rules that control which errors are printed. That is out of scope for this course, but please refer to the documentation if you are interested.

9 CMake, a system for generating build scripts

Make is a standard tool for building programs, but it is quite low-level and for larger projects, a more high-level build system is commonly used. One example is CMake, which is used to specify how to build a project in an operating system and in a compiler-independent manner. CMake then generates the required Makefiles (or project files for one of the supported IDEs).

The directory *cmake-example* contains a small project, consisting of an example library, a configuration file, and a main program. This gives a brief overview of how CMake works. Please note that that example includes details that you may not need for this course, such as generating the header file *SimpleConfig.h* and inserting values into that file from cmake.

With cmake, you usually build the project in a directory separate from the source, typically named *build*. This has the advantages that you can easily make several separate builds (e.g. testing and production) simply by doing them in separate build directories. It also means that the generated files are kept separate from the source code, so that removing them is done by simply removing the entire build directory.

The steps to create the build files and then build the project, assuming you are standing in the project root directory, in this case *cmake-example*, are:

```
mkdir build && cd build
cmake ..
make
```

A13. (optional) You should now separate your programs into a library named `lab1` [10] (containing the files `coding.o` and `editor.o`) and the main programs. Refer to Section 10 for an introduction to libraries. Note that it is the compiled object files that go into the library, but that the specification in *CMakeLists.txt* contain the source files used to build the library.

[9] If you are on Mac OS and fail to install valgrind, you can skip this and use the google sanitizers instead.

[10] Library names and file names can be confusing. The library files typically have the prefix *lib* and a suffix depending on the type of library and OS. A statically linked library with the name lab1 is typically stored in a file called *liblab1.a*.

Here we will use cmake to generate the commands that create the library files. Study the *cmake-example* project and then write *CMakeLists.txt* files for lab1 and use that to build the programs. The source files from which the *lab1* library is built should be moved to a subdirectory and made into a library with its own *CMakeLists*.txt. (Copy all your files to a separate directory for this task, in order to keep your makefile-only solution.)
Verify that both the library and the main program is rebuilt if you change the library source.

A14. (optional) Study *buggy_programs/CMakeLists.txt* to see how different options can be set for debug and release builds. When building, use different directories for the different builds.

10 Object Code Libraries

A lot of software is shipped in the form of libraries, e.g. class packages. In order to use a library, a developer does not need the source code, only the object files and the headers. Libraries may contain thousands of object files and cannot reasonably be shipped as separate files. Instead, the files are collected into library files that are directly usable by the linker.

10.1 Static Libraries

The simplest kind of library is a *static library*. The linker treats the object files in a static library in the same way as other object files, i.e., all code is linked into the executable files. In Unix, a static library is an *archive file*, *lib<name>.a*. In addition to the object files, an archive contains an index of the symbols that are defined in the object files.

A collection of object files *f1.o*, *f2.o*, *f3.o*, ..., are collected into a library *libfoo.a* using the `ar` command:

```
ar crv libfoo.a f1.o f2.o f3.o ...
```

(Some Unix versions require that you also create the symbol table with `ranlib libfoo.a`.) In order to link a program *main.o* with the object files *obj1.o*, *obj2.o* and with the object files in the library *libfoo.a*, you use the following command line:

```
g++ -o main main.o obj1.o obj2.o -L. -lfoo
```

The linker searches for libraries in certain system directories. The `-L.` option makes the linker search also in the current directory.[11] The library name (N.B.! the *name*, *without* the prefix `lib` and the suffix `.a`) is given after `-l`.

For debugging, it can sometimes be interesting to look at the symbols defined in an object file or library. For this, the utility `nm` can be used. The symbols in an object file `foo.o` is listed with

```
nm foo.o
```

By default, `nm` lists all symbols in a file. To restrict it to just defined or undefined symbols, the options `--defined-only` and `--undefined-only` can be used. If you run it, you see that c++ function names are *mangled* to avoid name clashes for overloaded functions and member functions. With GNU nm, the option `--demangle` makes the names more readable. If that option is not available, the utility `c++filt` can be used to demangle symbol names, e.g.

```
nm --undefined-only main.o | c++filt
```

[11] You may have several `-L` and `-l` options on a command line. Example, where the current directory and the directory */usr/local/mylib* are searched for the libraries *libfoo1.a* and *libfoo2.a*:

```
g++ -o main main.o obj1.o obj2.o -L. -L/usr/local/mylib -lfoo1 -lfoo2
```

A15. (optional) Collect the object files *editor.o* and *coding.o* in a library *liblab1.a*. Change the makefile so the programs (*test_editor* , *encode*, *decode*) are linked with the library. The `-L` option belongs in `LDFLAGS`, the `-l` option in `LDLIBS`.

Note that this does not tell make how to create *liblab1.a*. For that, add a rule

```
liblab1.a: coding.o editor.o
    ar crv liblab1.a coding.o editor.o
```

Note that you cannot easily write rules to make the programs depend on the lib, so you must first `make liblab1.a` and then `make`. See section 9 for how CMake handles this.

Please note that putting code into a library is usually a way to separate common, reusable, and stable, parts (the libraries) from more specific, and often more actively developed parts (the main program). For the remainder of the labs in this course, it is probably overkill to make parts of the code into libraries even if it is generic.

10.2 Shared Libraries

Since most programs use large amounts of code from libraries, executable files can grow very large. Instead of linking library code into each executable that needs it the code can be loaded at runtime. The object files should then be in *shared libraries*. When linking programs with shared libraries, the files from the library are not actually linked into the executable. Instead a "pointer" is established from the program to the library.

In Unix shared library files are named *lib<name>.so[.x.y.z]* (*.so* for shared objects, *.x.y.z* is an optional version number). The linker uses the environment variable `LD_LIBRARY_PATH` as the search path for shared libraries. In Microsoft Windows shared libraries are known as DLL files (dynamically loadable libraries).

A16. (Advanced, optional) Create a shared library with the object files *editor.o* and *coding.o*. Link the executables using the shared library. Make sure they run correctly. Compare the sizes of the dynamically linked executables to the statically linked (there will not be a big difference, since the library files are small).

Use the command `ldd` (list dynamic dependencies) to inspect the linkage of your programs. Shared libraries are created by the linker, not the `ar` archiver. Use the `gcc` and `ld` manpages (and, if needed, other manpages) to explain the following sequence of operations:

```
g++ -fPIC -std=c++11 -c *.cc
g++ -shared -Wl,-soname,liblab1.so.1 -o liblab1.so.1.0 list.o coding.o
ln -s liblab1.so.1.0 liblab1.so.1
ln -s liblab1.so.1 liblab1.so
```

You then link with `-L. -llab1` as before. The linker merely checks that all referenced symbols are in the shared library. Before you execute the program, you must define `LD_LIBRARY_PATH` so it includes the current directory. You do this with the following command (on the command line):

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

11 Reflection

1. What is the difference between a declaration and a definition?
2. How does an include guard prevent multiple definitions?
3. How can you tell if an error comes from the compiler or the linker? Does a linker error mean that you have an error in your source code? How do you (typically) fix a linker error?
4. Do you have to make any changes to `MakefileWithDeps` to build your hello world program?
5. In `encode` and `decode`, the type `unsigned char` is used. Would your code work the same way if that type is changed to `char` or `signed char`?
6. In the coding problem, reading the file with `char ch; while (infile >> ch) ...` doesn't work. Why?
7. If your program crashes, how can you use the debugger to get a stack trace similar to that of `Exception.printStackTrace()` in Java?
8. What is the difference between a debugger breakpoint and a watchpoint?

2 Introduction to the Standard Library

Objective: to solve a moderately large problem using C++. Some parts of the standard library that haven't yet been introduced in the course will be used. They are introduced in section 3. Section 1 gives an overview of the problem, and section 2 contains a sequence of smaller assignments towards that goal. It is recommended to read the entire lab before you start writing code.

1 Spelling Correction

Most word processors can check the spelling of a document and suggest corrections to misspelled words. Often, a dictionary is used — words that aren't in the dictionary are considered to be misspelled. The suggested corrections are the words in the dictionary that are "similar" to the misspelled word.

Your task is to write a class `Dictionary` which can be used as in the following example:

```
void check_word(const string& word, const Dictionary& dict)
{
    if (dict.contains(word)) {
            cout << "Correct." << endl;
    } else {
            vector<string> suggestions = dict.get_suggestions(word);
            if (suggestions.empty()) {
                    cout << "Wrong, no suggestions." << endl;
            } else {
                    cout << "Wrong. Suggestions:" << endl;
                    for (const auto& w : suggestions) {
                            cout << "    " << w << endl;
                    }
            }
    }
}
int main() {
        Dictionary dict;
        string word;
        while (cin >> word) {
                transform(word.begin(), word.end(), word.begin(), ::tolower);
                check_word(word, dict);
        }
}
```

Examples:

```
expertise
Correct.
seperate
Wrong. Suggestions:
    separate
    desperate
    federate
    generate
    imperate
```

Notes:

- The function `contains` (section 2.2) must be efficient (fast).
- In `get_suggestions` you can spend time on finding good suggestions for corrections.
- It can be advantageous to "preprocess" the file which contains the dictionary (section 2.1).
- It is not certain that the data structures which you shall use are optimal (or even necessary), but you shall solve the assignments as they are given. You are encouraged to improve the program, but do that as a separate project.

The following shall be done in `get_suggestions`:

1. Search the dictionary and find candidates for corrections (section 2.3). To begin with, the words in the dictionary which have approximately the same number of letters (plus/minus one letter) as the misspelled word should be considered. Of these candidates, the words which contain at least half of the "trigrams" of the misspelled word should be kept. A trigram is three adjacent letters — for example, the word `summer` contains the trigrams `sum umm mme mer`.
2. Sort the candidate list so the "best" candidates are first in the list (section 2.4). The sort key is the cost to change the misspelled word to one of the candidate words.
3. Keep the first 5 candidates in the list (section 2.5).

Expressed in program code:

```
vector<string> Dictionary::get_suggestions(const string& word) const {
        vector<string> suggestions;
        add_trigram_suggestions(suggestions, word);
        rank_suggestions(suggestions, word);
        trim_suggestions(suggestions);
        return suggestions;
}
```

2 Assignments

2.1 Preprocess the Dictionary

A1. The file */usr/share/dict/words* contains a large number of words (one word per line). The file is UTF-8 encoded; ignore this and treat all characters as 8-bit. Write a program that reads the file and creates a new file *words.txt* in the current directory. Each line in the file shall contain a word, the number of trigrams in the word, and the trigrams.[12] The trigrams shall be sorted in alphabetical order; upper case letters shall be changed to lower case. Example:

```
...
hand 2 and han
handbag 5 and bag dba han ndb
handbook 6 and boo dbo han ndb ook
...
```

Copy the *Makefile* from the *lab1* directory, modify it to build the program, build, test.

2.2 Determine If a Word is Correct

A2. Implement the constructor and the function `contains` in `Dictionary`. The preprocessed list of words is in the file *words.txt*. The words shall be stored in an `unordered_set<string>`. Wait with the trigrams until assignment A4.

Modify the makefile (the main program shown in section 1 is in *spell.cc*), build, test.

2.3 Use Trigrams to Find Candidates

Instead of storing words in an `unordered_set<string>` you will now use a data structure with more information.

A3. The words together with their trigrams must be stored in the dictionary. Each word shall be stored in an object of the following class:

[12] Note that there are short words with zero trigrams.

```
class Word {
public:
        /* Creates a word w with the sorted trigrams t */
        Word(const std::string& w, const std::vector<std::string>& t);

        /* Returns the word */
        std::string get_word() const;

        /* Returns how many of the trigrams in t that are present
           in this word's trigram vector */
        unsigned int get_matches(const std::vector<std::string>& t) const;
};
```

Implement this class. The trigram vector given to the constructor is sorted in alphabetical order (see assignment A1). The function `get_matches` counts how many of the trigrams in the parameter vector that are present in the word's trigram vector.[13] You may assume that the trigrams in the parameter vector also are sorted in alphabetical order. Use this fact to write an efficient implementation of `get_matches`.

A4. The class `Dictionary` shall have a member variable that contains all words with their trigrams. It must be possible to quickly find words which have approximately the same length as the misspelled word. Therefore, the words shall be stored in the following array:

```
vector<Word> words[25]; // words[i] = the words with i letters,
                        // ignore words longer than 25 letters
```

Modify the `Dictionary` constructor so the `Word` objects are created and stored in the `words` member variable, and implement the function `add_trigram_suggestions`. Use a named constant (e.g. `constexpr int maxlen{25};`) instead of the literal 25.

2.4 Sort the Candidate List

After `add_trigram_suggestions` the suggestion list can contain a large number of candidate words. Some of the candidates are "better" than others. The list shall be sorted so the best candidates appear first. The sorting condition shall be the "edit distance" (also called "Levenshtein distance") from the misspelled word to the candidate word.

The cost $d(i,j)$ to change the i first characters in a word p to the j first characters in another word q can be computed with the following formula (i and j start from 1):

$$d(i,0) = i$$
$$d(0,j) = j$$
$$d(i,j) = \text{minimum of} \begin{cases} \text{if } p_i = q_j \text{ then } d(i-1,j-1) \text{ else } d(i-1,j-1)+1, \\ d(i-1,j)+1, \\ d(i,j-1)+1. \end{cases}$$

The minimum computation considers the cost for replacing a character, inserting a character and deleting a character. The cost to change p to q, that is the edit distance, is $d(p.length, q.length)$.

It is recommended that the computation of the edit distance is put in a separate function, so that it can be tested independently. A sketch of a unit test for the edit distance is included in the file *test_edit_distance.cc*.

[13] You don't have to consider multiple occurrences of the same trigram.

A5. Implement the functions `rank_suggestions`, and `edit_distance`. Do *not* write a recursive function, it would be very inefficient. Instead, let d be a matrix (with $d(i,0) = i$ and $d(0,j) = j$) and compute the elements in row order (dynamic programming). That is, write two nested loops

```
for("all indices i in p") {
  for("all indices j in q") {
    // Compute the lowest d(i,j) according to the given formula.
    // Note that you shall use the previously computed values of
    // d(i,j) for smaller values of i and j that are available
    // in the matrix d.
  }
}
```

Declare d with the type `int[maxlen+1][maxlen+1]`, where `maxlen` is the maximum word length (see comment in A4), to avoid problems with a dynamically allocated matrix.

2.5 Keep the Best Candidates

A6. Implement the function `trim_suggestions`. Make sure to handle the case where there are fewer than 5 candidates correctly.

3 More Information About the Assignments

- In the main program in *spell.cc*, the call to `transform` applies the function `tolower` to all the characters in a string (between `begin()` and `end()`), and stores the function result in the same place. `tolower` converts a character from upper case to lower case. The scope operator `::` is necessary to get the right version of the overloaded `tolower` function.
- To sort a vector `v`, call `std::sort(v.begin(), v.end())` (include `<algorithm>`).
- The standard library class `unordered_set` is in header `<unordered_set>`. An element is inserted in a set with the function `insert(element)`. The function `count(element)` returns the number of occurrences of an element (0 or 1).
- Here's one way to sort the suggested candidates in edit distance order (another way is to use a `map`):
  - Define a `vector` with elements that are pairs with the first component an `int`, the second a `string`: `vector<pair<int, string>>`.
  - Compute the edit distance for each candidate word, insert the distance and the word in the vector: `push_back(make_pair(dist, word))`.
  - Sort the vector (pairs have an `operator<` that first compares the first component, so the elements will be sorted according to increasing edit distance).
  - For a pair `p`, `p.first` is the first component, `p.second` the second component.
- Read more about computing edit distance on the web. You may also consider using the Damerau–Levenshtein distance.
- A vector can be resized to size `n` with `resize(n)`.

4 Reflection

1. The code
   ```
   std::string s;
   while(std::cin >> s){
     // do something with s
   }
   ```
   reads whitespace separated words from standard in until the stream is closed (typically by pressing CTRL-D). What does using the expression std::cin >> s as a bool value mean?

2. When reading the preprocessed file, did you use formatted input (i.e., operator>>) to read the file directly into variables? If not, how would you do that?

3. What type does the variable a have if declared as int a[10];

4. Why does the compiler issue warnings about comparing signed and unsigned values?

5. What happens if you call resize(5) on an empty std::vector<string>?

3 Debugging

Objective: to practice debugging a C++ program
Note: To make it easier to look at the program in the debugger, make sure you build your code with optimization turned off (`-O0`). See lab 1 for details.

1 A (broken) system for access control

A company has created a system for controlling access to its facilities. To open a door, an employee inserts a card in a card-reader. The system looks up the card number in a table. If the card number is found, the name of the employee is presented on a display and the door is unlocked. Otherwise the door remains locked.

The class `UserTable` is used to check who a certain card number belongs to. The access control system includes the following code:

```
UserTable t;
...
int cardNbr = sensor.getCardNbr();

User u = t.find(cardNbr);          // Find the owner of the card

if (u == UserTable::user_not_found) {
  display.showText("** access denied **");
} else {
  display.showText("Welcome, " + u.getName());
  door.unlock();
}
```

The interesting line is marked with a comment. Given a card number (an integer) the system looks up a user. The variables `sensor`, `display`, and `door` refer to objects outside the scope of this task. Can you understand the general operation of the above code?

The class `UserTable` is unfortunately broken. Your task is to find and correct the mistakes. Present your solution according to the table at the end of this section, together with a working test program.

2 Tasks

1. Study the source code in the files *User.h*, *User.cc*, *UserTable.h* och *UserTable.cc*. There is also a text file, *users.txt*, containing all users and their card numbers. Look at the file and see how it is organized. In the class `UserTable` the user database is read from the text file and stored in an array.

2. We will now debug the class `UserTable`. To test the class, it is impractical to try a lot of cards in the card reader. A better option is to create a simple test-program that creates a `UserTable` object and calls the function `find` in the same way the card-reader does. Then you can simply make up and test different card numbers yourself.

Create such a test program (including a `main` function) that

- creates a `UserTable` object,
- calls the function `find(int)` to find the user with card number 21330,
- calls the function `find(int)` to find the user with card number 21331 and
- calls the function `find(std::string)` to find the user named ”Jens Holmgren”.

What is the result of the program? What did you expect? (Compare with the file *users.txt*.)

3. The class `UserTable` has a member function for printing the contents of the table. Extend your test program with a call to this function and see what is printed.

Study the constructor for the class `UserTable`. Here, the user database is read from a file and stored in an array. Either the reading from the file is not working, or the read lines are not stored correctly in the array.

Add some lines to your test program so that a new (made up) user with card number 1234 is added, and then print the contents of the table. Run your test program. How many users are now in the table? How many did you expect?

4. Find and fix the mistake that you identified in the previous task. To get further clues you can ask yourself the following questions:

- The function `add` inserts a `User` object at the right position in the array. What positions are the objects put in? That is, what value does the variable `pos` get? (You can either use debug printouts, or run it in the debugger and use breakpoints. Remember to build without optimization when using the debugger.)
- Does the function `getNbrUsers` return the correct result? (Test.)

When you have corrected the mistake, check if searching for a card number works. Verify that your test program calls `find(int)` with a card number, and that the result is the same card number and the correct user name. Compare with the corresponding line in *users.txt*. Always compile with warnings enabled (the option `-Wall` to `g++`).

5. It has been reported that `find(std::string name)` does not work. The function is implemented with *linear search*: starting at the first position, go through the array until a person with the name `name` is found. Study the implementation of the function to find the bug.

When the function is corrected, the search for Jens Holmgren should work.

6. Verify that searching works. There is function `testFindNumber()` that can be used to test the member function `UserTable::find(int)`. Let your test program call that test function and verify that all users (card numbers) now can be found.

7. Check the test function. It has been reported that `testFindNumber()` performs the tests correctly, but that the tested `UserTable` object is left in an unusable state afterwards. Investigate what happens when `testFindNumber()` is called.

Note. The names in the table have been generated from the 100 most common swedish family names and male and female names[14] (excluding names containing `åäö`). They do not refer to actual persons.

[14] Source: SCB, `http://www.scb.se/namnstatistik/`.

Solution

To pass the task you have to show a working program, according to the above instructions. In addition you must be prepared to account for your debugging work. Record information about each bug so that you remember them when you present your solution.

Bug (what function?, what line?)	*Describe the problem.*	*How did you find the bug?*

4 Strings and Streams. Testing.

Objective: to practice using the standard library string and stream classes.

Read:

- Book: strings, streams, function templates, exceptions.

1 Unit testing

When writing code, testing is important and a common methodology in modern *agile* software development methodologies is *test driven design* or "test first". One major benefit of writing tests before writing the code is that this helps with understanding the problem and structuring the code, starting from the desired result and working ones way towards the solution. That is an application of the principle of *programming by intention* (or "wishful thinking") and helps designing functions with suitable parameters and return types.

In the strict formulation, you are only allowed to write new code if there is a unit test that doesn't pass. So to add new functionality you first write a unit test and make the test call the desired new function. In doing so, you specify both what the arguments to the function will be, and what type and value the function should return for the given arguments. This also includes defining any new types that you need to express the desired functionality. At this stage, your test will not compile, so now you add an empty function (with `return 0;`, `return false;` or what is suitable). Run the tests and make sure that they fail — if not, either the functionality is already supported or the test case is wrong. Then, implement the function to make the test case pass.

Another good testing principle is to write *unit tests*, that tests "the smallest testable unit" (e.g., functions, classes) in addition to large-scale tests that verifies the funciton of the entire system. Unit tests are valuable during development as they make it easy to check if additions to the system have affected the behaviour of the old (apparently unrelated) functionality. If the unit tests ran successfully before a modification to the code, they should also run successfully after it.

It is also often a good idea to write test programs that do not require manual user input or manually checking the output. For two small examples of such test programs, study the examples given in lab1: `test_coding` and `test_editor`.

A0. Read through the assignments of this lab (A1 – A5), and write test programs for each assignment. For instance, for the first assignment (A1), create a file (or string) where you have manually removed the HTML tags from the given HTML file and write a program that calls your tag removal function and compares its result with the manually created file (string). Start with smaller unit tests, e.g. for testing the removal of HTML tags and the replacement of special characters. For those, write small test cases that test just one thing ("unit of functionality").

For the second assignment (A2), you can translate the example with the numbers 0–35 into code, tetsting that your functions returns both the correct strings (`"CCPPC..."` and the corresponding prime number sequence).

For the third assignment (A3), there is a main program (that requires user input) given. You can base your test program on that. A good option here is to give the streams to use as parameters instead of hard-coding `std::cin` and `std::cout`. Then, one can use `std::stringstream` to automate both the input and checking the results. (See section 3.2 of this lab for info on stringstreams.)

For the last assignment (A5), remember to also test that your function throws exceptions correctly.

2 Class string

2.1 Introduction

In C, a string is a null-terminated array of characters. This representation is the cause of many errors: overwriting array bounds, trying to access arrays through uninitialized or incorrect pointers, and leaving dangling pointers after an array has been deallocated. The <cstring> library contains operations on C-style strings, such as copying and comparing strings.

C++ strings hide the physical representation of the sequence of characters. The exact implementation of the string class is not defined by the C++ standard.

The string identifier is not actually a class, but a type alias for a specialized template:

```
using string = std::basic_string<char>;
```

This means that string is a string containing characters of type char. There are other string specializations for strings containing "wide characters". We will ignore all "internationalization" issues and assume that all characters fit in one byte.

string::size_type is a type used for indexing in a string. string::npos ("no position") is a value indicating a position beyond the end of the string; it is returned by functions that search for characters when the characters aren't found.

2.2 Operations on Strings

The following class specification shows most of the operations on strings:

```
class string {
public:
    /*** construction ***/
    string();                // creates an empty string
    string(const string& s); // creates a copy, also has move constructor
    string(const char* cs);  // creates a string with the characters from cs
    string(size_type n, char ch); // creates a string with n copies of ch

    /*** information ***/
    size_type size();        // number of characters

    /*** character access ***/
    const char& operator[](size_type pos) const;
    char& operator[](size_type pos);

    /*** substrings */
    string substr(size_type start, size_type n = npos); // the substring starting
                             // at position start containing n characters

    /*** inserting, replacing, and removing ***/
    string& insert(size_type pos, const string& s); // inserts s at position pos
    string& append(const string& s);                // appends s at the end
    string& replace(size_type start, size_type n, const string& s); // replaces n
                             // characters starting at pos with s
    void erase(size_type start = 0, size_type n = npos); // removes n
                             // characters starting at pos

    /*** assignment and concatenation ***/
    string& operator=(const string& s);  // also move assignment
    string& operator=(const char* cs);
    string& operator=(char ch);
    string& operator+=(const string& s); // also const char* and char

    /*** access to C-style string representation ***/
    const char* c_str();
    /*** finding things (see below) ***/
}
```

- Note that there is no constructor `string(char)`. Use `string(1, char)` instead.
- The subscript functions (`operator[]`) do not check for a valid index. There are similar `at()` functions that do check, and that throw `out_of_range` if the index is not valid.
- The `substr()` member function takes a starting position as its first argument and the number of characters as the second argument. This is different from the `substring()` method in `java.lang.String`, where the second argument is the end position of the substring.
- There are overloads, for C-style strings or characters, of most of the functions.
- Strings have iterators like the standard library collections (e.g., `std::vector`).
- There is a bewildering variety of member functions for finding strings, C-style strings or characters. They all return `npos` if the search fails. The functions have the following signature (the `string` parameter may also be a C-style string or a character):

  ```
  size_type FIND_VARIANT(const string& s, size_type pos = 0) const;
  ```

  `s` is the string to search for, `pos` is the starting position. (The default value for `pos` is `npos`, not `0`, in the functions that search backwards).

  The "find variants" are `find` (find a string, forwards), `rfind` (find a string, backwards), `find_first_of` and `find_last_of` (find one of the characters in a string, forwards or backwards), `find_first_not_of` and `find_last_not_of` (find a character that is not one of the characters in a string, forwards or backwards). For example:

  ```
  string s = "accdcde";
  auto i1 = s.find("cd");              // i1 = 2 (s[2]=='c' && s[3]=='d')
  auto i2 = s.rfind("cd");             // i2 = 4 (s[4]=='c' && s[5]=='d')
  auto i3 = s.find_first_of("cd");     // i3 = 1 (s[1]=='c')
  auto i4 = s.find_last_of("cd");      // i4 = 5 (s[5]=='d')
  auto i5 = s.find_first_not_of("cd"); // i5 = 0 (s[0]!='c' && s[0]!='d')
  auto i6 = s.find_last_not_of("cd");  // i6 = 6 (s[6]!='c' && s[6]!='d')
  ```

There are global overloaded operator functions for concatenation (`operator+`) and for comparison (`operator==`, `operator<`, etc.). They all have the expected meaning. Note that you cannot use `+` to concatenate a string with a number, only with another string, C-style string or character (this is unlike Java). In the new standard, there are functions that convert strings to numbers and vice versa: `stod("123.45") => double`, `to_string(123) => "123"`.

A1. Write a class that reads a file and removes HTML tags and translates HTML-encoded special characters[15]. The class should be used like this:

```
int main() {
        TagRemover tr(std::cin); // read from cin
        tr.print(std::cout);     // print on cout
}
```

- All tags should be removed from the output. A tag starts with a `<` and ends with a `>`.
- You can assume that there are no nested tags.
- Tags may start on one line and end on another line.
- Line separators should be kept in the output.
- You don't have to handle all special characters , only `<`, `>`, ` `, and `&` (which represent the characters `<`, `>`, space, and `&`, respectively).
- Make sure that you use the standard library. Manual iteration and copying character by character is not a good solution.
- Assignments like this should be a good fit for regular expressions. Study and use the C++ `regex` library if you're interested[16].

Copy the makefile from one of the previous labs, modify it, build and test.

[15] In HTML, characters that are part of the HTML syntax have to be *escaped*. All such escape sequences start with an ampersand (`&`) and end with a semicolon.

[16] However, be warned that regular expressions grow quite complicated very quickly, and a more straight forward solution using find and replace or erase, as outlined above, is often preferable for both correctness and readability.

A2. The Sieve of Eratosthenes is an ancient method for finding all prime numbers less than some fixed number M. It starts by enumerating all numbers in the interval $[0, M]$ and assuming they are all primes. The first two numbers, 0 and 1 are marked, as they are not primes. The algorithm then starts with the number 2, marks all subsequent multiples of 2 as composites, finds the next prime, marks all multiples, ... When the initial sequence is exhausted, the numbers not marked as composites are the primes in $[0, M]$.

In this assignment you shall use a string for the enumeration. Initialize a string of appropriate length to `PPPPP...PPP`. The characters at positions that are not prime numbers should be changed to `C`.

Example with the numbers 0–35:

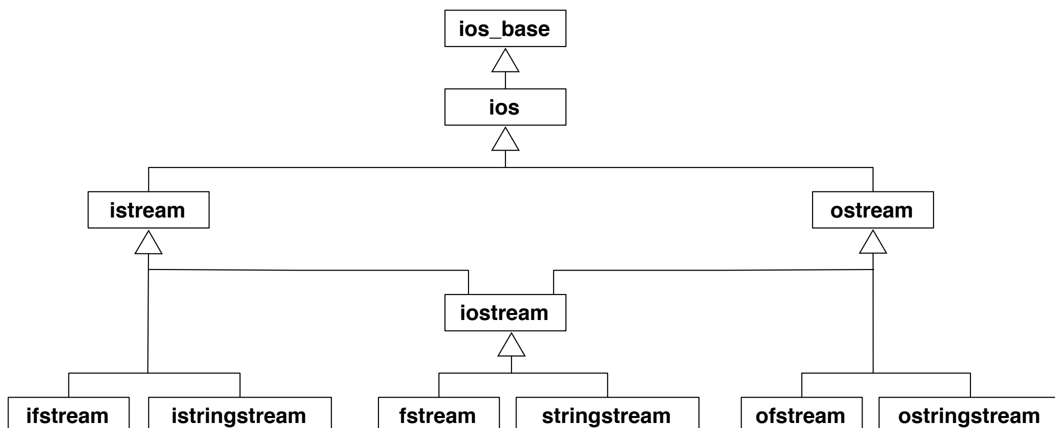
```
                                     1         2         3
                           012345678901234567890123456789012345
Initial:                   CCPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
Find 2, mark 4,6,8,...:    CCPPCPCPCPCPCPCPCPCPCPCPCPCPCPCPCPCP
Find 3, mark 6,9,12,...:   CCPPCPCPCCCPCPCCCPCPCCCPCPCCCPCPCCCP
Find 5, mark 10,15,20,...: CCPPCPCPCCCPCPCCCPCPCCCPCCCCCPCPCCCC
Find 7, mark 14,21,28,35:  CCPPCPCPCCCPCPCCCPCPCCCPCCCCCPCPCCCC
Find 11, mark 22,33:       CCPPCPCPCCCPCPCCCPCPCCCPCCCCCPCPCCCC
...
```

- Write a program that prints the prime numbers between 1 and 200 and also the largest prime that is less than 100,000. Use member functions in `std::string` for searching instead of manual loops.
- Do not expose the internal string representation in your interface. For instance, if you want to return a sequence of primes, use `std::vector<int>` and not a string.

3 The iostream Library

3.1 Input/Output of User-Defined Objects

In addition to the stream classes for input or output there are `iostream`'s that allow both reading and writing. The stream classes are organized in the following (simplified) hierarchy:



The classes `ios_base` and `ios` contain, among other things, information about the stream state. There are, for example, functions `bool good()` (the state is ok) and `bool eof()` (end-of-file has been reached). There is also a conversion operator `operator bool()` that returns true if the state is good, and a `bool operator!()` that returns true if the state is not good. We have used these operators with input files, writing for example `while (infile >> ch)` and `if (!infile)`.

To do formatted stream input and output of objects of user-defined classes, `operator>>` and `operator<<` must be overloaded.

A3. The files *date.h*, *date.cc*, and *date_test.cc* describe a simple date class. Implement the class and add operators for input and output of dates (operator>> and operator<<). Dates should be output in the form 2015-01-10. The input operator should accept dates in the same format. (You may consider dates written like 2015-1-10 and 2015 -001 - 10 as legal, if you wish.)

The input operator should set the stream state appropriately, for example is.setstate (ios_base::failbit) when a format error is encountered. Write your code so that the right hand operand of operator>> is not changed if the conversion fails.

3.2 String Streams

The string stream classes (istringstream and ostringstream) function as their "file" counterparts (ifstream and ofstream). The only difference is that characters are read from/written to a string instead of a file. In the following assignments you will use string streams to convert objects to and from a string representation (in the new standard, this can be performed with functions like to_string and stod, but only for numbers).

A4. In Java, the class Object defines a method toString() that is supposed to produce a "readable representation" of an object. This method can be overridden in subclasses.

Write a template function toString for the same purpose. Also write a test program. Example:

```
double d = 1.234;
Date today;
std::string sd = toString(d);
std::string st = toString(today);
```

You may assume that the argument object can be output with <<.

A5. Type casting in C++ can be performed with, for example, the static_cast operator. Casting from a string to a numeric value is not supported, since this involves executing code that converts a sequence of characters to a number.

Write a function template string_cast that can be used to cast a string to an object of another type. Examples of usage:

```
try {
        int i = string_cast<int>("123");
        double d = string_cast<double>("12.34");
        Date date = string_cast<Date>("2015-01-10");
} catch (std::invalid_argument& e) {
        cout << "Error: " << e.what() << endl;
}
```

You may assume that the argument object can be read from a stream with >>. The function should throw std::invalid_argument (defined in header <stdexcept>) if the string could not be converted.

4 Reflection

1. In your tests, how did you test the error handling (e.g., that a wrong `string_cast` actually throws?)

2. In `TagRemover`, why do you think the constructor takes an `istream` instead of just the filename?

3. In `TagRemover`, did you process the file line by line, or did you first read the entire file? What are the pros and cons of these two approaches?

4. How do you read the entire contents of an `std::istream` into a `std::string` without using a `for` or `while` loop?

5. In `TagRemover`, do you have duplicate code for translating the special characters? If so, how would you refactor your code to avoid duplicate code?

6. How do you check if an input or output operation on a stream (e.g., `operator>>` or `operator<<`) has failed?

7. How do you know if you have reached the end of an `istream`?

8. Does `string_cast<int>("123kalle")` return the value 123 or does it throw an exception? How do you implement each of those behaviours?

9. When calling the function template `toString`, the template type argument is not explicitly given in the call. For `string_cast`, on the other hand, you have to specify `string_cast<int>` or `string_cast<Date>`. What is the difference? When should explicit template arguments be given to function templates?