Laboratory Exercises, C++ Programming

General information:

- The course has four compulsory laboratory exercises.
- You shall work in groups of two people. Sign up for the labs at `sam.cs.lth.se/Labs`.
- The labs are mostly homework. Before each lab session, you must have done the assignments (A1, A2, ...) in the lab, written and tested the programs, and so on. Reasonable attempts at solutions count; the lab assistant is the judge of what's reasonable. Contact a teacher if you have problems solving the assignments.
- Smaller problems with the assignments, e.g., details that do not function correctly, can be solved with the help of the lab assistant during the lab session.
- Extra labs are organized only for students who cannot attend a lab because of illness. Notify the course coordinator if you fall ill, *before* the lab.

The labs are about:

1. Basic C++ programming, compiling, linking, debugging.
2. Introduction to the standard library.
3. Strings and streams.
4. Standard containers and algorithms.

Practical information:

- You will use many half-written "program skeletons" during the lab. You must download the necessary files from the course homepage before you start working on the lab assignments.
- The lab files are in separate directories *lab[1-4]* and are available in gzipped tar format. Download the tar file and unpack it like this:

  ```
  tar xzf lab1.tar.gz
  ```

  This creates a directory *lab1* in the current directory.

Good sources of information about C++:

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

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

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

Read:

- Book: basic C++, variables and types including pointers, 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 should be organized like this (a main program that uses a class `List`):

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

```
#ifndef LIST_H // include guard
#define LIST_H
// include necessary headers here

class List {
public:
    List();
    int size() const;
    ...
private:
    ...
};
#endif
```

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

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

List::List() { ... }
int List::size() const { ... }
...
```

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

```
#include "list.h"
#include <iostream>

int main() {
    List list;
    std::cout << "Size: " << list.size() << std::endl;
    ...
}
```

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

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 *ltest*, which is then executed:

```
g++ -c list.cc -std=c++11
g++ -c ltest.cc -std=c++11
g++ -o ltest ltest.o list.o
./ltest
```

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 ltest ltest.cc list.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
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 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 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. Some platforms define higher optimization levels, `-O3`, `-O4`, etc. You should not use these optimization levels unless you know very well what their implications are.

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 template classes (or any other complex template classes).

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*[1].
- 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 list program from section 1. The files *list.cc* and *ltest.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 ltest is older than ltest.o or list.o,
# then link ltest".
ltest: ltest.o list.o
        g++ -o ltest ltest.o list.o

# Rules to create the object files.
ltest.o: ltest.cc list.h
        g++ -c ltest.cc -std=c++11
list.o: list.cc list.h
        g++ -c list.cc -std=c++11
```

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 *list.h*, *list.cc*, and *ltest.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 ltest.o`. Run `make notarget`. Read the manual [2]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

[1] If no filename is given `make` looks for `Makefile` or `makefile` in the current directory.
[2] 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 $@`
`LDFLAGS` are options to the linker, such as `-Ldirectory` which makes the linker look for library files in *directory*. `LOADLIBES` and `LDLIBS`[3] are variables intended to contain libraries, such as `-llab1` or `mylibrary.a`. The variable `$^` 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

If invoked without arguments, 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 *ltest* 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: ltest
.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 ltest
```

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`[4] makes the preprocessor look at all `#include` directives and produce a file with the extension *.d* which contains the corresponding prerequisite. Suppose the file *ltest.cc* contains the following `#include` directive:

```
#include "list.h"
```

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

```
ltest.o : ltest.cc list.h
```

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

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

[4] 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 = ltest test_list

all: $(PROGS)

# Targets rely on implicit rules for compiling and linking
ltest: ltest.o list.o
test_list: test_list.o list.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 Writing small programs

A7. The class `List` describes a linked list of integers.[5] The numbers are stored in nodes. A node has a pointer to the next node (`nullptr` in the last node).

In this assignment you shall use raw pointers and manual memory allocation and deletion. This is common in "library classes" which must be very efficient and are assumed to be error free. In an application you would use one of the safe pointer types that were introduced in the new standard.

```
class List {
public:
    /* creates an empty list */
    List();

    /* destroys this list */
    ~List();

    /* returns true if d is in the list */
    bool exists(int d) const;

    /* returns the size of the list */
    int size() const;

    /* returns true if the list is empty */
    bool empty() const;

    /* inserts d into this list as the first element */
    void insertFirst(int d);

    /* removes the first element less than/equal to/greater than d,
       depending on the value of df. Does nothing if there is no value
       to remove. The enum values are accessed with List::DeleteFlag::LESS,
       ..., outside the class */
    enum class DeleteFlag { LESS, EQUAL, GREATER };
    void remove(int d, DeleteFlag df = DeleteFlag::EQUAL);

    /* prints the contents of this list */
    void print() const;

    /* forbid copying of lists */
    List(const List&) = delete;
    List& operator=(const List&) = delete;
private:
    /* a list node */
    struct Node {
        int value;  // the node value
        Node* next; // pointer to the next node, nullptr in the last node
        Node(int v, Node* n) : value(v), next(n) {}
    };

    Node* first; // pointer to the first node
};
```

`Node` is a `struct`, i.e., a class where the members are public by default. This is not dangerous, since `Node` is private to the class.

The copy constructor and assignment operator are deleted, so lists cannot be copied.

Implement the member functions in *list.cc*, build and test. Execution errors like "segmentation fault" are addressing errors. Read section 6 about finding execution errors.

[5] In practice, you would never write your own list class. There are several alternatives in the standard library.

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[6], 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.

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. Use the Unix `diff` command.

Note: the programs will work also for files that are UTF-8 encoded. In UTF-8 characters outside the "ASCII range" are encoded in two bytes, and the `encode` and `decode` functions will be called twice for each such character.

6 Finding Errors

With the GNU debugger, `gdb`[7], you can control a running program (step through the program, set breakpoints, inspect variable values, etc.). Debug information is inserted into the executable program when you compile and link 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 all optimizations that do not conflict with debugging.

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

```
gdb ./program
```

Some useful commands:

`help [command]`	Get help about gdb commands.
`run [args...]`	Run the program (with arguments).
`start [args...]`	Set a temporary breakpoint at `main` and run the program

[6] 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.

[7] 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. A good introduction can be found on the llvm project website: `https://lldb.llvm.org/use/tutorial.html`

`continue`	Continue execution.
`next`	Step to the next line *over* function calls.
`step`	Step to the next line *into* function calls.
`finish`	Continue until just after the current function returns.
`where`	Print the call stack.
`list [nbr]`	List 10 lines around the current line or around line `nbr` (the following lines if repeated).
`break func`	Set a breakpoint on the first line of a function `func`.
`break nbr`	Set a breakpoint at line `nbr` in the current file.
`print expr`	Print the value of the expression `expr`.
`watch var`	Set a watchpoint, i.e., watch all modifications of a variable. Can be very slow but can be the best solution to find some bugs.

`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 windows.

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

6.1 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++, errors of this kind are not caught, instead they 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).

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, 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.

A10. Go to the directory *buggy_programs* and build the programs using *Makefile*. Then run each program under valgrind and see what problems, if any, it finds. Make sure you understand the messages printed by valgrind.

A11. Run *ltest* under control of `valgrind`. The leak summary should show that 0 bytes have been lost. If it doesn't, the `List` destructor probably contains an error.

Introduce an addressing error in one of the `List` member functions (e.g., remove the check for end-of-list in `exists`). Run the program, first as usual, then under valgrind.

Introduce an error in the `List` destructor (e.g., delete all nodes but one). Run the program, first as usual, then under valgrind. Remove the errors that you introduced before continuing.

6.2 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.

A12. 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).

A13. 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.

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`).

7 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 are, assuming you are standing in the project root directory, in this case *cmake-example*, are:

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

A14. You should now separate your programs into a library named liblab1 (containing the files `coding.o` and `list.o`) and the main programs. Study the *cmake-example* project and then write *CMakeLists.txt* files for lab1 and use that to build the programs. The files that should go into *liblab* 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.

A15. 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.

8 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. Object file libraries may contain thousands of files and cannot reasonably be shipped as separate files. Instead, the files are collected into library files that are directly usable by the linker.

8.1 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.[8] 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
```

A16. (optional) Collect the object files *list.o* and *coding.o* in a library *liblab1.a*. Change the makefile so the programs (*ltest* , *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 list.o
    ar crv liblab1.a coding.o list.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 7 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.

[8] 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
```

8.2 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).

A17. (Advanced, optional) Create a shared library with the object files *list.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
```

9 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?

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.

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 which 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.[9] 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

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:

[9] 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.[10] 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 `words`, implement the function `add_trigram_suggestions`. Use a constant instead of the number 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 $d(p.length, q.length)$.

A5. Implement the function `rank_suggestions`. 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). Declare d with the type `int[26][26]` to avoid problems with a dynamically allocated matrix.

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

2.5 Keep the Best Candidates

A6. Implement the function `trim_suggestions`.

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 it mean when the expression `std::cin >> s` is used as a `bool` value?
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?

3 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.

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[11]. 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 `<`, `>`, ` `, `&` (corresponding to `< > space &`).
- 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[12].

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

[11] 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.

[12] 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:

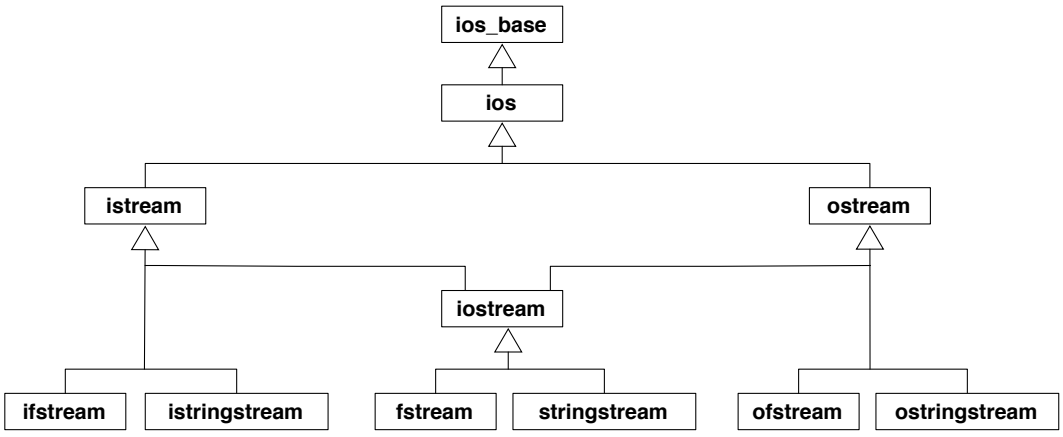
```
                                1         2         3
                       01234567890123456789012345678901234
Initial:               CCPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
Find 2, mark 4,6,8,...:CCPPCPCPCPCPCPCPCPCPCPCPCPCPCPCPCP
Find 3, mark 6,9,12,...CCPPCPCPCCCPCPCCCPCPCCCPCPCCCPCPCCCP
Find 5, mark 10,15,20,.CCPPCPCPCCCPCPCCCPCPCCCPCCCCCPCPCCCC
Find 7, mark 14,21,28,3CCPPCPCPCCCPCPCCCPCPCCCPCCCCCPCPCCCC
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.
- 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>("123abc")` return the value 123 or does it should 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?

4 Standard Containers and Algorithms

Objective: to practice using the standard library container classes and algorithms, with emphasis on efficiency. To learn more about operator overloading and iterators.

Read:

- Book: containers and algorithms. Operator overloading, iterators.

1 Name Servers and the Container Classes

On the web, computers are identified by IP addresses (32- or 128-bit numbers). Humans identify computers by symbolic names. A name server is a component in the Domain Name System (DNS) that translates a symbolic name to the corresponding IP address. The DNS is a very large distributed database that contains billions (or at least many millions) of IP addresses and that receives billions of lookup requests every day. Furthermore, the database is continuously updated.

In this lab, you will implement a local name server in C++. With "local" we mean that the name server does not communicate with other name servers; it can only perform translations using its own database. The goal is to develop a time-efficient name server. You shall implement four versions of the name server, using different container classes. All four classes implement the interface `NameServerInterface`:

```
using HostName = std::string;
using IPAddress = unsigned int;
const IPAddress NON_EXISTING_ADDRESS = 0;

class NameServerInterface {
public:
    virtual ~NameServerInterface() = default;
    virtual void insert(const HostName&, const IPAddress&) = 0;
    virtual bool remove(const HostName&) = 0;
    virtual IPAddress lookup(const HostName&) const = 0;
};
```

`insert()` inserts a name/address pair into the database, without checking if the name already exists. `remove()` removes a name/address pair and returns `true` if the name exists; it does nothing and returns `false` if the name doesn't exist. `lookup()` returns the IP address for a specified name, or `NON_EXISTING_ADDRESS` if the name doesn't exist.

You shall use library containers and algorithms as much as possible. This means, for example, that you are not allowed to use any `for` or `while` statements in your solutions. (There is one exception: you may use a `for` or `while` statement in the hash function, see assignment A1d.)

A1. The definition of the class `NameServerInterface` is in the file *nameserverinterface.h*.

a) Implement a class `VNS` (vector name server) that uses an unsorted `vector` to store the name/address pairs. Use the `find_if` algorithm to search for a host name. The third parameter to the algorithm should be a lambda.

   This implementation is clearly inefficient. A sorted vector would be better alternative, but maybe not for a name server with many insertions and deletions.

b) Implement a class `MNS` (map name server) that uses a `map` to store the name/address pairs. The average search time in this implementation will be considerably better than that for the vector implementation.

c) Implement a class `UMNS` (unordered map name server) that uses an `unordered_map` to store the name/address pairs.

d) An unordered map is implemented using a hash table. You shall compare this implementation with your own implementation of a hash table. Implement a class `HNS` (hash name server) that uses a hash table — a `vector` of `vector`'s — to store the name/address pairs.

The hash table implementation is open for experimentation: you must select an appropriate size for the hash table (given as an argument to the constructor) and a suitable hash function.[13,14] You should be able to obtain approximately the same search times as for the unordered map implementation.

Copy the makefile from one of the previous labs, modify it. Use the program *nstest.cc* to verify that the insert/remove/lookup functions work correctly. Then, use the program *nstime.cc* to measure and print the search times for the four different implementations, using the file *nameserverdata.txt* as input (the file contains 284,353 name/address pairs[15]).

A2. Examples of average search times in milliseconds for a name server with 284,353 names are in the following table.

	284,353	1,000,000
vector	0,332	
map	0.00098	
unordered	0.00039	
hash	0.00026	

Search the Internet for information about efficiency of searching in different data structures, or use your knowledge from the algorithms and data structures course, and fill in the blanks in the table. Write a similar table for your own implementation.

2 Bitsets, Subscripting, and Iterators

2.1 Bitsets

To manipulate individual bits in a word, C++ provides the bitwise operators `&` (and), `|` (or), `^` (exclusive or), and the shift operators << (shift left) and >> (shift right). The standard class `bitset<N>` generalizes this notion and provides operations on sets of `N` bits indexed from `0` through `N-1`. `N` may be arbitrary large, so the bitset may occupy many words.

For historical reasons, `bitset` doesn't provide any iterators. We will develop a simplified version of the `bitset` class where all the bits fit in one word, and extend the class with iterators so it becomes possible to use the standard library algorithms with the class. Our goal is to provide enough functionality to make the following program work correctly:

```
int main() {
    // Define an empty bitset, set every third bit, print
    Bitset bs;
    for (size_t i = 0; i < bs.size(); i += 3) {
        bs[i] = true;
    }
```

[13] Note that a good hash function should take all (or at least many) of the characters of a string into account and that `"abc"` and `"cba"` should have different hash codes. For instance, a hash function that merely adds the first and last characters of a string is not acceptable.

[14] `std::hash<string>` is a good hash function.

[15] The computer names are from `http://httparchive.org`. The IP addresses are running numbers.

```
    copy(bs.begin(), bs.end(), ostream_iterator<bool>(cout));
    cout << endl;

    // Find the first five bits that are set, complement them, print
    size_t cleared = 0;
    auto it = bs.begin();
    while (it != bs.end() && cleared != 5) {
        it = find(it, bs.end(), true);
        if (it != bs.end()) {
            *it = !*it;
            ++cleared;
            ++it;
        }
    }
    copy(bs.begin(), bs.end(), ostream_iterator<bool>(cout));
    cout << endl;

    // Count the number of set bits, print
    cout << "Number of set bits: " << count(bs.begin(), bs.end(), true) << endl;
}
```

The output from the program should be (on a 64-bit computer):

```
1001001001001001001001001001001001001001001001001001001001001001
0000000000000001001001001001001001001001001001001001001001001001
Number of set bits: 17
```

An iterator for bitsets has to support both reading and writing, so it must be a model of `ForwardIterator`. Actually, it is not difficult to make it a model of `RandomAccessIterator`, but this would mean that we had to supply more functions.

The solution will be developed in several steps:

- Implement the "bit fiddling" methods necessary to set, clear, and test an individual bit in a word (this we have done for you).
- Implement `operator[]`. This is rather difficult.
- Implement the bitset iterator. This turns out to be relatively simple.

A3. The files *simplebitset.h* and *simplebitset.cc* contain the implementation of a simple version of the bitset class, with `get` and `set` functions instead of a subscripting operator. Study the class and convince yourself that you understand how the bits are manipulated. Copy the makefile from one of the previous labs, modify it. Use the program in *simplebitsettest.cc* to check the function of the class.

2.2 Subscripting

Subscripting is handled by `operator[]`. In order to allow subscripting to be used on the left hand side of an assignment, a reference must be returned (e.g., like `int& operator[](size_type)` in a `vector<int>` class). For a bitset, a reference to an individual bit in a word is needed, but there are no "pointers to bits" in C++. We must write a "proxy class", `BitReference`, to represent the reference. This class contains a pointer to the word that contains the bits and an integer that is the position of the bit in the word.

Outline of the class (`BitsetStorage` is the type of the word that contains the bits):

```
class BitReference {
public:
    BitReference(Bitset::BitStorage* pb, std::size_t p) : p_bits(pb), pos(p) {}
    // ... operations will be added later
private:
    Bitset::BitStorage* p_bits; // pointer to the word containing bits
    std::size_t pos;            // position of the bit in the word
};
```

The `Bitset` class looks like this:

```
class Bitset {
    friend class BitReference;
public:
    ...
    bool operator[](std::size_t pos) const;
    BitReference operator[](std::size_t pos);
    ...
private:
    using BitStorage = unsigned long;
    BitStorage bits;
    static const std::size_t
        BPW = std::numeric_limits<BitStorage>::digits; // "Bits per word"
};
```

The const version of `operator[]` is easy: it is identical to the `get` function in `SimpleBitset`. The non-const version should be defined as follows:

```
BitReference operator[](std::size_t pos) {
    return BitReference(&bits, pos);
}
```

The actual bit fiddling is performed in the `BitReference` class. In order to see what we need to implement in this class we study the results of expressions involving `operator[]`:

```
bs[3] = true; // bs.operator[](3) = true; =>
              // BitReference(&bs.bits,3) = true; =>
              // BitReference(&bs.bits,3).operator=(true);
```

From this follows that the following operator function must be implemented in `BitReference`:

```
BitReference& operator=(bool b); // for bs[i] = b
```

This function should set the bit referenced by the `BitReference` object to the value of `b` (just like the `set` function in the `SimpleBitset` class). There are more ways of using `operator[]`:

```
bool b = bs[6]; // b = bs.operator[](6); =>
                // b = BitReference(&bs.bits,6); =>
                // b = BitReference(&bs.bits,6).operator bool();
```

A conversion function must be implemented:

```
operator bool() const; // for b = bs[i]
```

The last use case:

```
bs[3] = bs[6];  // bs.operator[](3) = bs.operator[](6); =>
                // BitReference(&bs.bits,3) = BitReference(&bs.bits,6); =>
                // BitReference(&bs.bits,3).operator=(BitReference(&bs.bits,6));
```

Another assignment operator must be implemented:

```
BitReference& operator=(const BitReference& rhs); // for bs[i] = bs[j]
```

A4. Use the files *bitset.h*, *bitset.cc*, *bitreference.h*, *bitreference.cc*, and *bitsettest1.cc*. Implement the functions in *bitreference.cc* and test.

2.3 Iterators

From one of the OH slides: "An iterator "points" to a value. All iterators are `DefaultConstruct\-ible` and `Assignable` and support `++it` and `it++`." A `ForwardIterator` should additionally be `EqualityComparable` and support `*it` for both reading and writing via the iterator.

The most important requirement is that an iterator should point to a value. A `BitsetIterator` should point to a Boolean value, and we already have something that does this: the class `BitReference`. The additional requirements (`++`, equality test, and `*`) are easy to implement in the iterator class. It will look like this:[16]

```
class BitsetIterator : public std::iterator<std::forward_iterator_tag, bool> {
public:
    BitsetIterator(Bitset::BitsetStorage& pb, size_t p) : ref(pb, p) {}
    bool operator!=(const BitsetIterator& bsi) const { ... }
    BitsetIterator& operator++() { ... }
    BitReference operator*() { ... }
    BitsetIterator& operator=(const BitsetIterator& rhs) {
        ref.p_bits = rhs.ref.p_bits;
        ref.pos = rhs.ref.pos;
        return *this;
    }
private:
    BitReference ref;
};
```

The base class `iterator` contains some type aliases, for example `value_type`, and the iterator tag `forward_iterator_tag`, which informs the compiler that the iterator is a forward iterator. The assignment operator is redefined so it makes a memberwise copy of the `BitReference` object, rather than using the assignment operator in `BitReference` which sets a bit in the bitset.

A5. Uncomment the lines in *bitreference.h*, *bitset.h* and *bitset.cc* that have to do with iterators, implement the `begin()` and `end()` functions. Implement the member functions in *bitsetiterator.h*. Use the program *bitsettest2.cc* to test your classes.

[16] This class only contains the constructs that are necessary for the test program. For example, we have not implemented postfix ++, -> or comparison with ==.

3 Reflection

1. Creating (unnecessary) copies of objecs is a big source of overhead. Did you (accidentaly) create copies in your name server? What impact did it have on execution times? How can you avoid copying objects? When do you need to create copies?

2. It was stated that `operator[]` must return a reference in order to make assignments like `x[0] = 1` possible. Why is that?

3. In `BitReference`, is `operator++()` pre-increment (i.e., `++x`) or post-increment (i.e., `x++`). How does the compiler distinguish the two unary ++ operators?

4. Do you have any duplicate code in the implementation of `BitReference::operator=(bool)` and `BitReference::operator=(const BitReference&)`? If so, can you implement one function by calling the other one? Can you do it in a simple and elegant way?

5. Some functions return by value and some return a reference. How do you determine which is best? Are there situations when only one of them is correct?