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, compiling, linking.
2. Introduction to the standard library.
3. Debugging
4. Strings and streams.

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 available in gzipped tar format. The archives can be unpacked from the command line like this:

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

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

Good sources of information about C++:

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

Programmering i C++, godkända laborationsuppgifter

Skriv ditt namn och din namnteckning nedan:

Namn: .....

Namnteckning: .....

Godkänd laboration	Datum	Laborationsledarens namnteckning
1		
2		
3		
4		

1 Basic C++ Programming, Compiling, Linking

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 `#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*.

In C++ (and in C) declarations are collected in header files with the extension *.h*. To distinguish C++ headers from C headers other extensions are sometimes used, such as *.hpp* or *.hh*. 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 editor class in a file *editor.h:*

```
#ifndef EDITOR_H // include guard
#define EDITOR_H
// 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.cpp*

```
#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.cpp*:

```
#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 template functions).

The g++ command line looks like this:

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

The *.cpp* 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.cpp
g++ -std=c++11 -c test_editor.cpp
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 *.cpp*.

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

Short explanations (you can read more about these and other options on the gcc and g++ manuals):

`-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`	treat "serious" warnings as errors
`-Wold-style-cast`	warn for old-style casts, e.g., `(int)` instead of `static_cast<int>`
`-std=c++11`	follow the new C++ standard (use `-std=c++0x` on early versions of `g++`)
`-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 erroneous or at least questionable.

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*.
- 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 *editor.cpp* and *test_editor.cpp* 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.cpp editor.h
        g++ -std=c++11 -c test_editor.cpp

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

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.

A2. 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 [1]and try other options.

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`. `LOADLIBES` and `LDLIBS`[2] are variables intended to contain libraries, such as `-llab1`. `$^` expands to all prerequisites. So this is a good rule, except for one thing: it uses `$(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

Make by default creates 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 *all* is created 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*, like this:

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

[1] See section 9 (*How to run make*) of the manual. The options are summarized in section 9.7.

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

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`[3] 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.cpp editor.h
```

The *.d* files are included in the makefile, so it functions the same way as if we had written the rules ourselves.

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++
```

[3] 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.

```
#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)
```

A3. The better makefile is in the file *BetterMakefile*. 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 the makefile to build your "Hello world!" program.

5 Writing small programs

A4. 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.

A5. 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.cpp`.

Then write a program, `encode`, that reads a text file[4], 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 of to use is `int main(int argc, const char** argv)`. See `print_argv.cpp` 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`, 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 (no `-O2` 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
`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.

A6. (Optional) Run the test programs under control of `gdb`, try the commands.

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

7 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.

7.1 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 object files from 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.[5] The library name (without `lib` and `.a`) is given after `-l`.

A7. Collect the object files *list.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`.

7.2 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).

A8. (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:

[5] 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
```

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