All elemental steps that will get you started for your new life as a computer science programmer.
Week C – Build environments
This week you will get to know your build environment a little better. So far, you only used the Makefiles that were provided with the code.

First, we are going to have a closer look at these Makefiles.

Then we learn how we can automatically create Makefiles with CMake.

There’s also a short introduction to the gdb (a C++ debugger).

At last, we will see how we can integrate all these things into an IDE, our weapon of choice will be Eclipse and Qt Creator.
Make – What did we use the last weeks?!

May just contain a few commands
May define different “targets”
Targets may depend on each other
May contain Macros
...not only good for C++

Use tutorials! (see below)

Make – What did we use the last weeks?!
If you need more information on Makefiles, I recommend to read one or two good tutorial, e.g.:

Chris Serson’s Makefile Tutorial


Makefiles are really great to make simple projects even more simple. They may just start very small and contain a single command. The key idea is that if you need a chain of commands to build your software, e.g. you first need to compile several files and then link them together in a last step. You can express these dependencies as targets and chain multiple targets to create new dependencies. I.e., make knows which parts of your program have to be recompiled and linked if certain source files changes.
This is a small example of a makefile (it builds the code from last week).

Let's have a closer look

```makefile
CC=g++
CFLAGS=-I. -g
EXE=assignment_03

$(EXE): main.o Quad.o ObjExporter.o Estate.o Tree.o
       Conifer.o House.o Storey.o Roof.o
    $(CC) $(CFLAGS) -o $@ $^$

main.o: main.cpp Tree.h Conifer.h Estate.h House.h Storey.h
        Roof.h Vector3D.h ObjExporter.h
    $(CC) $(CFLAGS) -c $<

Quad.o: Quad.cpp Quad.h Vector3D.h
        $(CC) $(CFLAGS) -c $<
        [...]$

Roof.o: Roof.cpp Roof.h Storey.o
        $(CC) $(CFLAGS) -c $<$

clean:
    rm -f *.o *~ $(EXE)
```
MACROS
The Makefile typically starts with a list of MACROS, which are simple placeholders. Usually, you define which compiler you use, which name your project has or which compiler flags you use. That makes Makefiles reusable.

TARGETS & DEPENDENCIES
The main part of the Makefile is a list of targets, for each target a chain of commands is called. In this example, the targets assignment_03, main.o, Quad.o, Roof.o and clean are defined. Targets may depend on each other, e.g. the target assignment_03 links together compiled files that have been created by other targets (e.g., Quad.o).
There are also some shortcuts which make it easier to write short Makefiles. The line

```
%.o: %.cpp
  $(CC) -c $^  
```

Automatically compiles all object files that are needed. The drawback is that Make does not know which headers these files depend on, so if you alter a header file and run make, your changes might not be incorporated.

You can use Makefiles for other applications not related to C++ programming too, e.g. for building .pdf files from LaTeX sources.
CMake – Why it’s better

“Meta-Make”
Simple Scripting language
Works on multiple platforms with multiple build systems
  - May also create VS Solutions, Eclipse projects
Can create Installer files (.deb, .dmg, .msi)

Use tutorials! (see below)

CMake- why it’s better
Here you find tutorials for CMake:

http://www.cmake.org/Wiki/CMake

It’s a good starting point for CMake basics and code snippets.

CMake is a good solution (among many others) if you want to work with bigger projects and if you would like to share your code with other people. CMake is some kind of “Meta-Make”: you use a more human-readable script-language to define your project: which files have to be compiled, which include directories are used which external libraries have to be found and linked. For many external libraries, there exist scripts that search for local installations and tell CMake where they are located on your disk.

One good thing is that you can also use CMake to create Visual Studio project files. So you can work on one project that runs on machines with Linux/GCC as well as Windows/VisualStudio.
The “basic” usage of cmake looks like this. You enter a directory that contains a CMakeLists.txt file. You type `cmake .` and Makefiles are automatically created. You may now use `make` to compile your code. The `Makefile` also checks if you alter the `CMakeLists.txt` file, so CMake is automatically invoked when you alter your project settings.

**CMake – Usage**

- Create `CMakeLists.txt` file
- Run `cmake .` in this directory (or use GUI to create Visual Studio project)
- Or run `cmake ..` in separate build directory
- Run `make` to compile
- Delete `CMakeCache.txt` to clean the cache (may be necessary when you move project to different location) and run `cmake .` again
- Once `Makefile` is created, `make` also checks for updates in `CMakeLists.txt`
The CMakeLists.txt file from last week looks much cleaner than the Makefile a few slides before. You state the project name, set the debugger flags, write a list of source files and tell cmake to build an executable files from all of them.

```cmake
PROJECT( assignment_03 )
SET(CMAKE_CXX_FLAGS "-g")
SET(CMAKE_CXX_FLAGS_DEBUG)
SET( assignment_03_SRCS
    main.cpp
    Quad.cpp
    ObjExporter.cpp
    Estate.cpp Tree.cpp
    Conifer.cpp
    House.cpp
    Storey.cpp
    Roof.cpp )
ADD_EXECUTABLE( assignment_03 ${assignment_03_SRCS} )
```
There are a lot of alternatives to CMake, if you do not like it, I encourage you to try other tools (you might discover that they are even worse). In my opinion, CMake is great because it’s not just cross-platform, it’s cross-build environments.

http://qt-project.org/
http://www.gnu.org/software/automake/
http://www.scons.org/
**gdb – debugging for pros**

Still debugging with text output?

**gdb** let’s you look at your app at run-time

- Variables
- Call-stack
- Breakpoints & step-by-step

Requires debug symbols “-g”

Use tutorials (see below!)

---

**Gdb – debugging for pros**
A very simple but useful tutorial can be found here:


When your program crashes or simply misbehaves, using a debugger is a very good idea. One thing you have to make sure is that your program is compiled with the appropriate debug symbols (the option –g should be passed to the compiler).

All you have to is to start your program with **gdb**.
Call `gdb` with your binary as an argument. Type `run`.

When your assertion is violated, you can inspect your call stack and the variables that are located on every level. In this example the call stack is displayed, the user goes up three levels (3x"up") and inspects the value of `m_size->z` in the `Conifer::getQuads()` context.
In this example, the program is trapped in an infinite loop. The user presses CTRL+C to interrupt execution and displays both tessellation and phi. One can see that tessellation has a strange value, while theta is still -180. From these cues one can often deduce what went wrong in the program. It is also possible to track the change of certain values over time, to set distinct breakpoints in the code, etc. But that is very tedious on the console, I recommend that you do more complicated stuff in a graphical environment.

So let’s take a look at eclipse!
The following slides show how to start an Eclipse project from scratch. Before we begin, you should install all relevant plugins for eclipse. The “vanilla” eclipse only contains the needed plugins for JAVA development. You can install the C++ component using the following manual:

http://max.berger.name/howto/cdt/ar01s04.jsp#installingcdt

or (in German):

http://wiki.ubuntuusers.de/Eclipse#CDT

Please follow it step by step, it’s not that hard.

Please note that all plugin files are stored in your GITZ home directory, so make sure you have at least 26 MB of free space there. You can check with this command:

```
fs lq
```
Let’s get started with eclipse: Create a new Make C++ project.

Make sure that the “C++” mode is active (upper right part of your screen, otherwise, you might still be in “JAVA” mode).
Select the location where you unzipped the files, give a name to your project.
Create Make targets by right-clicking on the Add Make target button.
When you want to start debugging, change to the “Debug” mode.
Your screen will change to a debug setting, where you can see the call stack, the code, local variables and more stuff.

Using the buttons, you can go through the code step-by-step. You can also set breakpoints very easily. Play around with the debugger.
Eclipse – fun facts

There are several ways to work with CMake and eclipse

- Create Makefiles and import C++ Make Project in eclipse
- Create Eclipse project with CMake (>2.6)
  http://www.cmake.org/Wiki/Eclipse_CDT4_Generator

I recommend the first one

When you created your project with CMake, there are actually several ways to work with eclipse.

A very recent one is to create Eclipse project files with CMake. While this sounds tempting, I prefer to treat your project as a Makefile project (i.e. create Makefiles once manually and import Makefile project to eclipse).

This way you can edit your CMakeLists.txt files directly and invoke make to automatically run cmake.
Eclipse – Alternatives

- MS Visual Studio (Windows)
- QtCreator
- KDevelop
- Bloodshed Dev C++
- Xcode (MacOS)

Why eclipse?
- Multi-platform
- Works with Make projects
QtCreator

Nice, light-weight IDE
Also supports cross-platforms (qmake)
WYSIWUG Editor for GUIs

More on that later ...