Build Systems, Version Control, Integrated Development Environment

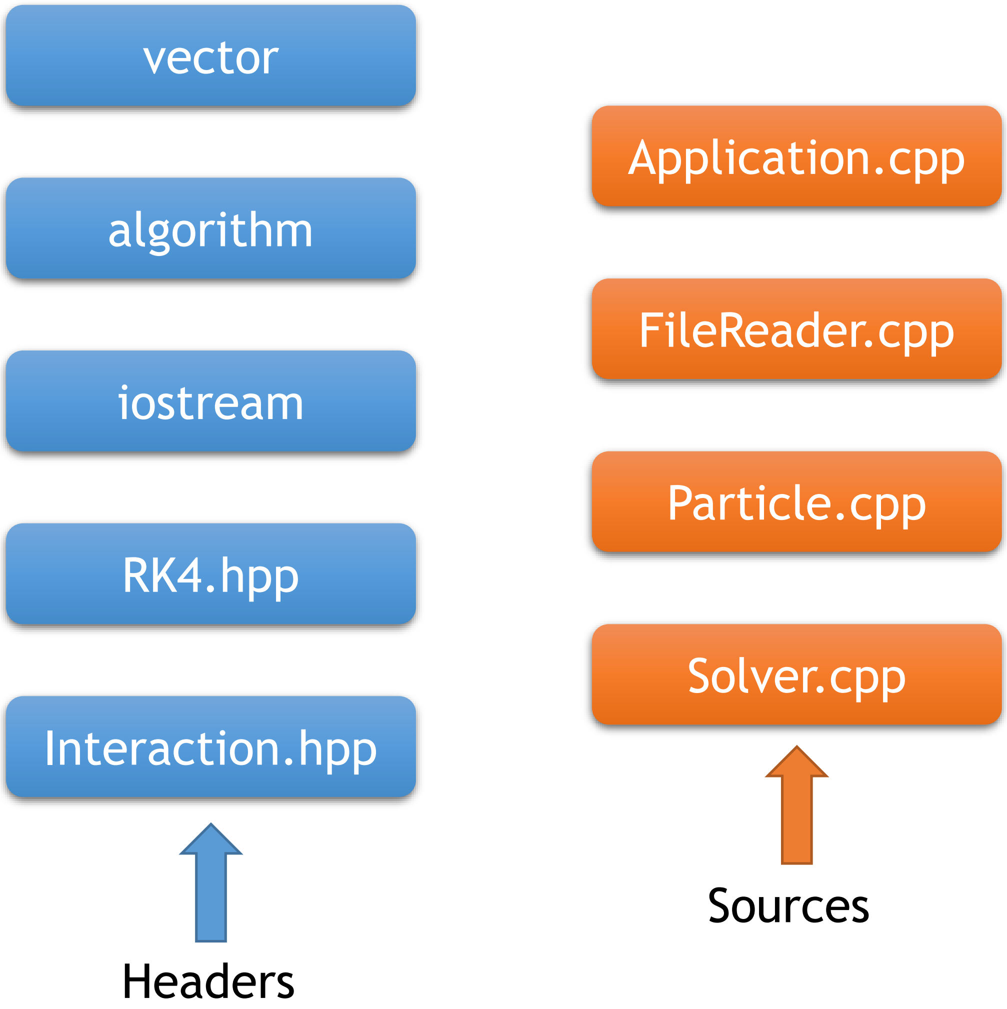
Lectures on Modern Scientific Programming
Wigner RCP
23-25 November 2015

D. Berényi – M. F. Nagy-Egri

Build System

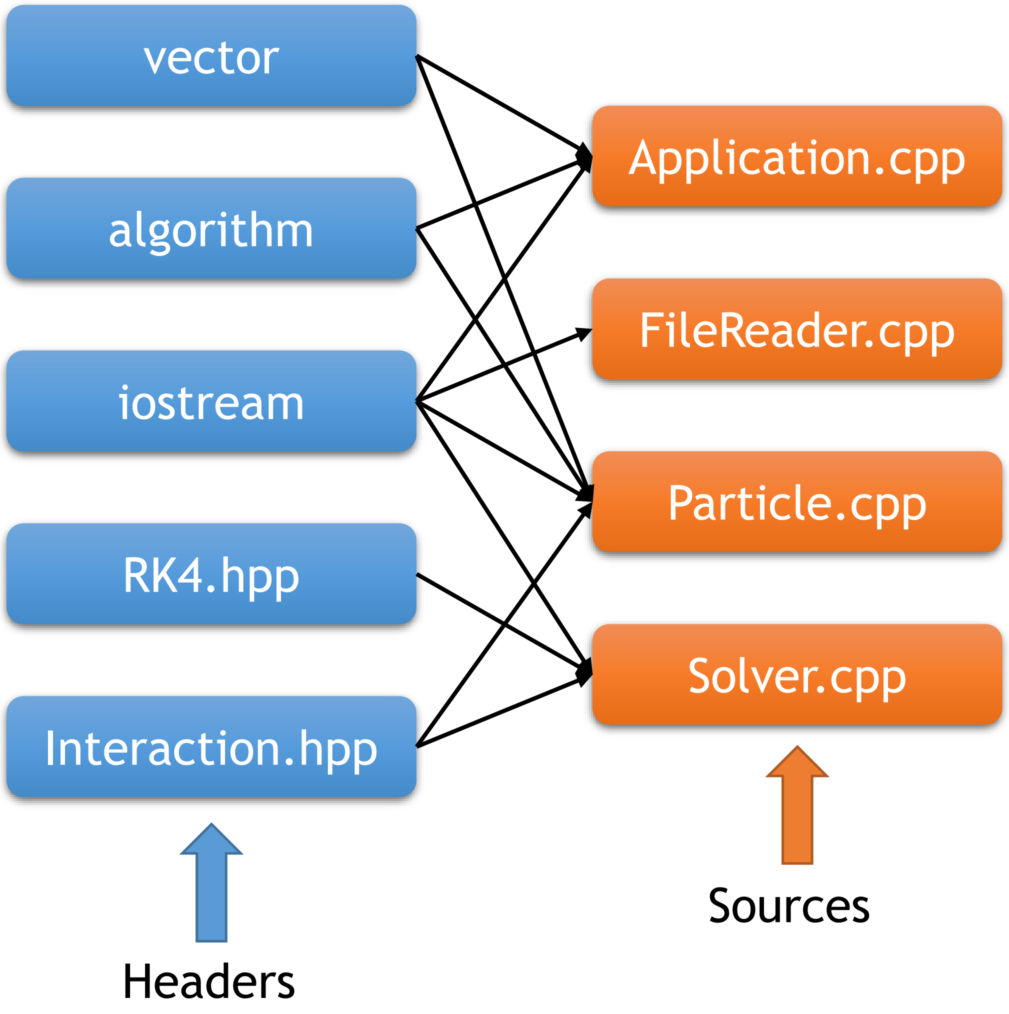
Please, no more compile.sh

How does a C/C++ application compile?



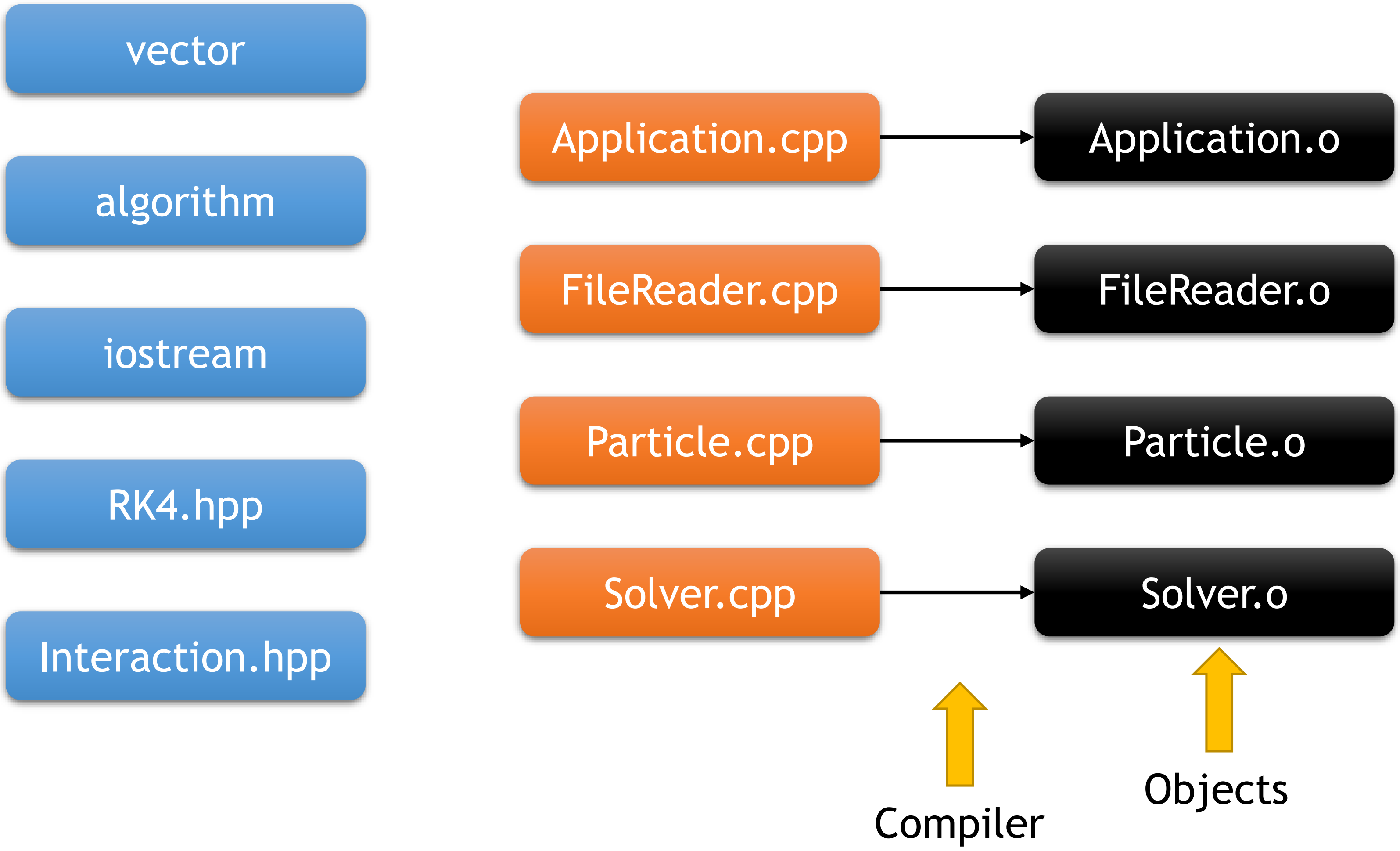
- Headers
  - Contain the declaration of functions
  - A declaration consists of the name of the function, and its signature
  - The signature are the types of the inputs and the type of the output
  - $func(\mathbb{M}, \mathbb{V}) \rightarrow \mathbb{V}$
- Sources
  - Contain the definition of functions
  - The definition is the actual body of the function, the series of commands to execute

How does a C/C++ application compile?



- Each source file references (includes) n headers
- Headers may reference each other
- C/C++ has a One Defintion Rule
  - Multiple inclusions of a header would violate ODR
  - Headers can be guarded against multiple inclusions (Include Guard)
- Why do we split code like this if it's so complicated?
  - Clear seperation of features from implementation
  - Compile times (see later)

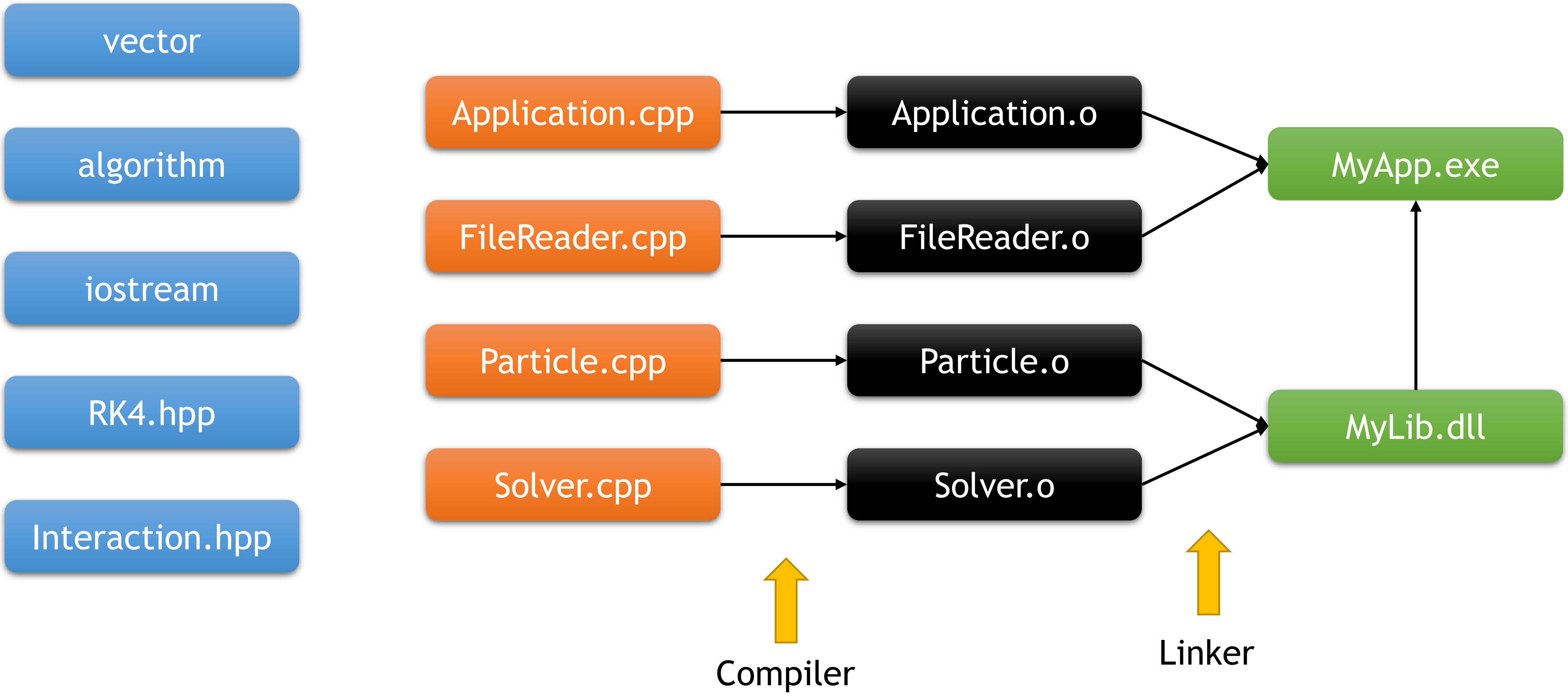
How does a C++ application compile?



- Object files contain decorated machine code
- They contain the native binary of the function bodies
- Decoration consists of compiler generated identifiers to functions called symbols

`__operator*(classMat,classVec)->classVec`

How does a C++ application compile?



How does a C++ application compile?

- Linking an executable
  - The linker inspects all object files, and looks for a special function (called main)
  - Checks which functions are actually needed to create a functional executable and throw away the rest
  - If some library is marked for linking, include those symbols too
  - Some functions may be compiled multiple times
    - If the binaries to the same symbol match, throw away all but one
    - If they mismatch, throw a link time error
  - If there is some symbol missing, throw a link time error
- By separating code to headers and sources, we minimize the chance of compiling the same function multiple times

Static versus dynamic libraries

Static

- Linking statically triggers inclusion of symbols directly into the executable
- Results in faster code
- If many executables refer to the same library, they all include the same code

Dynamic

- Linking dynamically triggers including only a reference to the symbol
- Results in smaller executable
- If many executable refer to the same library, the code exists only once on disk

What is a Build System?

- A tool that takes care of building your application in the fastest way possible with minimal user effort.
- The input is a make file, and the output is one or more binary/ies (hopefully). ☺
- Examples of build Systems:
  - GNU Make
  - NMake
  - MSBuild
  - Ninja
  - Qmake
  - CMake

Why use a Build System?

- Didn't I just say „Minimal user effort"?!
  - Build Systems aim at being as comfortable to use as possible
  - User declares the task, instead of specifying what to do
    - Declarative DSL, not imperative
- Didn't I just say „Maximum throughput"?!
  - Detects the minimal portion of the program that must be recompiled when editing code.
    - Uses time stamps
  - Processes independent parts of the build tasks in parallel
- Requires learning, but pays off in the long run!

Choosing a build system

Build System	Human readable	Graphical front-end	Portable	Generator
GNU Make	✓			
NMake	✓			
MSBuild	(✓)	✓	✓	
Ninja			✓	
Scons				
Waf				
Invoke-Build				
QMake	✓	✓	✓	✓
CMake	✓	(✓)	✓	✓

GNU Make

```
edit : main.o kbd.o command.o
display.o \ insert.o search.o
files.o utils.o cc -o edit main.o
kbd.o command.o display.o \
insert.o search.o files.o utils.o
main.o : main.c defs.h cc -c main.c
kbd.o : kbd.c defs.h command.h cc -
c kbd.c command.o : command.c
defs.h command.h cc -c command.c
display.o : display.c defs.h
buffer.h cc -c display.c insert.o :
insert.c defs.h buffer.h cc -c
insert.c search.o : search.c defs.h
buffer.h cc -c search.c files.o :
files.c defs.h buffer.h command.h
cc -c files.c utils.o : utils.c
defs.h cc -c utils.c clean : rm
edit main.o kbd.o command.o
display.o \ insert.o search.o
files.o utils.o
```

- Part of the GNU open-source software stack
- It is included in all Linux distributions
- User provides set of tasks
  - Task name
  - Dependency of task
  - Command-line to execute

NMake

Sample pending

- Part of Microsoft's Visual Studio software stack
- Should be considered legacy
- User provides set of tasks
  - Task name
  - Dependency of task
  - Command-line to execute
- Cannot perform tasks in parallel

MSBuild

Sample pending

- The build system that is currently used by Microsoft's Visual Studio
- It has been open-sourced and is available on Linux
- XML-based
  - Limited human-readability
  - Best used with a graphical front-end

Ninja

- Incredibly fast build system
- Sacrifices human readability
  - DSL favors not the user, but the machine
- It is meant to be generated by other tools, not hand authored
- Portable
- Open-source

QMake

- Make file generator
  - Provide one input
  - Ability to produce make files for multiple other build systems
- Portable
- Open-source
- Designed to serve the needs of the Qt Project

CMake

```
PROJECT(my_app)
LIST(SOURCES)
APPEND(SOURCES main.cpp
vector.cpp)
ADD_EXECUTABLE(${PROJECT_NAME}
SOURCES)
```

- Make file generator
- Portable
- Open-source
- Knows most languages by default
  - The known ones are EASY to use
  - Others can be taught
- DSL script language sometimes unfriendly
- Most cross-platform projects use it

Use something

- We are not workflow nazis anything is better than compile.sh
- If you don't know any build system,
  we highly recommend learning CMake
  - Extremely simple for small projects
  - Scales well (depending on scripting affinity/skill)
  - It is portable
  - It is mainstream (has great momentum)
  - Actively being developed (and is actually evolving)
- Even if you know one, we recommend giving CMake a chance

CMake + CTest + CPack + Cdash = EXIT_SUCCESS

- Kitware is the company behind the CMake suite of tools
- Full-fledged scripting language to do virtually anything
  - It is (finally) documented
  - Gazillions of tutorials online
- Feature missing?
  - It's open-source, so feel free to contribute
  - Don't have time? Hire us to do it!
- Big projects using CMake suite of tools
  - Bullet Physics Engine, CLion, Compiz, cURL, ROOT, GEANT4, GROMACS, KDE, libPNG, LAPACK, LLVM, Clang, MySQL, OGRE, OpenCV, SFML, zlib, …

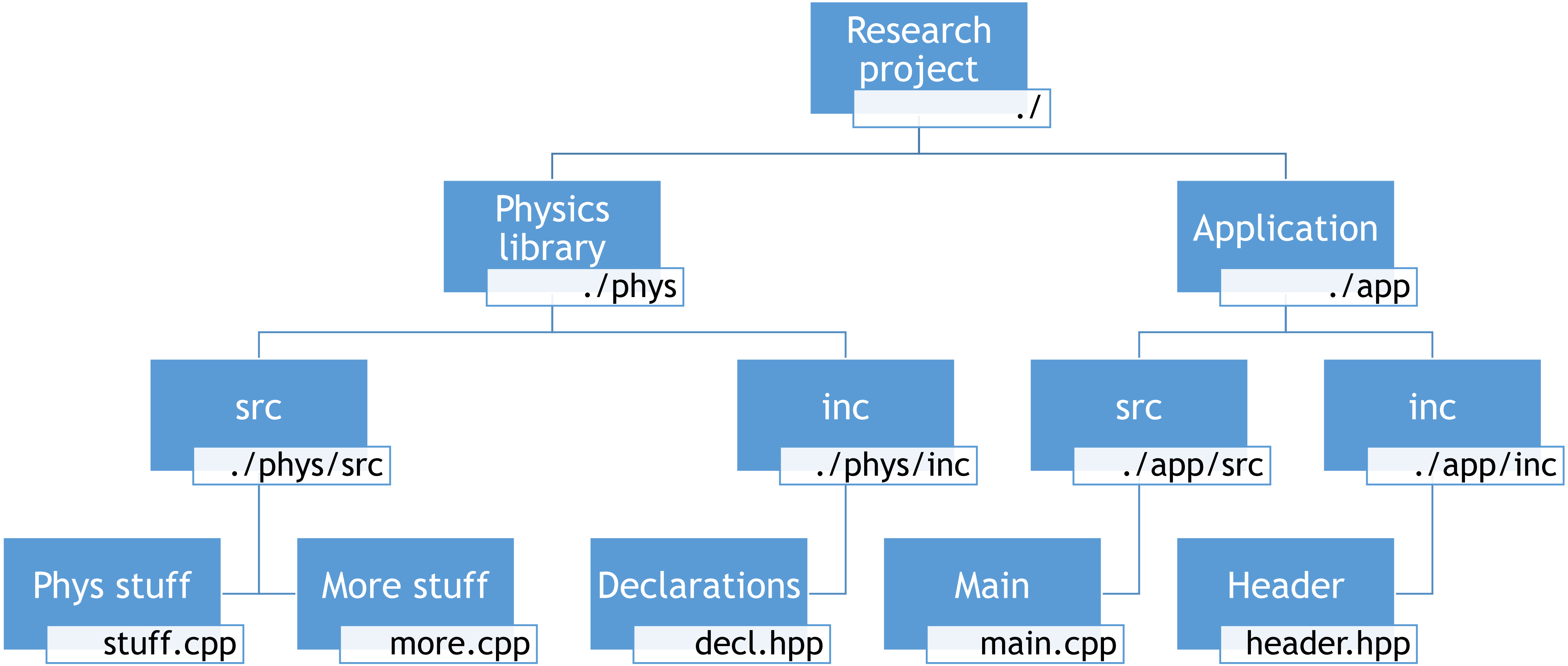
Why strive on remaining portable

- Portability is important!
  - Today, you might write the code for yourself, but tomorrow you might have to give it to a collegue
  - If your code is bound to a specific OS, compiler, etc. They will be more reluctant to use your code
- Dependencies
  - The portability of code is the union of restrictions imposed by:
    - Tools required to build the application
    - Environment required to run the application
  - Prefer portable tools over non-portable (have good reason to defect)
  - Understand the costs of depending upon external software (even OSS)

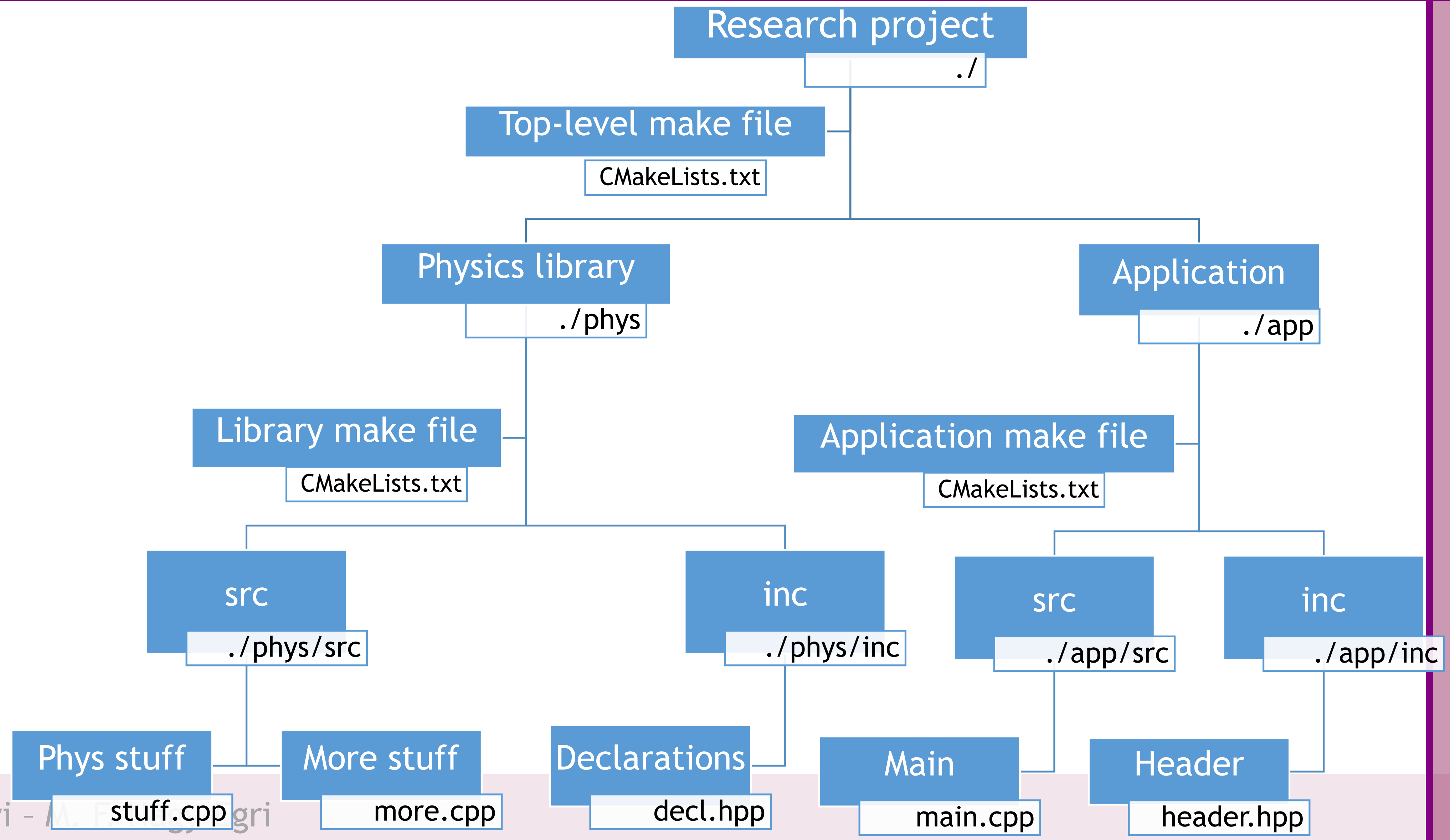
What can CMake do for you?

- A decent scripting language for authoring make files.
  - It is not declarative, but imperative (more powerful, but makes room for errors)
- Multiple (semi-)automated ways of discovering dependencies
- Ability to separate common build rules from platform, compiler specific rules

What can CMake do for you?



What can CMake do for you?



Top-level CMakeLists.txt

```
cmake_minimum_required (VERSION 2.8.11)
# CMakeLists files in this project can
# refer to the root source directory of the project as ${RESEARCH_SOURCE_DIR} and
# to the root binary directory of the project as ${RESEARCH_BINARY_DIR}.
project (RESEARCH)

# Recurse into the „phys" and „app" subdirectories. This does not actually
# cause another cmake executable to run. The same process will walk through
# the project's entire directory structure.
add_subdirectory (phys)
add_subdirectory (app)
```

Library CMakeLists.txt

```
cmake_minimum_required (VERSION 2.8.11)
# Create a library called „Phys" which includes the source files „stuff.cpp” and „more.cpp”.
# The extension is already found. Any number of sources could be listed here.
add_library (Phys src/stuff.cpp src/more.cpp)

# Make sure the compiler can find include files for our Phys library
# when other libraries or executables link to Phys
target_include_directories (Phys PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/inc)
```

Application CMakeLists.txt

```
cmake_minimum_required (VERSION 2.8.11)
# Add executable called „Application" that is built from the source files
# „main.cpp”. The extensions are automatically found.
add_executable (Application src/main.cpp)

# Make sure the compiler can find include files for our Application sources
target_include_directories (Application PUBLIC
${CMAKE_CURRENT_SOURCE_DIR}/inc)

# Link the executable to the Phys library. Since the Phys library has
# public include directories we will use those link directories when building
# Application
target_link_libraries (Application LINK_PUBLIC Phys)
```

Configuring the build system

```
PS C:\Users\Matty\Build\Research\NMake> cmake -G "NMake Makefiles" C:\Users\Matty\OneDrive\Develop\Tests\CMake\CMake_example\
-- The C compiler identification is MSVC 19.0.23026.0
-- The CXX compiler identification is MSVC 19.0.23026.0
-- Check for working C compiler: C:/Kellekek/Microsoft/Visual Studio/14.0/VC/bin/amd64/cl.exe
-- Check for working C compiler: C:/Kellekek/Microsoft/Visual Studio/14.0/VC/bin/amd64/cl.exe -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: C:/Kellekek/Microsoft/Visual Studio/14.0/VC/bin/amd64/cl.exe
-- Check for working CXX compiler: C:/Kellekek/Microsoft/Visual Studio/14.0/VC/bin/amd64/cl.exe -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: C:/Users/Matty/Build/Research/NMake
```

Invoking the build system

```
PS C:\Users\Matty\Build\Research\NMake> nmake

Microsoft (R) Program Maintenance Utility Version 14.00.23026.0
Copyright (C) Microsoft Corporation.  All rights reserved.

Scanning dependencies of target Phys
[ 20%] Building CXX object phys/CMakeFiles/Phys.dir/src/stuff.cpp.obj
stuff.cpp
[ 40%] Building CXX object phys/CMakeFiles/Phys.dir/src/more.cpp.obj
more.cpp
[ 60%] Linking CXX static library Phys.lib
[ 60%] Built target Phys
Scanning dependencies of target Application
[ 80%] Building CXX object app/CMakeFiles/Application.dir/src/main.cpp.obj
main.cpp
[100%] Linking CXX executable Application.exe
[100%] Built target Application
```

Few things to note

- Where did we specify in the make scripts how to invoke the compiler?
  - CMake looks for installed compilers and choses one it likes
  - Can be overriden when configuring the build
- What are the actual compiler switches, to make things work?
  - User must not need to know compiler options in the most common cases
  - Can be extensively customized if needed
- What order must things be built?
  - CMake builds dependency graph and generates make files accordingly

Just the tip of the iceberg

- CMake scripts are not declarative,
  but an imperative script language
- Turing complete (you can do ANYTHING with it)
- file command
  - Write to a file
  - Read from a file
  - Hash a file
  - Create directories
  - Download files
  - Upload files
  - Collect file names matching regex

What about my dependencies?

- Depending on a library built alongside the application is simple, but what about external dependencies?
- Find module
  - Module config files look for a given library in the most common install locations
    - On Linux it’s fairly trivial, on Windows it usually relies on env. vars.
  - If the library is found, it sets some variables that facilitate consumption
  - If not, it prompts the user to provide the root directory of the installation
  - There are 143 pre-installed FindModule.cmake files shipping with CMake.
- Let us omit the body of such a file. No black magic, but it is vastly outside to scope of this showcase.

Application CMakeLists.txt

```
# Look for common installation layouts of MPI
# If found, it will set some variables, otherwise it will throw an error
find_package (MPI REQUIRED)

# Make sure our application's sources find the include files of MPI
target_include_directories (Application PUBLIC ${MPI_INCLUDE_DIRS})

# Link the executable to the MPI library.
target_link_libraries (Application ${MPI_LIBRARIES})
```

But we can do better

- Couldn’t everything be done automatically?
- Package config
  - Package config files provide end-users with the exact layout of a given installation and all the tasks needed to consume the library
  - The libraries will always be found without user interaction, no matter how exotic the installation is
- How does it work?
  - Windows, HKEY_CURRENT_USER and HKEY_LOCAL_MACHINE registry entries hold paths for user wide and system wide registered packages
  - Linux, $(HOME)/.cmake/packages folder holds files with package paths

Application CMakeLists.txt

```
# Look for a registered clFFT installation
# Without „PACKAGE” it starts by looking for package and then for modules
find_package (CLFFT PACKAGE REQUIRED)

# We don’t need to set any include directories, as the package promotes
# usage to consumers

# Link the executable to the clFFT library.
target_link_libraries (Application PUBLIC CLFFT)
```

Unit Testing

- Writing modular code is good
  - Easier to maintain
  - Better chance at being reusable
  - Faster to compile (!)
  - Testable
- Imagine our phys library to contain only the impementations of physical phenomena
- This code might be reused elsewhere, our concrete simulation might only be one use case
- Seeing the expected results in one application does not mean that phys contains no bugs

Unit Testing

- Isolate parts of the code that can stand on it's own
- Create minimal use cases that have predictable outcome
  - Vector addition
  - Matrix multiplication
  - Periodic boundaries
  - Numerical stability
  - Etc.
- Check if all of your code behaves as expected in these minimal use cases
- If all your code passes Unit Testing, you have a much better chance to avoid bugs in consuming code

Enter CTest

```
# Enable testing functionality
enable_testing ()

add_executable (UnitTest1 src/test1.cpp)
target_link_libraries (UnitTest1 LINK_PUBLIC Phys)

# Add unit test that reads an input file, processes it and validates against
# a file of known correct results
add_test (NAME „Vector operations”
              COMMAND UnitTest1 --input detector.dat --validate result.dat)
```

CTest output

```
PS C:\Users\Matty\Build\Research\NMake> ctest
Test project C:/Users/Matty/Build/Research/NMake
    Start  1: Vector operations
 1/1 Test  #1: UnitTest1 ......................   Passed    1.58 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   1.58 sec
```

- By default checks if the exit code of UnitTest1 is 0 or not.
- Can be customized to match console or file output to another file or even a regular expression instead
- The formatting of CTest's output can also be customized

CPack for cross-platform packaging

- Applications built with CMake can trivially be packaged for distribution
- Because packaging varies greatly between platforms, requires duplicated „boilerplate”
  - Boilerplate is package author, company name, version, icons, contact, etc.
- 10-20 lines per platform can create
  - DEB packages
  - RPM packages
  - Self-extracting EXE installers

Version Control

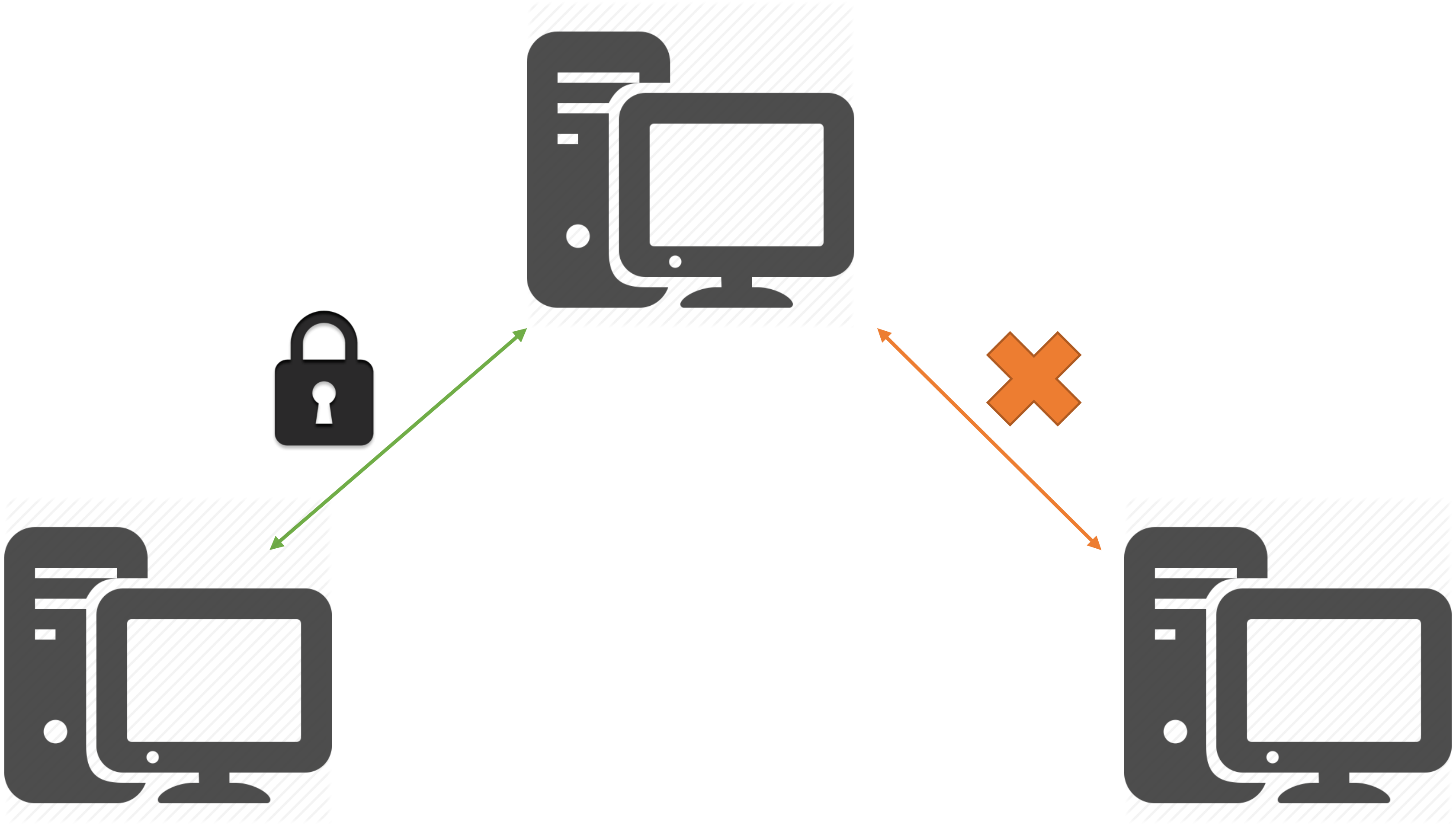
The art of roll-back

D. Berényi – M. F. Nagy-Egri

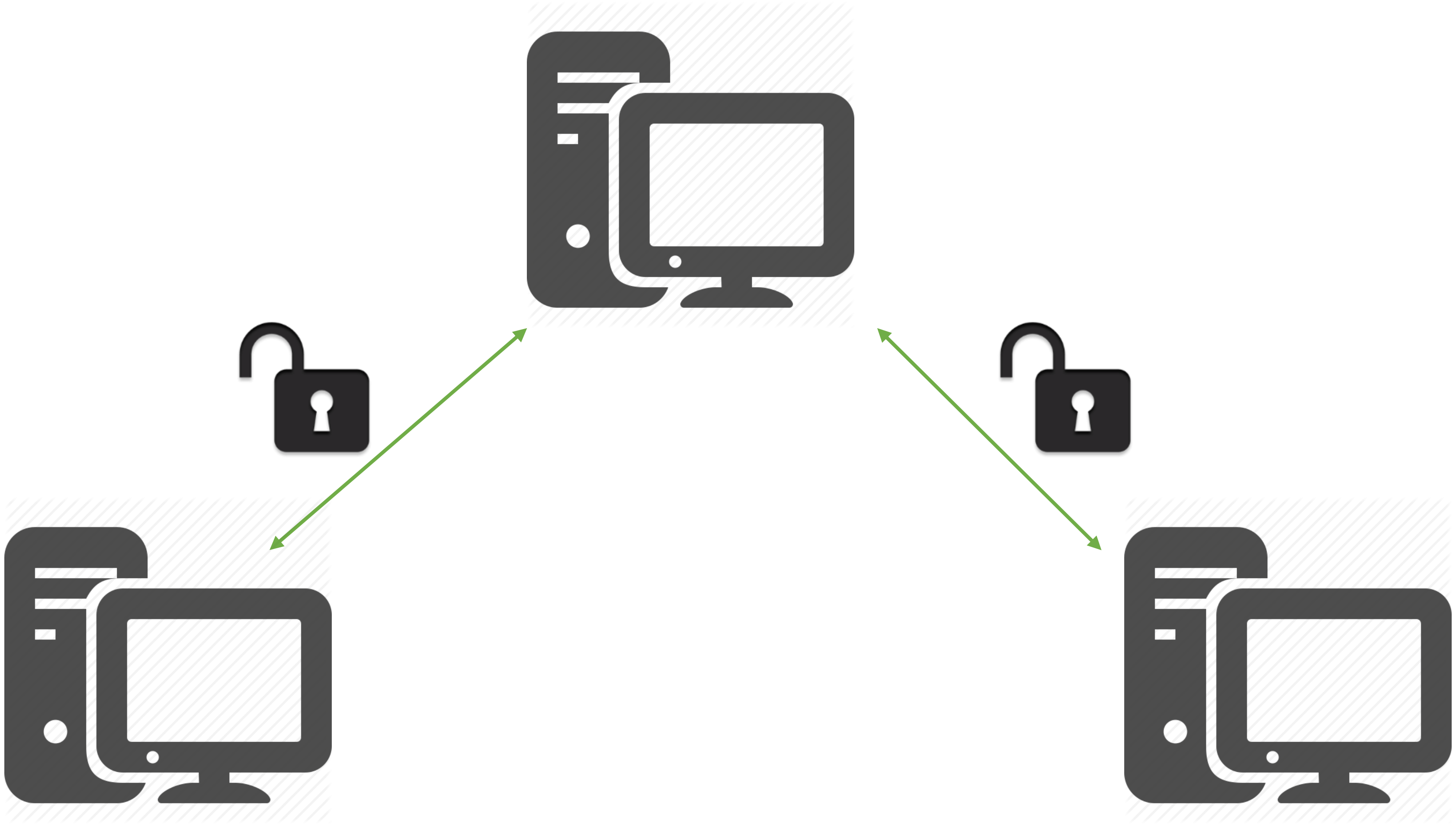
Version Control: Why should you care?

- Short version: the entire world is using it, so should you.
- Long version: even small scale software development is full of „trial and error”, which is not a linear workflow, but rather tree-like.
  - Updating the working copy of the source tree will result in times when your application is not functioning (might not even build)
  - Manually keeping functioning copies of the code base with feature A, feature A+B, feature A+C-B, etc. is tedious and you WILL MESS UP
  - Back-up is essential, cloud storage helps, but not alone
  - Collaborating without version control is very hard
- There is no holy grail, the best kind depends on your workflow

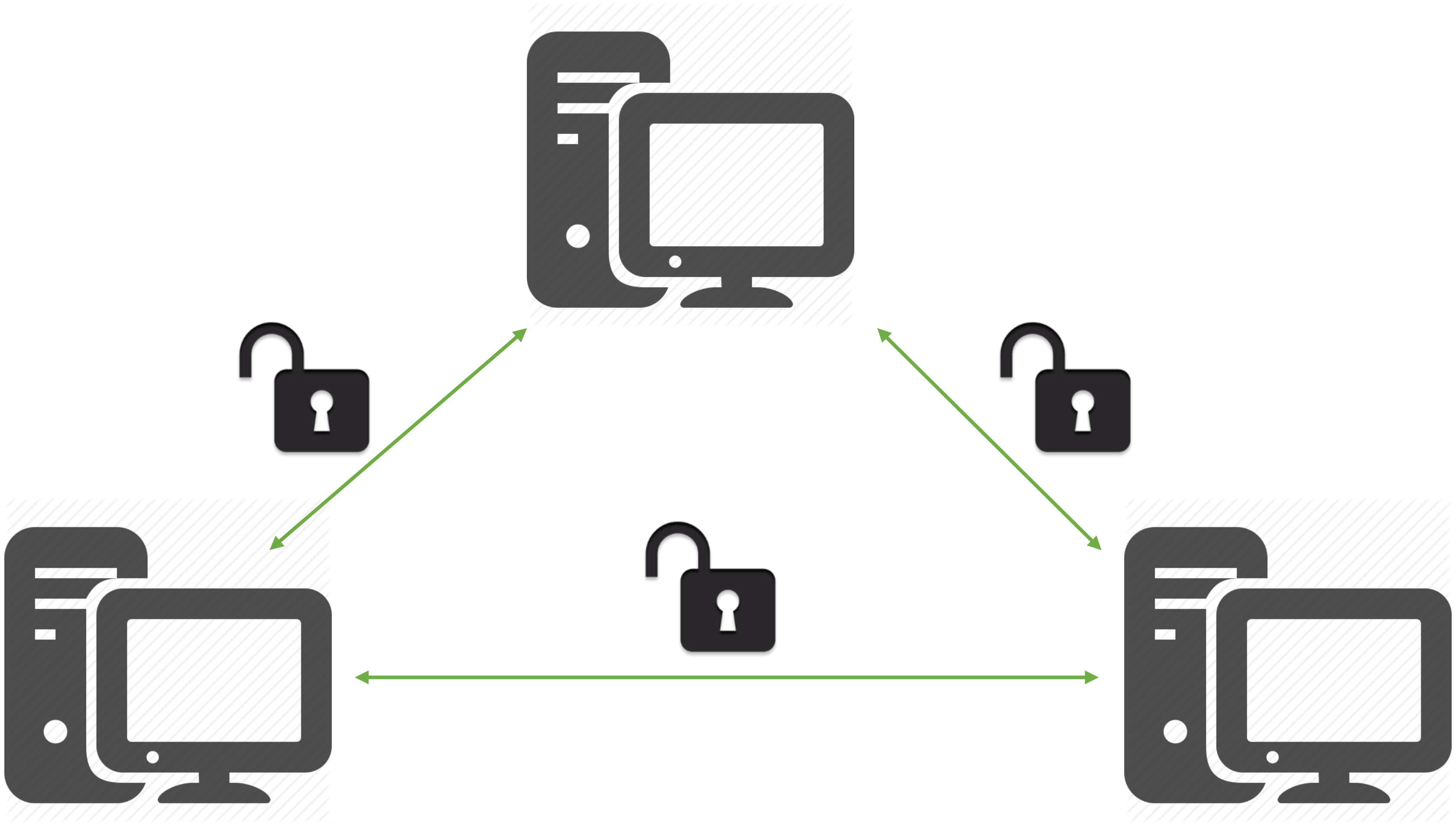
Centrallized Version Control - Locking



Centrallized Version Control - Merging



Distributed Version Control - Merging



Differences between

	Centrallized	Distributed
Hard drive space required for history?	None	Could be a lot
Who has the latest version?	Central „master version”	Depends on policy
Where is the full history?	Central machine	Local machine
Work offline?	No*	Yes
How fast are operations?	Network-dependent	Blazing, most are local
Branching and merging?	Reliable, use with caution	Reliable, use often
Learning curve?	Relatively simple	Relatively hard

Chosing the right one

- Examples of VCS
  - CVS
  - Subversion
  - Bazaar
  - VSS
  - TFVC
  - Mercurial
  - Git
- Some might suit your needs better than others, but we recommend one of two:
  - Git: very powerful, widespread/mainstream, fairly hard to learn
  - Mercurial: very good, widespread, easier to learn

Comparing Mercurial to Git

Mercurial is like James Bond

- Has all those sexy and easy to use gadgets
- Solves most problems in an instant
- In the rare cases when none of the gadgets are useful, he’s pretty much screwed

Git is like MacGyver

- Has a screwdriver and a hammer
- Can solve anything, with the given time and effort
- When hell breaks loose, he can assemble some ugly script that will ultimately save the day

Git FTW

- There are too many good tutorials online to provide an in-depth course in this limited time
  - Using Git with Visual Studio 2013
  - Learn Git branching
- There is a decent set of IDE support available as well as GUI and command line auxiliary tools
  - Posh-git
  - Tortoise Git
  - Git Extensions

General Git/Mercurial workflow

- Declare one branch as stable and always functional (master)
- Create branches for features/fixes you want to implement
- When a feature is ready, merge it into master
- This way
  - Switching between branches to work on half-baked features is safe and trivial
  - If your collegue asks you to do something with your app, there is always a functioning master to switch to

Setting up Git

- Set the default name, e-mail and push method associated with your commits

```
git config --global user.name "Gipsz Jakab„
git config --global user.name gipsz.jakab@wigner.mta.hu
git config --global push.default simple
```

- Set up SSH authentication to the Wigner Git server
  - In your `$(HOME)/.ssh/config` create an entry like

```
host wigner-git
        hostname git.wigner.mta.hu
        user gitolite
        port 9419
        identityfile ~/.ssh/id_rsa
```

- Write an e-mail to admin@wigner.mta.hu with your Public SSH Key for authetntication

Start working with Git

- Create a local repository on your dev box
  - `git init`
  - The repo is initially empty, at least one commit is required to create the default `master` branch
    - `git commit -a`
- Create a repository on a remote machine
  - Write an e-mail to admin@wigner.mta.hu with repo name and access control
  - Clone (fetch) the remote content (initially empty)
    - `git clone wigner-git:reponame`
  - `Do the first commit to create the master branch`

A simple development cycle

- Create a branch for a given feature
  - `git branch my-feature`
- Change to seeing the new branch (initially identical to master)
  - `git checkout my-feature`
- Create/delete/modify files, folders as needed
- Occasionally commit your work to the local repo
  - `git commit -A`
- When the feature is done and tested, merge it into `master`
  - `git checkout master`
  - `git pull master`
  - `git merge my-feature`
- Push your work to the remote repository
  - `git push`

Help?

- Whenever in doubt
  - `git branch`
  - `git status`
  - http://google.com

Integrated Development Environment

The swiss army knife of programming

D. Berényi – M. F. Nagy-Egri

The lazy programmer

"I will always choose a lazy person to do a difficult job because a lazy person will find an easy way to do it."

- Bill Gates, former Microsoft CEO

What is an IDE?

- Text editor
- Compiler
- Build System
- Versioning Control
- Profiler
- Documentation Generator
- Bug tracker
- Collaboration tool
- …

IDE versus toolchain

Integrated Development Environment

- Pro
  - End-to-end automation
  - Workflow is natural
  - Easy to learn, hard to master
- Con
  - Gotta cook with what you got

Toolchain

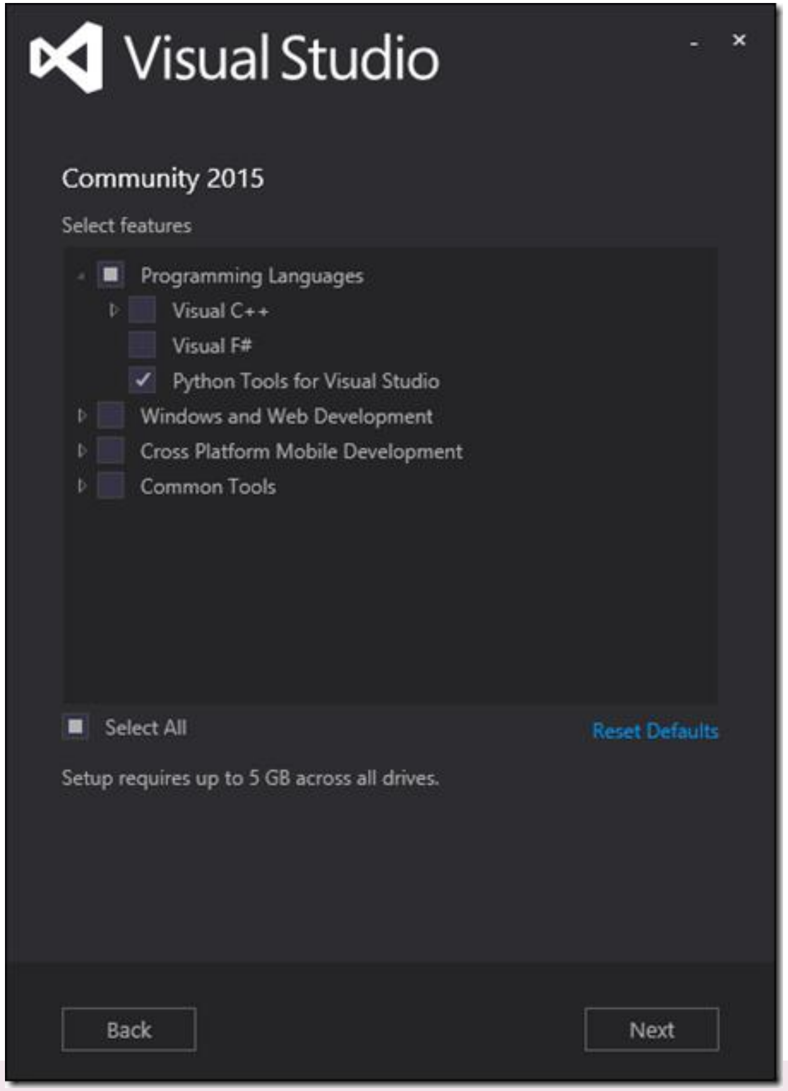
- Con
  - Distinct tools for everything
  - Some glitches here and there
  - Hard to learn, hard to master
- Pro
  - Choose the best of everything

Visual Studio

- The industry standard IDE
- Used to develop all of Microsoft's software
- By far the most full feautered IDE
- Exhaustive list of Add-Ins
- Is totally free for small dev teams or non-profit use

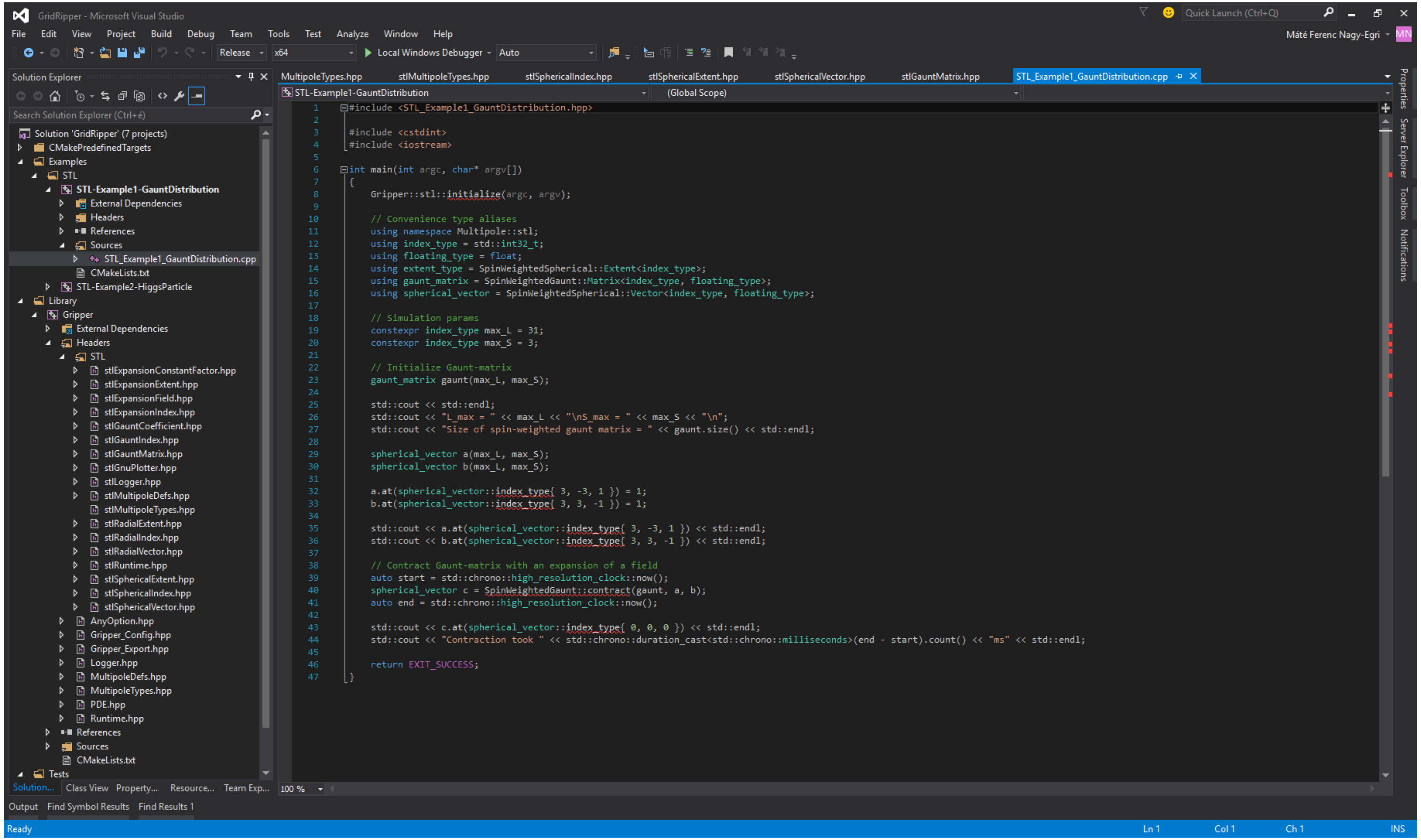
Installing Visual Studio

- https://www.visualstudio.com/
- Download Community 2015
- Run the installer
- Select development tools you need
    - Visual C++
    - Visual F#
    - Python
- Go and have lunch



How it looks like

- Text editor usually dominates the UI
- IntelliSense
- Visual representation of the build system
- Debug code visually
- Performance counters visualized
- Source Control integrated



Developing on Linux

- While Visual Studio pretty much rocks, not everyone is content with having to work on a Windows desktop
- Using IDEs are somewhat alien to the Linux developer community
  - Usually toolchains are preferred
  - While there are good IDEs out there, there is no real competition
- A non-exhaustive list of decent IDEs
  - Qt Creator
  - Code::Blocks
  - Eclipse
  - KDevelop

Qt Creator

Code less.
Create more.
Deploy everywhere.

- Widespread IDE for cross-platform development
- Used to develop most Qt applications
- Easy to install
- Easy to learn
- Is totally free for developers of open-source software

Installing Qt Creator

- Ubuntu
  - sudo apt-get install qtcreator
- OpenSUSE
  - zypper install qt-creator
- Scientific Linux
  - Yum install qt-creator

How it looks like

- Text editor usually dominates the UI
- Code completion
- Visual representation of the build system
- Debug code visually
- Create portable projects