

Intro to Efficient Programming

Mikolaj Krzewicki¹ Patrick Huhn³ Sandro Wenzel²

¹FIAS/University of Frankfurt

²CERN

³University of Frankfurt

HGS-HIRe power week

Limburg

June 2019

Lecture I

Introduction

Who we are

Mikolaj Krzewicki

- ▶ in ALICE since 2007
- ▶ ALICE High Level Trigger / ALICE O2
- ▶ Software validation
- ▶ TPC calibration
- ▶ Analysis (correlations and flow)

Patrick Huhn

- ▶ in ALICE since 2014
- ▶ PhD student since 2017
- ▶ former participant of this power week
- ▶ Analysis (charged particle R_{AA})

Sandro Wenzel

- ▶ at CERN since 2012; in ALICE since 2015
- ▶ Detector Simulation
- ▶ Previous experience in various computing activities (PhD: Quantum Monte Carlo; The Blue Brain Project)

Credits

Material of this course building on previous courses given by Jens Wiechula, David Rohr, Matthias Richter and Jochen Klein.

Outline / Goals of the week

Software development / computing in high energy physics is ubiquitous and essential knowledge. This week should help you to ...

- ▶ Be able to come up with efficient algorithms in C++ implementing software solutions to problems in (particle) physics.
- ▶ Be able to use ROOT as a library and to look into data using ROOT
- ▶ Be able to decompose problem into small pieces, structure code and work on software projects incrementally
- ▶ Understand tools and practices in the software development process
- ▶ Know about modern C++11 features and few optimization strategies
- ▶ Know about easy parallelism options and SIMD

Introduction

There are many aspects of *efficiency*

- ▶ Coding concepts
- ▶ Tools
- ▶ Fast development (prototyping)
- ▶ Fast code execution (optimization)
- ▶ Small memory imprint
- ▶ Code design
- ▶ Code flexibility / configurability
- ▶ ...

Programme

What we want to cover:

- ▶ Modern C++ features and concepts (C++11, C++14, ...)
- ▶ Tools
 - ▶ gcc
 - ▶ (c)make
 - ▶ git
 - ▶ doxygen
 - ▶ gdb
 - ▶ profilers:
valgrind,
perf, ...
- ▶ Methods
 - ▶ object orientation, templates
 - ▶ libraries
- ▶ Algorithms
- ▶ Parallelisation
- ▶ SIMD vectorisation

Outlook Programme 2

What we can't cover here but intend to do in part 2:

- ▶ GP-GPU
- ▶ Parallelization in more depth
- ▶ distributed computing and messaging
- ▶ ...

Course format

There will be some lectures but focus will be on practical side!

- ▶ Lot's of do-it yourself exercises/examples
- ▶ A real coding project touch typical high energy physics subject
- ▶ Possibility to do code reviews / interaction with lecturers

link to dynamic plan

<https://tinyurl.com/hgspw1>

A small project

- ▶ small groups (up to 5 people)
- ▶ work shall be carried out over the whole week, presentation of results on Friday (20+5), code reviews in between
- ▶ we want you to
 - ▶ use the tools
 - ▶ try the methods
 - ▶ test the algorithmswhich are discussed during the meeting
- ▶ you should learn something
⇒ try and understand what you are doing

Computing I

- ▶ local servers (hostnames: power[1-4].power.week)
personal user accounts:
username: first letter of firstname + lastname,
initial password: pwLimburg
(reference environment, you can compare to your machine)
- ▶ Every groups gets assigned one server (please use it exclusively)
- ▶ separate network with WLAN access (or cable)
SSID: PowerWeek_01
pw: powerweek
- ▶ passwordless login
often it is convenient to login using ssh keys

```
ssh-keygen  
ssh-copy-id <you-user-id>@power[1-4]
```

Computing II

- ▶ examples and slides are provided via git

```
git clone https://github.com/hgspowerweek/powerweek1/  
cd powerweek1  
git pull
```

do the last step before every session and you will get the latest examples and slides

- ▶ Slides are directly in this folder as pdfs
- ▶ Examples are in the folder *examples*

examples/bla

Lecture II

Introduction to code / document
management using git – absolute basics

Code repository

Why would you use a code repository?

- ▶ keep control over your changes
- ▶ keep a history of changes and go back to any previous state
- ▶ add logging messages to individual changes
- ▶ develop different topics in parallel
- ▶ keep a working version as a reference
- ▶ create releases for distribution of the code
- ▶ synchronize several developers

A code repository can serve as

- ▶ back-up solution
- ▶ communication medium
- ▶ team and product management tool

git - the stupid content tracker

- ▶ Developed by Linus Torvalds and others in 2005 for the Linux kernel community
- ▶ Nobody knows what git stands for at least one does not get a real answer.
- ▶ Instead of being stupid - see manpage - it's an extremely powerful scalable, distributed revision control system.

In contrast to other versioning systems (CVS, subversion),

- ▶ git allows to use the full functionality of a code repository locally
- ▶ can be distributed
- ▶ does not require a central server, but can be used with a server

git commands in a nutshell

Basic operations:

- ▶ `git init`
- ▶ `git clone`
- ▶ `git add` – add something to staging area
- ▶ `git commit` – commit staging area to local repository
- ▶ `git checkout <commit>` – retrieve certain state

Getting information

- ▶ `git log` – show commit history
- ▶ `git status`
- ▶ `git diff` – show differences (between commits)

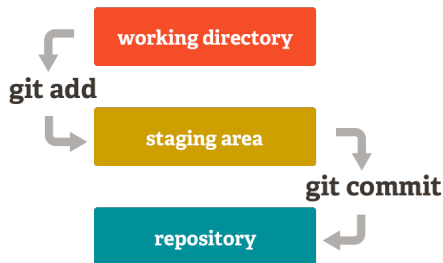
Advanced commands

- ▶ `git stash`
- ▶ `git push` – publish repo somewhere else (some URL)
- ▶ `git pull` – sync/get changes from somewhere else (some URL)
- ▶ `git rebase/merge` – integrate someone elses changes; change history

git - the stupid content tracker

How is a git repository structured:

- ▶ **local copy** (tree) - those are your files.
 - ▶ **staging area** (intermediate store) - changes (diffs) staged for commit.
 - ▶ **local repository** - full repository containing all changes to all branches.
- Everybody has a full copy, there is no concept of a central repository - you still may declare some repository the central one.



git - One time (identity) setup

- ▶ `git config` Configure git or query configuration
- ▶ **Some essential setup**: Give yourself a git identity:

```
git config --global user.name "Foo Bar"  
git config --global user.email foo.bar@cern.ch
```

- ▶ `git config -l` will show you the whole configuration including your identity
- ▶ configuration is stored in a file `${HOME}/gitconfig` which can also be edited

git - Creating and cloning repositories

Creating an initial repository:

```
mkdir -p ~/src/project  
cd ~/src/project  
git init  
Initialized empty Git repository in ~/src/project/.git
```

Cloning a repository:

```
git clone power1:/data/PowerWeek  
Cloning into 'PowerWeek'...  
Password:  
remote: Counting objects: 6, done.  
remote: Compressing objects: 100% (4/4), done.  
remote: Total 6 (delta 1), reused 0 (delta 0)  
Receiving objects: 100% (6/6), done.  
Resolving deltas: 100% (1/1), done.  
Checking connectivity... done
```

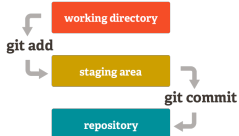
git - status of the local repository

Command: `git status`

```
richterm@power1 ~ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   twoparticle.C
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       result.root
#       twoparticle_C.d
#       twoparticle_C.so
no changes added to commit (use "git add" and/or "git commit -a")
```

- ▶ information about the current branch
- ▶ files which are staged for commit
- ▶ tracked files with local changes
- ▶ untracked files (can be masked by `.gitignore`)

git - committing

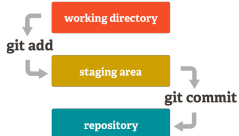


Step 1: `git add` mark changes to be committed

```
richterm@power1 ~/src/example_01 $ git add twoparticle.C
richterm@power1 ~/src/example_01 $ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   twoparticle.C
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       result.root
#       twoparticle_C.d
#       twoparticle_C.so
```

git - committing

Step 2: `git commit` changes



```
richterm@power1 ~/src/example_01 git commit -m "initial version  
of particle class"  
[master a56e827] initial version of particle class
```

Now it is locally committed, check the log

```
richterm@power1 ~/src/example_01 git log  
commit a56e8270fd6f3c99d4cdbcd0e45f287e1c71711  
Author: Matthias Richter <richterm@power1.power.week>  
Date: Tue Nov 26 11:51:09 2013 +0100
```

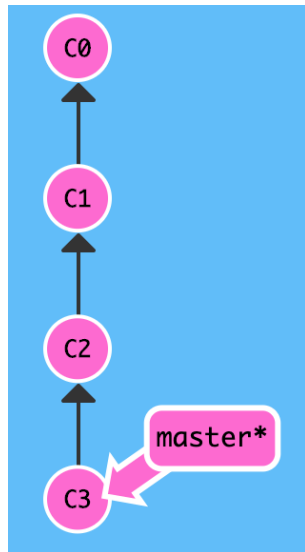
```
    initial version of particle class
```

```
commit ea2c328e7ac3e21c5546480dad1a55af4a0f5e35  
Author: Jochen Klein <jochen.klein@cern.ch>  
Date: Mon Nov 25 13:51:12 2013 +0100
```

```
    - initial commit of example
```

git commits; git checkout

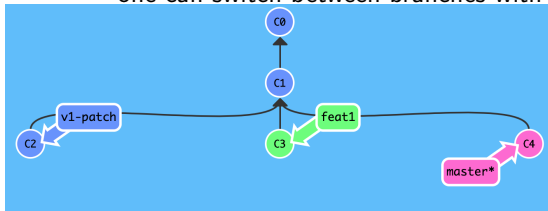
- ▶ git organizes commits in a connected **tree** of nodes; A **git commit** adds a new node
- ▶ each node consists of
 - ▶ The actual changeset/diff to files
 - ▶ Metadata (author, ...)
 - ▶ A commit message
 - ▶ A **SHA-256 hash digest** - This hash uniquely identifies the precise node and all its history!
- ▶ one can checkout specific nodes by using **git checkout <commit-sha>**
- ▶ pointer to last node – of main development line – is typically called **master**



The commit tree and branches

The git commit structure can be a tree. Pointers to leave nodes are called **branches**.

- ▶ The master branch is the main development line
- ▶ Other branches typically used for feature development in isolation (Feat1) or for releasing a certain stable version and patches (v1-patch)
- ▶ branches are started with `git checkout -b NewFeature` on the currently checked out commit
- ▶ one can switch between branches with `git checkout branchname`

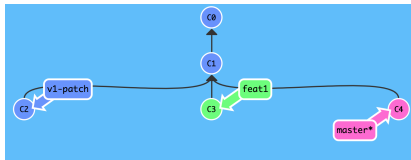


```
$ git checkout -b v1-patch
$ git commit
$ git checkout master
$ git checkout -b feat1
$ git commit
$ git checkout master
$ git commit
```

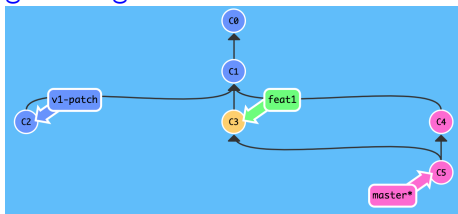
nice online learning platform: <https://learngitbranching.js.org/?NODEMO>

Short intro to merging and rebasing

- ▶ **merging/rebasing** : operations on the tree to bring together 2 branches
- ▶ used to integrate commits from one branch into the other
- ▶ for example when feature is fully developed

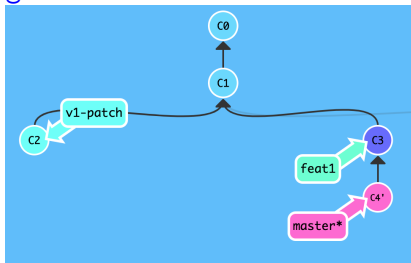


`git merge feat1`



- ▶ tree (old commits) stays intact
- ▶ merge creates adds a special commit

`git rebase feat2`



- ▶ branches are linearized
- ▶ no new commit; but old commits rewritten

git - looking at the difference

Command: `git diff`

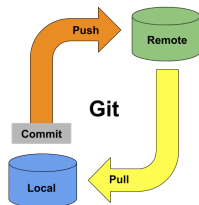
```
richterm@power1 ~/src/example_01 git diff twoparticle.C
diff --git a/twoparticle.C b/twoparticle.C
index fdca6ac..77d77c5 100644
--- a/twoparticle.C
+++ b/twoparticle.C
@@ -1,7 +1,7 @@
    // a simple macro with surprises for the purpose of training usage of valgrind a

    // include header files for the purpose of compilation
-#ifndef __CINT__
+#if !defined(__CINT__) || defined(__MAKECINT__)
    #include "TParticle.h"
    #include "TSystem.h"
    #include "TH1.h"
```

- ▶ shows local differences in tracked files
- ▶ without arguments: for all tracked files
- ▶ can be used to show differences between revisions

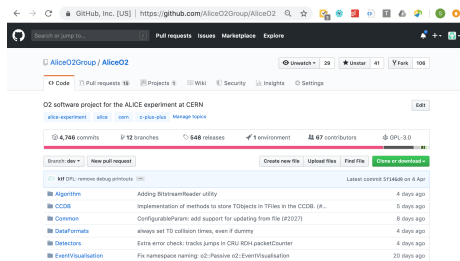
Distributed gits: push and pull

- ▶ A git repository can exist at various locations (or in multiple copies) at the same time (it's distributed); one speaks of local and remote repositories.
- ▶ the authoritative version of a git repository is often hosted on some web server (remote)
 - ▶ if you cloned from the remote it is called 'origin'
 - ▶ otherwise you can declare a remote repo with
`git remote add foo URL`
- ▶ Users synchronize local and remote repositories via git pull and git push commands.
 - ▶ `git pull [-rebase] foo` get all remote changes and apply locally
 - ▶ `git push foo` publish your own changes to the remote



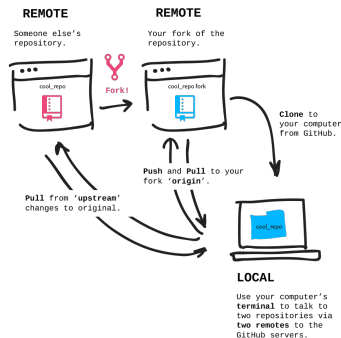
Github/Gitlab/Bitbucket

- ▶ many open source projects host git on platforms like Github, Gitlab or Bitbucket
- ▶ free git server and infrastructure
- ▶ collaborative code reviews
- ▶ integrations with other services
 - ▶ task tracking
 - ▶ continuous integration (CI) - automatic testing of new code and accepting only if good
 - ▶ documentation



Github/Gitlab/Bitbucket - workflow

- ▶ A typical workflow on those platforms uses 3 repositories
 - ▶ the authoritative repo (remote)
 - ▶ a fork (a user copy remote)
 - ▶ the local user repository
- ▶ Changes are integrated via push to fork and pull requests to real repo



git - Further reading

We found the following nice looking pages ... but there are tons other really

- ▶ <https://rogerdudler.github.io/git-guide/index.de.html>
- ▶ <https://try.github.io/> – resources to learn git;
- ▶ <https://www.atlassian.com/git/tutorials>
- ▶ <https://www.edureka.co/blog/git-tutorial/>
- ▶ `git help`

git - stash

You are in the middle of developing and get a request to fix something or update your clone

⇒ `stash` allows to save your current status

```
> git pull
# ... pull fails due to merge conflicts ...
> git stash save
> git pull
> git stash pop
```


git - checking logs revisited

tig; gitk; etc

Exercises

- ▶ Create a git repo; Do some add-commit cycles
- ▶ Play with branching merging:
 - ▶ locally
 - ▶ as an interactive game <https://learngitbranching.js.org/>
- ▶ Get familiar with github by forking or cloning the PowerWeek github repo <https://github.com/hgspowerweek/powerweek1>.
 - ▶ Look around
 - ▶ Contribute to the documentation of F.A.Q. section
- ▶ In order to contribute to an github repo, you need to create a github account
- ▶ For the coding project, we suggest to use gitlab.cern.ch in a private repository
 - ▶ needs CERN lightweight account account.cern.ch
 - ▶ keeps code private (also for future power weeks participants)
 - ▶ enables code review features

Lecture III

Code compilation

What is GCC

- ▶ Originally 'only' GNU C Compiler
- ▶ Release in March 1987 as the first **free** ANSI C optimizing compiler
- ▶ C++ support was added in December of that year
- ▶ Now, many other languages are supported as well, e.g. Objective-C, Objective-C++, Fortran (gfortran), Java (gcj), ...
- ▶ In addition, many different CPU architectures supported, e.g. Intel, ARM, Alpha, PowerPC, ...
- ▶ Today GCC stands for GNU Compiler Collection

Compilers

A compiler translates the human readable code into machine executable code

The main compilers of the GCC suite we are interested in are the GNU C and C++ compilers: gcc and g++

Brian Gough

<http://www.network-theory.co.uk/docs/gccintro/>

examples/gcc

First steps with g++

```
#include <iostream>
int main()
{
    std::cout << "hello world" << std::endl;
    return 0;
}
```

The hello world example above (hello_world.cpp) can be compiled using

```
g++ -Wall hello_world.cpp -o hello_world
```

-o specifies the name of the executable (default it is *a.out*)
-Wall turns on most commonly used compiler warning → **highly recommended to use**

To run the program simply type

```
./hello_world
```

⇒ Try it!

First steps with g++

```
#include <iostream>
int main()
{
    std::cout << "hello world" << std::endl;
    return 0;
}
```

The hello world example above (hello_world.cpp) can be compiled using

```
g++ -Wall hello_world.cpp -o hello_world
```

-o specifies the name of the executable (default it is *a.out*)
-Wall turns on most commonly used compiler warning → **highly recommended to use**

To run the program simply type

```
./hello_world
```

⇒ Try it!

First steps with g++

```
#include <iostream>
int main()
{
    std::cout << "hello world" << std::endl;
    return 0;
}
```

The hello world example above (hello_world.cpp) can be compiled using

```
g++ -Wall hello_world.cpp -o hello_world
```

-o specifies the name of the executable (default it is *a.out*)
-Wall turns on most commonly used compiler warning → **highly recommended to use**

To run the program simply type

```
./hello_world
```

⇒ Try it!

First steps with g++

```
#include <iostream>
int main()
{
    std::cout << "hello world" << std::endl;
    return 0;
}
```

The hello world example above (hello_world.cpp) can be compiled using

```
g++ -Wall hello_world.cpp -o hello_world
```

-o specifies the name of the executable (default it is *a.out*)

-Wall turns on most commonly used compiler warning → **highly recommended to use**

To run the program simply type

```
./hello_world
```

⇒ Try it!

Splitting code

Often it is useful to split the code into separate logical files

- ▶ Enhances readability and maintenance
- ▶ Enables to compile code parts independently
 - ▶ Saves compilation time, not all code needs to be recompiled if somethings changes
- ▶ Allows to compile code using the functionality of other code without knowing the actual implementation

We split the `hello_world` example into three files:

- ▶ `main.cpp`
- ▶ `hello_fn.h`
- ▶ `hello_fn.cpp`

Splitting code

Often it is useful to split the code into separate logical files

- ▶ Enhances readability and maintenance
- ▶ Enables to compile code parts independently
 - ▶ Saves compilation time, not all code needs to be recompiled if somethings changes
- ▶ Allows to compile code using the functionality of other code without knowing the actual implementation

We split the `hello_world` example into three files:

- ▶ `main.cpp`
- ▶ `hello_fn.h`
- ▶ `hello_fn.cpp`

Splitting code - example

main.cpp

```
#include "hello_fn.h"
int main()
{
    hello("world");
    return 0;
}
```

hello_fn.h

```
void hello(const char* to);
```

hello_fn.cpp

```
#include <iostream>
void hello(const char* to)
{
    // function to print hello to someone on the command line
    std::cout << "Hello " << to << std::endl;
}
```

Splitting code - header files

- ▶ Separate the *declaration* of classes / functions from the actual *implementation*
- ▶ The declaration is given in the *header file* ending on `.h`
- ▶ When using external code in an own class, during compilation only the declaration is needed
- ▶ A declaration should not be included several times (compilation time), this is handled by a pre-compiler directive (*header guard*)

```
#ifndef MYCODE_H
#define MYCODE_H
void myfunction(int x, float y);
#endif
```

Compile multiple source files

To compile the code run

```
g++ -Wall main.cpp hello_fn.cpp -o hello_world
```

⇒ Try it!

Not won too much, still all code is compiled all the time

- ▶ compile parts of the code into separate *object files*
- ▶ *link* the *object files* to the executable

Compile multiple source files

To compile the code run

```
g++ -Wall main.cpp hello_fn.cpp -o hello_world
```

⇒ Try it!

Not won too much, still all code is compiled all the time

- ▶ compile parts of the code into separate *object files*
- ▶ *link* the *object files* to the executable

Creating object files

We create one *object file* per input file:

```
g++ -Wall -c main.cpp hello_fn.cpp
```

- ▶ `-c` tells the compiler to create an object
- ▶ object files are machine code, but not yet executable

Produces the object files *main.o* and *hello_fn.o*

This can also be run separately for each file

```
g++ -Wall -c main.cpp  
g++ -Wall -c hello_fn.cpp
```

⇒ Try it!

Creating object files

We create one *object file* per input file:

```
g++ -Wall -c main.cpp hello_fn.cpp
```

- ▶ `-c` tells the compiler to create an object
- ▶ object files are machine code, but not yet executable

Produces the object files *main.o* and *hello_fn.o*

This can also be run separately for each file

```
g++ -Wall -c main.cpp  
g++ -Wall -c hello_fn.cpp
```

⇒ Try it!

Linking objects to an executable

Now the objects can be *linked* together to the executable:

```
g++ main.o hello_fn.o -o hello_world
```

NOTE:

The code is already compiled → You don't need warning options

⇒ Try it!

- ▶ Modify something in one of the files (e.g. world → moon in main.cpp)
- ▶ Recompile only main.cpp
- ▶ Link all files to one executable

⇒ Try it!

Linking objects to an executable

Now the objects can be *linked* together to the executable:

```
g++ main.o hello_fn.o -o hello_world
```

NOTE:

The code is already compiled → You don't need warning options

⇒ Try it!

- ▶ Modify something in one of the files (e.g. world → moon in main.cpp)
- ▶ Recompile only main.cpp
- ▶ Link all files to one executable

⇒ Try it!

Makefiles

- ▶ The steps mentioned above can be automatized using the *make system*
- ▶ Define dependencies (e.g. the executable can only be built if all objects files are available)
- ▶ Only compiles code which changed

Documentation:

<https://www.gnu.org/software/make/manual/make.html>

Makefiles – An example

```
CXX      = /usr/bin/g++
CXXFLAGS = -Wall -Wextra -Wconversion -Wshadow -g
LDFLAGS  =

OBJ      = hello_fn.o main.o

split: $(OBJ)
        $(CXX) -o $@ $(OBJ) $(LDFLAGS)

main.o: main.cpp hello_fn.h
        $(CXX) -o $@ -c $< $(CXXFLAGS)

%.o: %.cpp %.h
        $(CXX) $(CXXFLAGS) -c $< -o $@

clean:
        @rm -f ${OBJ} split
```

Makefiles – Primer

Makefiles consist of rules that tell the make system what to do. A rule has the form:

```
target ... : prerequisites ...  
<tab> recipe  
<tab> ...  
<tab> ...
```

target is usually an output file name, prerequisites can be other targets or e.g. file names

NOTE: A recipe MUST be started with a `< tab >`

When the `make` command is called, it looks for a file called *Makefile* or *makefile* in the present directory and processes the first target (*default target*)

Makefiles – Primer

A few important automatic variables are defined in the `make` system:

- ▶ `$@` The target name
- ▶ `$<` The first prerequisite
- ▶ `$^` The names of all the prerequisites, with spaces between them

For more see <https://www.gnu.org/software/make/manual/make.html#Automatic-Variables>

⇒ Try it!

CMake - introduction

- ▶ treatment of dependencies and automatic re-compilation covered by Makefiles and make
- ▶ manual maintenance of Makefiles can become tedious and error-prone
- ▶ configuration for specific setup of SDK and external dependencies covered by autotools suite or CMake

CMake - introduction

CMake is an open-source, cross-platform family of tools designed to build, test and package software.

- ▶ universal (toolchain agnostic) description of build flow in CMakeLists.txt
- ▶ automated creation of Makefiles depending on configuration options environment found
- ▶ automatic re-evaluation when needed
- ▶ possibly add testing and packaging steps

here: CMake with Makefiles

typical build layout

- ▶ separate build into different directories:
 - ▶ source: no generated files
 - ▶ build: generated files, object files, libraries, executables
possibly more than one with different build options
 - ▶ install: final files only
- ▶ test stage
- ▶ delivering/deploying stage

a minimal CMake project

```
cmake_minimum_required(VERSION 3.9)
project(MyPowerProject CXX)

add_executable(testExe test.cc)
```

- ▶ always specify minimum version of CMake (depending on the features you use)
 - ▶ declare project and language used
 - ▶ add targets (here just one executable)
-

example project:

```
examples/cmake_simple
```

a CMake run

```
cd <build directory>  
cmake <source directory>  
make -j$(nproc)
```

build happens in phases:

- ▶ configuration:
evaluation of CMakeLists.txt files, additional options, toolchain probing
- ▶ generation:
generation of build files (depending on selected generator)
- ▶ compilation:
actual build using Makefiles

libraries



```
add_library(power power.cc)
add_executable(main main.cc)
target_link_libraries(main power)
```

- ▶ add target for library and source files needed to build it
- ▶ link executable against library
- ▶ include directories are propagated to the targets using the library

using lists

...

```
set(SOURCES s1.cc s2.cc s3.cc)
add_library(s ${SOURCES})

set(EXECUTABLES e1 e2 e3)
foreach(EXE ${EXECUTABLES})
    add_executable(${EXE} ${EXE}.cc)
    target_link_libraries(${EXE} power)
endforeach()
```

- ▶ define lists that can be reused
- ▶ avoid overly repetitive code

ROOT integration

...

```
find_package(ROOT)
include(${ROOT_USE_FILE})
if(ROOT_FOUND)
    message(STATUS "Using ROOT: ${ROOT_VERSION} <${ROOT_CONFIG}>")
    target_compile_definitions(power PUBLIC "-DUSE_ROOT")
    target_include_directories(power PRIVATE ${ROOT_INCLUDE_DIRS})
    target_include_directories(power PRIVATE .)
    ROOT_GENERATE_DICTIONARY(G__Power ${CMAKE_CURRENT_SOURCE_DIR}/power.h LINKDEF Lin
    target_sources(power PRIVATE power_rooted.cc G__Power)
    target_link_libraries(power ROOT::Core ROOT::Gui ROOT::Tree)
endif(ROOT_FOUND)
```

- ▶ ROOT comes with additional tools to build dictionaries

warnings and errors

```
message("something important")  
message(STATUS "just a status message")  
message(WARNING "something's fishy here")  
message(ERROR "this is plain wrong")  
message(FATAL_ERROR "this is too wrong, I rather die ...")
```

- ▶ use messages to check on your build,
don't be blind on what is happening

language options

...

```
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
message(STATUS "Using C++${CMAKE_CXX_STANDARD}")

enable_language(CUDA)
```

- ▶ compiler-agnostic settings of language standard
N.B.: you can also request specific language features
- ▶ enabling of additional programming languages

build types and compiler options

```
set(CMAKE_CXX_FLAGS_DEBUG "-O0 -ggdb -DDEBUG -D__DEBUG")
set(CMAKE_CXX_FLAGS_RELWITHDEBINFO "${CMAKE_CXX_FLAGS_RELEASE} -ggdb")
set(CMAKE_CXX_FLAGS_RELEASE "-O3 -march=native -ftree-vectorize -ffast-math -DNODEBUG")
message(STATUS "Using CXX flags for ${CMAKE_BUILD_TYPE}: ${CMAKE_CXX_FLAGS_${CMAKE_BUILD_TYPE}}")
```

- ▶ compiler options/flags are controlled by build types
- ▶ can be changed separately for different build types
- ▶ don't put target-specific stuff here

```
cmake -DCMAKE_BUILD_TYPE=DEBUG <source> <dir>
```

default build type

```
# by default build optimized code with debug symbols
if(NOT CMAKE_BUILD_TYPE AND NOT CMAKE_CONFIGURATION_TYPES)
  set(CMAKE_BUILD_TYPE RELWITHDEBINFO)
endif()

set(CMAKE_ALLOWED_BUILD_TYPES DEBUG RELEASE RELWITHDEBINFO)
if(NOT CMAKE_BUILD_TYPE IN_LIST CMAKE_ALLOWED_BUILD_TYPES)
  message(FATAL_ERROR "Invalid build type ${CMAKE_BUILD_TYPE}. Use one of: ${CMAKE_
```

- explicit control over default build type

install

```
install(TARGETS main power  
        LIBRARY DESTINATION lib  
        RUNTIME DESTINATION bin  
)
```

- ▶ this should install only the final build products
- ▶ try and adhere to conventions about installation paths

some more involved stuff

- ▶ nested projects
- ▶ tests

nested projects

```
add_subdirectory(sub1)
add_subdirectory(sub2)
```

- ▶ include sub-projects in sub-directories, hierarchical layout
- ▶ sub-directories can contain project, this allows to build the sub-project independently
- ▶ different directories:
 - ▶ CMAKE_SOURCE_DIRECTORY: directory from which cmake was run
 - ▶ CMAKE_PROJECT_SOURCE_DIRECTORY: (sub-)project directory
 - ▶ CMAKE_CURRENT_SOURCE_DIRECTORY: current directory in source tree

CTest

- ▶ you can add tests which can be run programatically

```
enable_testing()  
  
add_test(NAME MyPowerTest COMMAND echo done)
```

```
ctest
```

You can also run this using the *make* process by

```
make test
```

summary

- ▶ surely not an extensive coverage of CMake, but it should get you started
- ▶ extensive documentation on project pages but not always good explanation of the underlying concepts

Lecture IV

Project

The problem

- ▶ consider an experiment to measure the X particle
- ▶ a renowned theorist told us some expectations:
 - ▶ decay to three charged pions
 - ▶ mass 50 – 100 GeV, known lifetime $c\tau = 0.5$ mm
 - ▶ production flat in η and $\propto p_T^{-5}$
 - ▶ in every n -th event (poisson around that mean), n most probably 5
- ▶ available detector covers $|\eta| < 2$ and full azimuth, provides one point at $R = 5$ cm with $\sigma = 0.1$ mm in z and $r\varphi$ direction, θ with a resolution of 1 degree and p_T with a resolution of $\Delta p_T / p_T = 0.5 \text{ \%} / (\text{GeV}/c) \cdot p_T$,
- ▶ on average 10 (poisson with mean 10) other primary particles (pions) produced, flat in η and $\propto p_T^{-8}$
exploit the known lifetime
- ▶ find a way to prove (or falsify) the existence of the X, which properties can be measured

Your task

Write the code to

- ▶ simulate the production of the X particle and its decay according to the specified properties
- ▶ simulate background particles produced in association with X
- ▶ smear the measured track properties (\vec{x} , p_T , η) (fast detector simulation)
- ▶ reconstruct the X particle
- ▶ analyze the performance of the reconstruction
- ▶ develop clever cuts on the analysis level

**Simulation, smearing, reconstruction, and analysis
should be kept separated!**

Some thoughts

- ▶ IO is usually quite slow, but might be important, optionally, for debugging purposes \Rightarrow use root trees
- ▶ simulation writes information to memory structure(s)
what information needed? how structured?
- ▶ smearing runs on these data / tree input
modify? copy? extend? file structure?
- ▶ reconstruction runs on these in memory structure(s) / tree input
what information is needed? what is produced?
- ▶ analysis runs on these in memory structure(s) / tree input
what information is needed?
- ▶ for specific problems (e.g. three-body decay) you might want to use external libraries (ROOT – e.g. TGenPhaseSpace)

You are free to do what you want.

Details

- ▶ Primary particles are created at the origin (exact position, no smearing).
- ▶ All spectra are flat in η and follow a power law in p_T , cut off at 100 MeV.
- ▶ The resolution of the tracking layers is $\sigma = 1$ mm in z and in $r\varphi$ direction (Gaussian). There is no error in r .
- ▶ The detector has full acceptance, every particle crossing a layer produces exactly one hit.
- ▶ For generating the hits, assume perfectly cylindrical detector layers.
- ▶ Mass of particles: X: 50 – 100 GeV
- ▶ Lifetime of particles: X: $c\tau = 0.5$ mm
- ▶ The mean number of background particles is $dN/d\eta = 2.5$, the mean number of X particles is $dN/d\eta = 0.25$ (Poissonian Distribution).
- ▶ The p_T spectrum of the X is proportional to p_T^{-5} , the p_T spectrum of the background is proportional to p_T^{-8} .

Lecture V

Object-oriented programming

Why object-oriented programming?

- ▶ paradigms:
 - ▶ machine operations (assembler)
 - ▶ functional formulation (Lisp, Lua, ...)
 - ▶ procedural languages (Fortran, C)
 - ▶ object-oriented (C++, Java, ...)
- ▶ problems before object-oriented programming:
 - ▶ inter-dependencies
 - ▶ name clashes
 - ▶ change of internal representations difficult
- ▶ solution:
 - ▶ encapsulate data and methods to access it
 - ▶ make it reusable without knowing the internal details
 - ▶ C with objects \rightsquigarrow C++

Objects

- ▶ user-defined type,
on equal footing with language types
- ▶ aggregate of
 - ▶ member variables (data)
 - ▶ methods
 - ▶ access policies
- ▶ use a object to describe a concept, e.g.
 - ▶ particle with properties
 - ▶ functor (map)
- ▶ C++ supports you to use objects,
it does not force you to write everything as a object!
- ▶ Object can be a *class* or a *struct*

Classes

Example

```
class particle {  
    public:  
        particle() = default;  
        ~particle() = default;  
        particle(const particle &rhs) = default;  
        particle& operator=(const particle &rhs) = default;  
  
        float pt() const  
            { return sqrt(p[0]*p[0] + p[1]*p[1]); }  
  
    protected:  
        std::array<float, 4> p{};  
};
```

- ▶ only a concept so far, no object yet

Instances

- ▶ you can create objects from a class (instances)

// on the stack

```
particle p1{};
```

// on the heap

```
auto p2 = make_unique<particle>();
```

- ▶ only then:
 - ▶ allocation of memory
 - ▶ execution of constructor
- ▶ allocation on the stack and heap behave differently, try it out!

Inheritance

- ▶ extend the concept described by a base class
- ▶ objects of the derived class can be used where an object of the base class is asked for

- ▶ avoid re-writing code

```
class particle_spin : public particle {  
    public:  
        particle_spin() : particle(), spin(0) {}  
  
    protected:  
        float spin;  
};
```

Virtual functions

```
class particle {  
    ...  
    virtual void decay();  
    ...  
};  
  
class particle_spin : public particle {  
    ...  
    virtual void decay();  
    ...  
};
```

Polymorphism

- ▶ code written for a particle should also work for a particle with spin

```
particle_spin p_spin();  
particle &p = p_spin;  
// ...  
p.decay();
```
- ▶ while you use a reference to a particle,
you (probably) want the method of the derived class to be called
- ▶ decision which method to call must happen at run-time!
↪ objects have vtable for virtual functions
- ▶ only overload virtual functions
(although C++ does not enforce this)

TFile class hierarchy

- ▶ TObject
- ▶ TNamed
- ▶ TDirectory
- ▶ TDirectoryFile
- ▶ TFile
- ▶ TNetFile, TMemFile, TXMLFile, TAlienFile, ...

have a look at:

<http://root.cern.ch/root/html/TFile.html>

Example

Try it out!

`examples/class`

Inheritance vs Member

- ▶ sometimes you can consider a class as a
 - ▶ parent class
 - ▶ member object
- ▶ with inheritance:
you can use the derived class in place of the parent class
- ▶ with member object:
you hide the details of the object
- ▶ think about what you want to achieve

Abstract base class

- ▶ you can write a class for which no actual realisation exists but which shall serve as the description of an interface
- ▶ achievable by requiring the implementation of a virtual function

```
class abstract {  
    public:  
        abstract();  
        virtual int get() const = 0;  
};
```

- ▶ only when all required virtual functions are implemented in a derived classes, instantiation is possible

override

What will happen?

```
class Foo {
public:
    virtual void print(const char* something) {
        std::cout << "Foo prints " << something << '\n';
    }
};

class Bar : public Foo {
public:
    virtual void print(const char* something) const {
        std::cout << "Bar prints " << something << '\n';
    }
};

int main ()
{
    Bar bar;
    Foo &foo = bar;
    foo.print("hi");
}
```

override

Help the compiler to find errors in inheritance with function overriding.

```
class Foo {
public:
    virtual void print(const char* something) {
        std::cout << "Foo prints " << something << '\n';
    }
};

class Bar : public Foo {
public:
    void print(const char* something) const override {
        std::cout << "Bar prints " << something << '\n';
    }
};

int main ()
{
    Bar bar;
    Foo &foo = bar;
    foo.print("hi");
}
```

Templates

- ▶ family of classes:
generic programming of a class unspecified data type

```
template <typename T>
class dummy {
public:
    dummy();
    const T& get_a();
protected:
    T a;
};
```

- ▶ compiler generates specializations for this class
when you use them

```
dummy<int> a;
dummy<float> b;
```

- ▶ more details and Standard Template Library (STL) later

Lecture VI

Selected c++11 features

Selected c++11 features

Discussed features

- ▶ `auto`
- ▶ Range based for loops
- ▶ `nullptr`
- ▶ Strongly typed enums
- ▶ `override` and `final`
- ▶ move semantics
- ▶ default and delete functions

What will not be covered

- ▶ Threading Support
- ▶ Static Assertions
- ▶ Variadic Templates
- ▶ `decltype`
- ▶ Alignment control (i.e., `alignof`, `alignas`, etc.)
- ▶ `static_assert` and type traits
- ▶ ...

examples: `git pull`

auto

auto leaves the type definition to the compiler. Very convenient in many situation. Can save lots of typing or typedef declarations.

```
auto i = 0; // not that useful
auto f = 0.f;

auto histogram = new TH1F("h", "h", 100,0,100); // more useful

std::vector<int> v{1,4,5};
for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) ...
for (auto it = v.begin(); it != v.end(); ++it) ... //save typing!
```

Range based for loops

Simple way to loop over iterable types (stl container, root containers , c-arrays, ...).

```
std::vector<int> v{1,4,5};
for (const auto& val : v) std::cout << val << '\n';

float arr[]={4.,5.,6.};
for (const auto& val : arr) std::cout << val << '\n';

for (auto& val : arr) val*=val;
for (const auto& val : arr) std::cout << val << '\n';

for (auto val : arr) val*=val;
for (const auto& val : arr) std::cout << val << '\n';
```

`examples/cpp11/range_based.cpp`

Range based for loops

Can also be used with ROOT containers.

BUT: you have to know that a pointer to TObject is returned.

```
TClonesArray arr("TParticle");
for (int i=0; i<10; ++i) {
    TParticle& part = *static_cast<TParticle*>(arr.ConstructedAt(i));
    part.SetPdgCode(i);
}

for (auto o : arr) {
    TParticle& part = *static_cast<TParticle*>(o);
    std::cout << "Pdg code: " << part.GetPdgCode() << '\n';
}
```

examples/cpp11/range_based.cpp

nullptr

A specific *type* for null pointer. Before c++11 mainly 0, 0x0, NULL were used which is not type safe. Consider:

```
void print(int *i) { std::cout << "integer pointer: " << i << '\n'; }
void print(int i) { std::cout << "integer: " << i << '\n'; }

int main()
{
    print(0);
    print(NULL);
}
```

which will not compile

`examples/cpp11/nullptr_fail.cpp`

nullptr

nullptr solves this issue:

```
void print(int *i) { std::cout << "integer pointer: " << i << '\n'; }
void print(int i) { std::cout << "integer: " << i << '\n'; }

int main()
{
    print(0);
    print(nullptr);
}
```

will compile.

`examples/cpp11/nullptr.cpp`

Strongly typed enums

While names in normal enums cannot be the same

```
enum Animals {Bear, Cat, Chicken};  
enum Birds {Eagle, Duck, Chicken}; // error! Chicken has already been declared!
```

`examples/cpp11/enum_fail.cpp`

this is perfectly fine using strongly typed enums

```
enum class Fruits { Apple, Pear, Orange };  
enum class Colours { Blue, White, Orange }; // no problem!
```

`examples/cpp11/enum_strong.cpp`

Strongly typed enums

Also unintuitive that you could e.g. make bit operations between enums which are not strongly typed:

```
enum Animals {Bear, Cat};  
enum Birds {Eagle, Duck};  
  
bool b = Bear == Duck; // what?
```

Modern compilers should at least give a warning about this.

`examples/cpp11/enum_stupid.cpp`

Compilation fails with strongly typed enums

```
enum class Fruits { Apple, Pear, Orange };  
enum class Colours { Blue, White, Orange };  
  
bool b = Fruits::Orange == Colours::Orange; // what?
```

`examples/cpp11/enum_stupid_fail.cpp`

Strongly typed enums

You can specify the underlying integral type of C++11 enums:

```
enum class Foo : char { A, B, C };
```

In C++11 this works even for the 'normal' enums:

```
enum Bar : char { A, B, C};
```

override and final

Help the compiler to find errors in inheritance with function overriding.

```
class Foo {
public:
    virtual void print(const char* something) const final {
        std::cout << "Foo prints " << something << '\n';
    }
};

class Bar : public Foo {
public:
    void print(const char* something) const override {
        std::cout << "Bar prints " << something << '\n';
    }
};

int main ()
{
    Bar bar;
    Foo &foo = bar;
    foo.print("hi");
}
```

Use `final` if you don't intend a function to be overridden.

move semantics

C++11 has introduced the concept of rvalue references (specified with `&&`) to differentiate a reference to an lvalue or an rvalue. An lvalue is an object that has a name, while an rvalue is an object that does not have a name (a temporary object). The move semantics allow modifying rvalues (previously considered immutable and indistinguishable from `const T&` types).

```
class Foo {  
    public:  
        Foo(int i) : val(i) {}  
        std::vector<int> val;  
};  
  
Foo getBigFoo(i) { Foo foo(100000); return foo; }  
  
int main() {  
    Foo foo = getBigFoo();  
}
```

The compiler implements a move operator

```
Foo(Foo&& other) ...  
const Foo& operator= (Foo&& other) ...
```


move semantics

c++11 implements `std::move` which explicitly

```
std::string foo = "foo-string";
std::string bar = "bar-string";
std::vector<std::string> myvector;

myvector.push_back (foo);           // copies
myvector.push_back (std::move(bar)); // moves

std::cout << "foo is: " << foo << '\n';
std::cout << "bar is: " << bar << '\n';

std::cout << "myvector contains:";
for (const auto& x:myvector) std::cout << ' ' << x;
std::cout << '\n';
```

`examples/cpp11/move_string.cpp`

default and delete

The compiler will always try to implement e.g. a copy and assignment constructor as well as its move counterpart. This can be forced as well as forbidden:

```
class Foo {  
    Foo():Foo() = default;  
    Foo(const Foo&) = delete;  
    Foo(Foo&&) = delete;  
    Foo& operator=(const Foo&) = default;  
    Foo& operator=(Foo&&) = delete;  
};
```

if you define 1 the other 5 will be automatically deleted! then you need to implement all 5.

Lambdas

A lambda is a nameless function with special properties:

- ▶ first class citizen - can be assigned to variables, passed to functions etc.
- ▶ can capture (i.e. copy or reference) variables from the outside scope irrespective of it's signature.

```
int x{4};  
int y{5};  
  
auto mylambda = [x,&y] (auto in, const auto& alsoin) {  
    return in + alsoin - x - y;  
};  
  
auto result = mylambda(3,7);
```

Lecture VII

Smart Pointers

pointers

- ▶ Pointers are a basic feature of C / C++
- ▶ Both one of the most powerfull and most dangerous features
- ▶ The majority of errors / bugs / security is related to memory management
- ▶ Use after free, buffer overflows are the most prominent cases
- ▶ (Semi-) Automatic memory management is desired, without loosing flexibility.

pointers

- Ownership problem

```
Particle* p = new Particle;  
Collection* c = new Collection;  
c->Add(p);
```

- Traditionally, containers need a flag whether they own or not?
- This is asking for trouble

pointers

- ▶ The solution in C++11: smart pointers

```
#include<memory>  
std::unique_ptr<int>  
std::shared_ptr<int>
```

- ▶ lifetime of managed resource managed by scoping (unlike raw pointers).
- ▶ The unique ptr is THE unique ptr pointing to the object, there cannot be a second one.
- ▶ Multiple shared pointers can point to an object, ownership is handled automatically, the last shared pointer that is destroyed also destroys the object.

pointers

- ▶ There are some limitations by design:
- ▶ Unique pointers cannot be copied or assigned (unique ownership!)

```
std::unique_ptr<int> p1(new int);  
std::unique_ptr<int> p2 = p1; //DOES NOT COMPILE!
```

- ▶ Operations like this require an explicit move (`std::move`) - The programmer should know what he does.

`examples/smart_pointers`

pointers in interfaces

- ▶ using raw pointers is still OK - just don't use "owning raw pointers"
- ▶ raw pointer only points at something which is guaranteed to exist (or use a reference).
- ▶ managed pointer signals ownership (transfer).

```
void function( int* look, std::unique_ptr<Particle> consume) {}
```

Lecture VIII

Standard Template Library

Template programming

Template programming is a different programming technique which allows to provide common functionality to many different data structures.

In this lecture we provide as much knowledge as it is needed to be able to work with the Standard Template Library (STL) functionality.

A template defines and implements a family of functions/classes with open data types.

```
template T class dummy {  
    public:  
        dummy();  
        const T& get_a();  
    protected:  
        T a;  
};
```

A template class or function can be used with any data type fulfilling the requirements. The compiler creates the specialized version for a specific data type.

Motivation - an example

Suppose there is an array of integers

```
const int size=10;
int array[10]={7,5,6,2,3,1,4,0,9,8};
```

An algorithm to *find* the position of an element can be

```
int *first=&array[0], *last=&array[10];
int value=7;
while (first != last && *first != value)
    ++first;
return first;
```

In a function it looks like

```
int* find(int* first, int* last, const int& value) {
    while (first != last && *first != value) ++first;
    return first;
}
```

Suppose, it's also needed for data type float

```
float* find(float* first, float* last, const float& value) {
    while (first != last && *first != value) ++first;
    return first;
}
```

Motivation - an example

Suppose there is an array of integers

```
const int size=10;
int array[10]={7,5,6,2,3,1,4,0,9,8};
```

An algorithm to *find* the position of an element can be

```
int *first=&array[0], *last=&array[10];
int value=7;
while (first != last && *first != value)
    ++first;
return first;
```

In a function it looks like

```
int* find(int* first, int* last, const int& value) {
    while (first != last && *first != value) ++first;
    return first;
}
```

Suppose, it's also needed for data type float

```
float* find(float* first, float* last, const float& value) {
    while (first != last && *first != value) ++first;
    return first;
}
```

Motivation - an example

Suppose there is an array of integers

```
const int size=10;
int array[10]={7,5,6,2,3,1,4,0,9,8};
```

An algorithm to *find* the position of an element can be

```
int *first=&array[0], *last=&array[10];
int value=7;
while (first != last && *first != value)
    ++first;
return first;
```

In a function it looks like

```
int* find(int* first, int* last, const int& value) {
    while (first != last && *first != value) ++first;
    return first;
}
```

Suppose, it's also needed for data type float

```
float* find(float* first, float* last, const float& value) {
    while (first != last && *first != value) ++first;
    return first;
}
```

Generalization

The two algorithms are identical, the functions only differ in the type
⇒ here the concept of templates helps to generalize the function

```
template<class T>
T* find(T* first, T* last, const T& value) {
    while (first != last && *first != value) ++first;
    return first;
}
```

Even more general, the type of the pointers does not need to be the same as the type of the value to search for

```
template<class Iterator, class T>
Iterator find(Iterator first, Iterator last, const T& value) {
    while (first != last && *first != value) ++first;
    return first;
}
```

The requirements to make this work are

- ▶ Iterator must support the prefix increment operator
- ▶ Iterator must support the !=-comparison operator
- ▶ dereferencing Iterator must give the type of the search value

Simplifying even further...

With C++14 we can even do:

```
auto find(auto first, auto last, const auto& value) {  
    while (first != last && *first != value) ++first;  
    return first;  
}
```

With the same requirements as before

Important to know

- ▶ the template class or function has to be implemented in the header file

A layout consisting of header file containing the function/class definition

```
template<class Iterator, class T>
Iterator find(Iterator first, Iterator last, const T& value);
```

and a source file with the implementation

```
template<class Iterator, class T>
Iterator find(Iterator first, Iterator last, const T& value) {
    while (first != last && *first != value) ++first;
    return first;
}
```

DOES NOT WORK!

The Standard Template Library

A set of C++ template classes to provide common programming data structures and functions:

- ▶ Container classes
<http://en.cppreference.com/w/cpp/container>
- ▶ iterator:
 - ▶ represent position in a container
 - ▶ declared to be associated with a single container class type
- ▶ algorithm:
 - ▶ routines to find, count, sort, search, ... elements in container classes

STL Containers

- ▶ Sequences:
 - ▶ vector: Dynamic array. Insert data at the end.
 - ▶ deque: Dynamic array. insertion/removal at beginning or end
 - ▶ list: linked list. Insert/remove anywhere.
- ▶ Associative Containers:
 - ▶ set: Collection of ordered data (no duplication). Fast search.
 - ▶ multiset: Collection of ordered data (duplication allowed). Fast search.
 - ▶ map: Collection of associative key-value pair with unique keys
 - ▶ multimap: Collection of associative key-value pair, duplicate keys allowed
- ▶ Container adapters:
 - ▶ stack LIFO
 - ▶ queue FIFO
 - ▶ priority_queue returns element with highest priority.
- ▶ String:
 - ▶ string: Character strings and manipulation
 - ▶ rope: String storage and manipulation
- ▶ bitset: intuitive method of storing and manipulating bits.

Example - the vector container

similar to an array, handles automatically its own storage requirements in case it grows

```
#include <vector>
using std::vector;

vector<int> v;
```

Basic operations:

push_back	Add element to end of collection.
pop_back	Remove element at end of collection
back	Get a reference to element at end of collection
front	Get a reference to element at end of collection
operator []	Access specified element

Note: the vector keeps the internally allocated memory also if the number of elements is reduced or all elements are erased.

Using a vector container

<code>empty</code>	determines if the collection is empty
<code>size</code>	number of elements in the collection
<code>capacity</code>	number of elements which can be added without growing the internal storage
<code>begin</code>	forward iterator pointing to the start of the collection
<code>end</code>	forward iterator pointing to one past the end of the collection
<code>rbegin</code>	backward iterator pointing to the end of the collection
<code>rend</code>	backward iterator pointing to one before the start of the collection
<code>clear</code>	erases all elements in a collection. Note: pointers must be deleted manually
<code>erase</code>	erase element or range of elements from a collection

STL containers vs. ROOT containers

ROOT also provides container classes, e.g. `TObjArray`, `TClonesArray`, `TList`.

Polymorphism is used to implement the ROOT collection classes

- ▶ every class needs to inherit from `TObject`
- ▶ collection classes only know about `TObject`
 - ⇒ every element return by access functions can only be of type `TObjArray`
 - ⇒ type casts are necessary

STL containers are type-safe because of the template approach

STL algorithms

- ▶ <https://en.cppreference.com/w/cpp/algorithm>
- ▶ expressive: say WHAT you mean to do, not HOW
- ▶ part of standard: in many compilers are allowed special optimizations.

e.g. instead of a (range-based) for loop:

```
std::vector<int> nums{3, 4, 2, 8, 15, 267};

int size{0};
std::for_each( nums.begin(), nums.end(), [&size] (auto& n) {
    ++n; ++size;
});
```

STL algorithms

- ▶ <https://en.cppreference.com/w/cpp/algorithm>
- ▶ expressive: say WHAT you mean to do, not HOW
- ▶ part of standard: in many compilers are allowed special optimizations.

or (a pretty generic) sort:

```
std::vector<int> nums{3, 4, 2, 8, 15, 267};

int ops{0};
std::sort( nums.begin(), nums.end(), [&ops] (const auto& a, const auto& b) {
    ++ops; return a < b;
});
```


Further reading

<http://en.cppreference.com/w/cpp/container>

<http://en.cppreference.com/w/cpp/algorithm>

`examples/template`

Lecture IX

More on gcc

Include files

- ▶ When dealing with larger code bases or external libraries, *header files* are used to define the interface of functions and classes
- ▶ During the compilation process the compiler needs to know where to look for header files
- ▶ This can be steered in two ways:
 - ▶ A compiler option
 - ▶ An environment variable
- ▶ A few system directories are included by default like
 - ▶ /usr/include
 - ▶ /usr/local/include

Include files - search path

compiler flag

- ▶ Search paths can be added using the `-I` compiler option
- ▶ One statement is needed per path
- ▶ No space between `-I` and the path
- ▶ With GCC, you can use `-isystem` instead of `-I` to suppress warning in include files.

```
g++ -I. -I/path/one -I/path/two ...
```

environment variable

- ▶ `C_INCLUDE_PATH`
- ▶ `CPLUS_INCLUDE_PATH`
- ▶ colon separated list of paths

```
export CPLUS_INCLUDE_PATH=./path/one:/path/two
```

Include files - search path

compiler flag

- ▶ Search paths can be added using the `-I` compiler option
- ▶ One statement is needed per path
- ▶ No space between `-I` and the path
- ▶ With GCC, you can use `-isystem` instead of `-I` to suppress warning in include files.

```
g++ -I. -I/path/one -I/path/two ...
```

environment variable

- ▶ `C_INCLUDE_PATH`
- ▶ `CPLUS_INCLUDE_PATH`
- ▶ colon separated list of paths

```
export CPLUS_INCLUDE_PATH=./path/one:/path/two
```

Include files - search path - example

- ▶ Create a subdirectory 'header'
- ▶ Move the file *hello_fn.h* there
- ▶ Try to compile the code again

⇒ Try it!

Try both possibilities to declare a search path:

```
g++ -Wall -Iheader main.cpp hello_fn.cpp -o hello_world
```

```
export CPLUS_INCLUDE_PATH=header  
g++ -Wall main.cpp hello_fn.cpp -o hello_world
```

Include files - search path - example

- ▶ Create a subdirectory 'header'
- ▶ Move the file *hello_fn.h* there
- ▶ Try to compile the code again

⇒ Try it!

Try both possibilities to declare a search path:

```
g++ -Wall -Iheader main.cpp hello_fn.cpp -o hello_world
```

```
export CPLUS_INCLUDE_PATH=header  
g++ -Wall main.cpp hello_fn.cpp -o hello_world
```

Libraries

- ▶ Object files can be combined in so-called *libraries*
- ▶ Two different kinds of libraries exist
 - ▶ Static libraries (ending on .a)
 - ▶ Objects needed for the code execution are copied into the executable
 - ▶ Increases the code size
 - ▶ If several programs use the same static libraries, they all load the same code into memory during execution time
 - ▶ Dynamic libraries (ending on .so)
 - ▶ Are linked against the program during compilation time
 - ▶ Are loaded into memory during the program execution
 - ▶ Does not increase the code size
 - ▶ The code in memory can be shared among all programs requiring it

Libraries

- ▶ Object files can be combined in so-called *libraries*
- ▶ Two different kinds of libraries exist
 - ▶ Static libraries (ending on `.a`)
 - ▶ Objects needed for the code execution are copied into the executable
 - ▶ Increases the code size
 - ▶ If several programs use the same static libraries, they all load the same code into memory during execution time
 - ▶ Dynamic libraries (ending on `.so`)
 - ▶ Are linked against the program during compilation time
 - ▶ Are loaded into memory during the program execution
 - ▶ Does not increase the code size
 - ▶ The code in memory can be shared among all programs requiring it

Libraries

- ▶ Object files can be combined in so-called *libraries*
- ▶ Two different kinds of libraries exist
 - ▶ Static libraries (ending on `.a`)
 - ▶ Objects needed for the code execution are copied into the executable
 - ▶ Increases the code size
 - ▶ If several programs use the same static libraries, they all load the same code into memory during execution time
 - ▶ Dynamic libraries (ending on `.so`)
 - ▶ Are linked against the program during compilation time
 - ▶ Are loaded into memory during the program execution
 - ▶ Does not increase the code size
 - ▶ The code in memory can be shared among all programs requiring it

Libraries

- ▶ As for include paths, g++ also needs to know where to search for the libraries when creating an executable
- ▶ This can be steered in two ways
 - ▶ A compiler option
 - ▶ An environment variable
- ▶ A few system directories are included by default like
 - ▶ /usr/lib
 - ▶ /usr/local/lib

Libraries - search paths

compiler flag

- ▶ Search paths can be added using the `-L` compiler option
- ▶ One statement is needed per path

```
g++ -L/path/one -L/path/two ...
```

environment variable

- ▶ `LIBRARY_PATH` - static libs
- ▶ `LD_LIBRARY_PATH` - dynamic libs
- ▶ colon separated list of paths

```
export LIBRARY_PATH=/path/one:/path/two
```

Libraries - search paths

compiler flag

- ▶ Search paths can be added using the `-L` compiler option
- ▶ One statement is needed per path

```
g++ -L/path/one -L/path/two ...
```

environment variable

- ▶ `LIBRARY_PATH` - static libs
- ▶ `LD_LIBRARY_PATH` - dynamic libs
- ▶ colon separated list of paths

```
export LIBRARY_PATH=/path/one:/path/two
```

Libraries - dynamic libraries

- ▶ Dynamic libraries are not compiled into the code
- ▶ They need to be loaded during the run time, when the program is executed
- ▶ For this the *dynamic linker* needs to know where to search for the libraries
- ▶ If dynamic libraries don't reside in the 'default' paths of the system it has to be declared using the `LD_LIBRARY_PATH` environment variable

Libraries - an example

create objects

```
g++ -fPIC -c hello_fn.cpp main.cpp
```

NOTE: -fPIC is required to prepare for dynamic libraries!

link function object into dynamic library

```
g++ -shared hello_fn.o -o libHello.so
```

create executable and bind library

```
g++ -o hello_dynamic main.cpp -L. -lHello
```

Libraries - an example

create objects

```
g++ -fPIC -c hello_fn.cpp main.cpp
```

NOTE: -fPIC is required to prepare for dynamic libraries!

link function object into dynamic library

```
g++ -shared hello_fn.o -o libHello.so
```

create executable and bind library

```
g++ -o hello_dynamic main.cpp -L. -lHello
```


Libraries - an example

create objects

```
g++ -fPIC -c hello_fn.cpp main.cpp
```

NOTE: -fPIC is required to prepare for dynamic libraries!

link function object into dynamic library

```
g++ -shared hello_fn.o -o libHello.so
```

create executable and bind library

```
g++ -o hello_dynamic main.cpp -L. -lHello
```

Warning options

-Wall

- ▶ Warnings for most common errors; We suggest to always use this
 - ▶ -Wcomment, -Wformat, -Wunused, -Wimplicit, -Wreturn-type, . . .
 - ▶ Can also be activated individually
- ▶ Includes many sub-switches
- ▶ Always use at least this one
- ▶ Consider using other warning options

Some examples

- ▶ Check non-void functions return a value

```
int myFunction() { float x=3.; }
```

- ▶ Check comparison between signed/unsigned

```
int i = 8;  
unsigned int j = 8;  
if ( i == j ) {}
```

Some specific warning options

- ▶ `-Wconversion` check implicit type conversion

```
unsigned int x = -1;
```

- ▶ `-Wshadow` check if a variable declaration can shadow another one

```
int x=1; {int x=5;}
```

- ▶ `-Wwrite-strings` don't allow non const c-strings

- ▶ ...

- ▶ `-Winconsistent-missing-override` and many many more

<https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>

Compiler optimisation

- ▶ GCC is an optimising compiler
- ▶ Optimisation could be
 - ▶ faster execution
 - ▶ smaller executable (memory stamp)
 - ▶ → speed-space tradeoff

Compiler optimisation levels

- ▶ `-O0` or none (default)
No optimisation, compilation in most straight-forward way, best for debugging
- ▶ `-O1` or `-O`
 - ▶ Most optimisations without speed-space tradeoff
 - ▶ Should result in smaller executable AND faster code
 - ▶ Might also compile faster
- ▶ `-O2`
 - ▶ More optimisation than `-O1`, still without speed-space tradeoff
 - ▶ Should result in faster code without and increase in size
 - ▶ Compilation takes longer and needs more memory
- ▶ `-O3`
 - ▶ Even more than `-O1`, `-O2`
 - ▶ Should result in faster code, might increase the executable size
 - ▶ Compilation takes longer and needs more memory

Compiler optimisation levels

- ▶ `-ffast-math`
 - ▶ Floating point operations on the CPU are non-associative
 $(x*(y+z) \neq x*y + x*z)$
 - ▶ Therefore the compiler usually does not perform math simplifications of your code
 - ▶ **-ffast-math** tells enables to compiler to do so but the numerical result might be different than without `-O2`
 - ▶ Fast math also disables some error treatment and assumes that basically all of your calculations are well defined.
 - ▶ Can really improve the speed

[link to discussion on stackoverflow](#)

Compiler optimisation levels

- ▶ Unroll loops: `-funroll-loops`
 - ▶ There is a trade-off: Less control-flow and fewer jumps in the code.
 - ▶ Larger code, thus worse cache utilization.
 - ▶ The compiler cannot always determine the optimal unroll level.
 - ▶ One can try manual unrolling (see example).
- ▶ Similar: `-finline-functions`.
- ▶ Disabled with `-fno-unroll-loops -fno-inline-functions`.

```
int x[3];  
for (int i=0; i<3;++i){  
    x[i]=i*i;  
}
```

```
int x[3];  
x[0]=0;  
x[1]=1;  
x[2]=4;
```

- ▶ `-Os`
 - ▶ Optimise for size (Optimizes also cache, Windows is compiled like this)
 - ▶ Use most optimisations also used in `-O2` (that don't increase the size)

[examples/performance/09_unroll/](#)

⇒ Try it!

List of some important compiler options

- ▶ Select the correct CPU architecture
 - ▶ New processors provide new more powerful instructions.
 - ▶ Compilers generate compatible code for all architectures.
 - ▶ One can enable additional sets of instructions with `-msse`, ..., `-msse41`, `-m3dnow`, `-mavx`.
 - ▶ Use `-march=ARCHITECTURE` flag to select target architecture (with instruction set).
 - ▶ ARCHITECTURE can be e.g. `i386`, `pentium`, `nocona`, `opteron`, **native**.
- ▶ `-fomit-frame-pointer`
 - ▶ Enabled with `-O2`, makes one additional CPU register available for optimizations.
 - ▶ Complicates access to stack frame, sometimes prevents creation of backtraces in GDB, valgrind, perf.
 - ▶ Can be disabled for debugging / profiling:
`-fno-omit-frame-pointer`.

List of some important compiler options

- ▶ `-o <executable name>` name the executable default is `a.out`
- ▶ `-g -ggdb` add debugging symbols
- ▶ `-O<level>` code optimization with different levels
- ▶ `-W<xxx>` compiler warnings
- ▶ `-c` create objects instead of an executable
- ▶ `-I<path>` add search path for include file
- ▶ `-L<path>` add search path for libraries
- ▶ `-l<LIB>` link `lib<LIB>.so` into the binary
- ▶ `-ansi` disable GNU C language extension that conflict with the ANSI/ISO C standard
- ▶ `-pedantic` used with `-ansi` ALL GNU C language extensions are disabled
- ▶ `-std=...` select a special c-standard (e.g. C99, `c++11`, `gnu++11`, `c++14`)

Remember

- ▶ Never compile code without gcc warnings enabled
- ▶ Take ALL warnings seriously
- ▶ Fix ALL warnings

Godbolt - Playground

A demonstration with godbolt.org

Lecture X

ROOT - Intro / Libs / IO / Trees

ROOT

Some info about ROOT

- ▶ root.cern.ch: A modular scientific (C++) software toolkit. It provides all the functionalities needed to deal with big data processing, statistical analysis, visualisation and storage.
- ▶ Used by most high energy physics experiments (e.g. at CERN, GSI, BNL, Fermilab, neutrino experiments, ...)
- ▶ Allows for simple code prototyping with a just-in-time compiler Cling (ROOT 6)
- ▶ Allows for C++ introspection and automatic serialization of any C++ class

ROOT : Our interest/scope

ROOT is vast and goes beyond the goal of this course. The features that we want to highlight here

- ▶ ROOT as a library: How to use some of its functionality
- ▶ ROOT as IO/serialization mechanism and data inspector
 - ▶ understand creation of dictionaries
 - ▶ understand TTree storage
 - ▶ understand data inspection from a TTree / TBrowser

ROOT as a library

ROOT has lot's of service classes which one could use (in your project) to get quickly started. The usage is the normal **include** / **use** / **link**.

```
#include <TRandom3.h>    // include a ROOT random number generator
#include <iostream>

int main() {
    TRandom3 generator;
    // get a poissonian distributed random number around 5
    std::cout << generator.Poisson(5) << std::endl;
}
```

`root-config` can be used to get include path and libraries:

```
g++ -O2 -I`root-config --incdir` -c main.cxx
g++ main.o `root-config --libs` -o main.exe
```

`examples/rootaslib`

Bonus: ROOT and cmake

- ▶ A minimal example how to add ROOT include paths and libraries in cmake
- ▶ more docu here
root.cern.ch/how/integrate-root-my-project-cmake

```
cmake_minimum_required(VERSION 3.0 FATAL_ERROR)
project(myproject)
find_package(ROOT REQUIRED)
include_directories(${ROOT_INCLUDE_DIRS})

#---Create the main program using the library
add_executable(main.exe main.cxx)
target_link_libraries(main.exe ${ROOT_LIBRARIES})
```

examples/rootaslib

ROOT as a library

What could be useful at first for your project

- ▶ random numbers
- ▶ random numbers from arbitrary distributions using TF1 class
- ▶ 3-body decay simulation provided by TGenPhasePhase

However, sometimes these implementations may be

- ▶ too generic hence not as performant as specialized code
- ▶ not thread-safe preventing parallelisation
- ▶ (not compilable on GP-GPU etc)
- ▶ superceeded by modern C++11 constructs

so that one needs to refactor (later on).

- ▶ It is generally no longer advised to use ROOT containers such as TList, TArray, TMatrix
- ▶ It is also no longer necessary to derive any class from TObject

Introduction to ROOT serialization and TTrees

The killer-feature of ROOT is the ability to serialize **any** C++ class, i.e., write it to disc in binary form and reinstantiate the object later on from disc, at almost no developer cost.

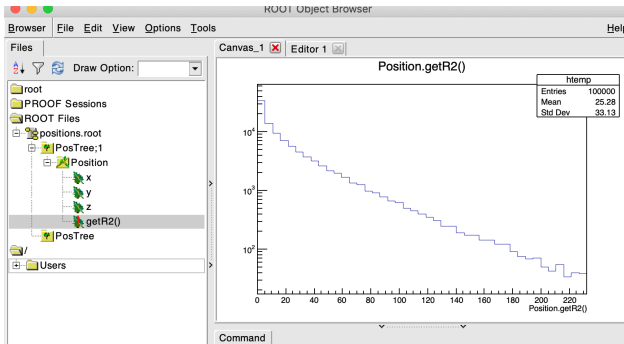
```
class Position {  
public:  
    Position() = default;  
  
    float getR2() const;  
  
private:  
    float x = 0;  
    float y = 0;  
    float z = 0;  
};
```

serialize



read

visualize
analyze



ROOT serialization example

Example writing a user object to a TFile and read it back in:

```
// In Foo.h
struct Foo {
    std::string name;
    float x = 0;
    double y = 0;
};
```

```
// in the main program
int main() {
    TFile file("outfile.root", "RECREATE");
    Foo f("hello", 1.5, 2.); // make a Foo instance

    // write to file using some key name
    file.WriteObjectAny((void*)f, TClass::GetClass(typeid(Foo)), "key");

    // read back from file
    Foo* anotherfoo = nullptr;
    file.GetObject("key", anotherfoo);
    return 0;
}
```

ROOT dictionaries

- ▶ in order for this to work we only need to create a dictionary for every class we want to write out
- ▶ The dictionary is an autogenerated shared library containing information about your class
- ▶ This needs a LinkDef.h file mentioning your class:

```
#ifndef __CLING__  
#pragma link off all classes;  
#pragma link off all functions;  
#pragma link off all globals;  
  
#pragma link C++ class Foo+;  
#endif
```

- ▶ the dictionary is created in 2 steps

```
rootcling Foo.h LinkDef.h -f FooDict.cxx  
g++ -shared -o libFooDict.so `root-config --ldflags` -O2 FooDict.cxx
```

- ▶ See code example for easy ways how this is done with cmake

[examples/basicrootio/write_read/](#)

Serialization/storage with TTrees

- ▶ **TTree** (or TBranch) is a **special container** used to serialize **many instances** of your class to disc
- ▶ Serialization is done very efficiently, i.e. using compression
- ▶ Serialization is done for each data member separately (SoA) - columnar data format
- ▶ This allows access to single data members without the need to read all data
- ▶ Easy way to quickly analyse and visualize data stored (TTree::Draw())

`examples/basicrootio/trees_userclass/`

`examples/trees`

A simple example.C : Fill the TTree container

```
TF1 fpt("fpt", "<mycomplex function>", 0.1, 100);

TFile f("output.root", "recreate");
TTree t("tree", "tree");
Float_t z=0.;
TLorentzVector *v=new TLorentzVector;

t.Branch("z", &z);
t.Branch("particle", &v);

for (Int_t ientry=0; ientry<10000; ++ientry) {
    z=gRandom->Gaus(0,5);
    Double_t phi = gRandom->Uniform(0.0, TMath::TwoPi());
    Double_t eta = gRandom->Uniform(-1, 1);
    Double_t pt = fpt.GetRandom();
    v->SetPtEtaPhiM(pt, eta, phi, .14);

    t.Fill();
}

f.Write();
f.Close();
delete v;
```

Short discussion about the program

- ▶ Trees can be associated to a file:
 - ▶ First open a file in write mode
 - ▶ Then create the tree
 - ▶ This ensures automatic saving of the data to file and frees the memory
- ▶ It is simple to add branches for primitive type and also complex classes (see user example)

Filling a tree

- ▶ Fill all the data members you associated with the branches
- ▶ Calling the Fill function then dumps all the values of the objects into the tree structure

Visualizing data in TTrees

- ▶ data in TTrees can be inspected in the `TBrowser`
- ▶ data in TTrees can be inspected in the `TTreeView`
- ▶ using directly the underlying `TTree::Draw(what, cut)` interface
 - ▶ goes over all data described in `what` under certain selection criteria `cut`
 - ▶ produces a histogram presenting the selected data or some other higher dimensional plot
 - ▶ see <https://root.cern.ch/doc/master/classTTree.html>

Demo

- ▶ compile and run the mentioned code example below; will generate a ROOT file containing some positions in a TTree
- ▶ TBrowser (click through)

```
> root positions.root  
root [0] new TBrowser  
  (TBrowser *) 0x7fc61cc077d0  
root [1]
```

- ▶ direct drawing from the TTree (tree name automatically known to ROOT)

```
> root positions.root  
root [2] PosTree->Draw("Position.x", "Position.y > 0.6")  
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1  
(long long) 39755
```

[examples/basicrootio/trees_userclass/](#)

Reading data (read.C) from a TTree using GetEntry

```
TH1F *hPt=new TH1F("hPt","p_{T}; p_{T} (GeV/c)", 100,0,10);
TH1F *hZ=new TH1F("hZ","z; z (cm)", 100,0,10);
TFile f(filename);
gROOT->cd();

TTree *t=(TTree*)f.Get("tree");

Float_t z=0.;
TLorentzVector *v=nullptr;

t->SetBranchAddress("z",&z);
t->SetBranchAddress("particle",&v);

for (Int_t iev=0; iev<t->GetEntries(); ++iev) {
    t->GetEntry(iev);
    hPt->Fill(v->Pt());
    hZ->Fill(z);
}

delete v;
```

More on Branches

- ▶ There are many different ways to create a branch, have a look at the documentation
- ▶ It even allows full C++ incl. STL, class does not need to derive from TObject

```
TBranch* Branch(const char* name, void** obj,  
               Int_t bufsize = 32000)
```

Two important things

- ▶ bufsize is the size of the branch buffer in memory before it is written to file
 - ▶ Can be used to optimize the I/O
 - ▶ Too small numbers might result in many writes of small data chunks
 - ▶ Too large numbers might fill the memory too quickly
- ▶ If complex objects contain 'sub-objects' they get assigned their own branch

Your exercise

- ▶ Understand the examples
- ▶ Try writing your own class to disc / TTree
- ▶ Play around with TTree::Draw, reading of data, ...
- ▶ Play around with the TBuffer buffer size, check execution time using a simple timer (ROOT offers TStopWatch)

Lecture XI

Code documentation using doxygen

Code documentation

What you have to keep in mind:

- ▶ You or somebody else wants to understand your code, especially after some time when it needs to be extended or a bug needs to be fixed
- ▶ You should use a consistent documentation style
- ▶ Better add more comments

A short introduction to get you started with **doxygen**

For further reading [doxygen](#) [dokuo](#)

Running Doxygen

To generate a manual for your project you typically need to follow these steps

1. document your source code with special documentation blocks
2. generate a configuration file by calling doxygen with the -g option:

```
doxygen -g <config_file>
```

3. edit the configuration file to match the project
4. run doxygen to generate the documentation

```
doxygen <config_file>
```

Sample session:

In the examples/tooling directory:

1. fetch the latest version of the example

```
git pull
```

2. Generate the documentation

```
doxygen -g doxygen.conf
```


Doxygen - The configuration file

The operation can be tuned by many options in the config file, all of them are described in the generated template. The name of the project is likely to be adjusted.

```
# The PROJECT_NAME tag is a single word (or sequence of words) that should  
# identify the project. Note that if you do not use Doxywizard you need  
# to put quotes around the project name if it contains spaces.
```

```
PROJECT_NAME           = "A simple test project"
```

Also the input, i.e. source and header files, can be adjusted,

```
#-----  
# configuration options related to the input files  
#-----  
# The INPUT tag can be used to specify the files and/or directories that contain  
# documented source files. You may enter file names like "myfile.cpp" or  
# directories like "/usr/src/myproject". Separate the files or directories  
# with spaces.
```

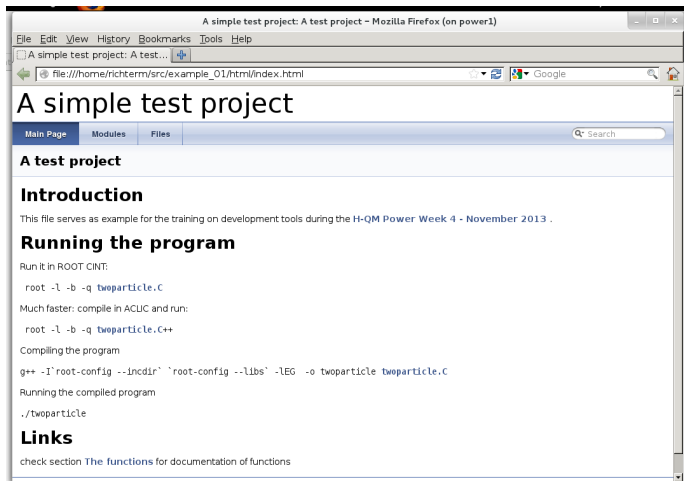
```
INPUT                  =
```

```
# If the value of the INPUT tag contains directories, you can use the  
# FILE_PATTERNS tag to specify one or more wildcard pattern (like *.cpp  
# and *.h) to filter out the source-files in the directories. If left  
# blank the following patterns are tested:  
# *.c *.cc *.cxx *.cpp *.c++ *.d *.java *.ii *.ixx *.ipp *.i++ *.inl *.h *.hh  
# *.hxx *.hpp *.h++ *.idl *.odl *.cs *.php *.php3 *.inc *.m *.mm *.dox *.py  
# *.f90 *.f *.for *.vhd *.vhdl
```

Build the documentation and open it

```
richterm@power1 ~/src/example_01 $ doxygen doxygen.conf
```

```
firefox file://$PWD/html/index.html
```



Doxygen - Special comment blocks

C or C++ style comment block with some additional markings parsed by doxygen

1. Javadoc style: C-style comment block starting with two `*`'s

```
/**
 * ... text ...
 */
```

- ## 2. Qt style

```

/*!
 * ... text ...
 */

```

3. block of at least two C++ comment lines, where each line starts with an additional slash or an exclamation mark

```

///
/// ... text ...
///

//!
//! ... text ...
//!
```

twoparticle.C documentation - copyright notice

The code should have a copyright information containing

- ▶ the author name/group
- ▶ terms of usage

```

/*****
/* This file is property of and copyright by ...
/* All rights reserved.
/*
/* Primary Authors: Matthias Richter <Matthias.Richter@ift.uib.no>
/*
/* Permission to use, copy, modify and distribute this software and its
/* documentation strictly for non-commercial purposes is hereby granted
/* without fee, provided that the above copyright notice appears in all
/* copies and that both the copyright notice and this permission notice
/* appear in the supporting documentation. The authors make no claims
/* about the suitability of this software for any purpose. It is
/* provided "as is" without express or implied warranty.
*****/

```

There are many different open source license templates.

twoparticle.C documentation - Overall description

```
/** @file twoparticle.C
    @author Matthias Richter
    @date 2013-11-25
    @brief A simple macro with surprises for the purpose of valgrind and gdb training
*/

/** @mainpage A test project

    @section intro Introduction
    This file serves as example for the training on development tools during the
    <a class="el" href="http://fias.uni-frankfurt.de/helmholtz/program/power-week/index-4.shtml">
    H-QM Power Week 4 - November 2013 </a> .

    @section execution Running the program
    Run it in ROOT CINT:
    <pre> root -l -b -q twoparticle.C </pre>
    Much faster: compile in ACLIC and run:
    <pre> root -l -b -q twoparticle.C++</pre>

    Compiling the program
    <pre>g++ -I`root-config --incdir` `root-config --libs` -LEG -std=c++17 -o twoparticle twoparticle.C</pre>
    Running the compiled program
    <pre>./twoparticle</pre>

    @section Links Links
    check section @ref functions for documentation of functions
*/
```

examples/doxygen

twoparticle.C documentation - Function documentation

```
/**
 * Correlate two particles
 * @param trigger      trigger particle
 * @param associated    associated particle
 * @param hDeltaphiDist histogram for delta phi 1D distribution
 * @param hDeltaPhiVsDeltaEta histogram for delta phi - delta eta 2D distribution
 * @return 0 on success
 * @ingroup functions
 */
int Correlate(const TParticle trigger, const TParticle associated, TH1 *hDeltaphiDist, TH2* hDeltaPhiVsDeltaEta) {
    /**
     * Pi constant to be used within the calculations
     */
    const Float_t Pii = 3.14159;
```

What next?

⇒ Try it!

add documentation to your project

Important keywords:

- ▶ @mainpage
- ▶ @section
- ▶ @ref
- ▶ @file
- ▶ @defgroup
- ▶ @ingroup
- ▶ @file
- ▶ @class
- ▶ @param
- ▶ @return

Lecture XII

Libraries

Command-line parameters

- ▶ common and frequent problem:
parsing of command line options
- ▶ based on array of strings passed from OS
- ▶ different approaches:
 - ▶ write your own parsing
 - ▶ getopt
 - ▶ boost program options
 - ▶ ...
- ▶ usually you want to provide a consistent help
(I forget the options I implemented yesterday)

main prototype

- ▶ every executable needs a main function as an entry point
- ▶ it can have
 - ▶ no arguments:
`int main()`
 - ▶ two arguments:
`int main(int argc, char *argv[])`
(or equivalent)
 - ▶ argc contains the number of arguments
 - ▶ first argument is the name of the executable
 - ▶ argv is an array of C-strings,
last string is 0

getopt

- ▶ C library function
very wide-spread, e.g. also on embedded systems
- ▶ documentation must be synchronised manually
- ▶ let's look at a short example,
then try and get it running

`examples/getopt`

getopt – example

```
int main(int argc, char *argv[]) {
    static struct option long_options[] = {
        {"file", required_argument, 0, 'f'},
        {0, 0, 0, 0}
    };
    char c; int option_index = 0;
    while (1) {
        c = getopt_long(argc, argv, "f:", long_options, &option_index);
        if (c == -1)
            break;
        switch (c) {
            case 'f':
                // remember filename
                break;
            default:
                // unrecognized option
        }
    }
}
```

boost program options

- ▶ boost is a collection of C++ libraries for various purposes, e.g. CRC calculation, configuration file reading, and program options
- ▶ boost makes extensive use of templating and meta programming, fine with recent compilers, problematic with old ones
- ▶ limited to C++
- ▶ far more features and more C++-like formulation
- ▶ let's look at a short example, then try and get it running

`examples/boost_option`

boost program options – example

```
#include <iostream>
#include <boost/bind.hpp>
#include <boost/program_options.hpp>
namespace po = boost::program_options;
void show_help(po::options_description desc)
{
    std::cout << "Hope it helps:" << std::endl;
    std::cout << desc;
}

int main(int argc, const char **argv) {
    // Declare the supported options.
    po::options_description desc("Allowed options");
    desc.add_options()
        ("help", "produce help message")
        ("compression", po::value<int>(), "set compression level")
        ;

    po::variables_map vm;
    po::store(po::parse_command_line(argc, argv, desc), vm);
    po::notify(vm);
    if (vm.count("help")) {
        std::cout << desc << "\n";
        return 1;
    }
    if (vm.count("compression")) {
        std::cout << "Compression level was set to "
        << vm["compression"].as<int>() << "\n";
    } else {
        std::cout << "Compression level was not set.\n";
    }
}
```

Lecture XIII

`gdb`

Debugging Tools

- ▶ GDB - GNU debugger
- ▶ DDD - GUI to GDB

And there are more ...

Preparation:

Compile with debugging support

- ▶ compilation flags for debugging: `-g -ggdb`
- ▶ optimization flags recommended to be removed: `-O0` (But: today, gdb works also with `-O3`, application performance is not affected, but binary size increases and instruction reordering complicates the GDB session).

Basic GDB operations

Debugging of programs written in C, C++, and Modula-2.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- ▶ Start your program, specifying anything that might affect its behavior.
- ▶ Make your program stop on specified conditions.
- ▶ Examine what has happened, when your program has stopped.
- ▶ Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Basic GDB operations

- ▶ running
- ▶ the stack
- ▶ code line
- ▶ print variables
- ▶ print memory
- ▶ breakpoints
- ▶ stepping (c, s, n)
- ▶ attaching to already running program

Sample session - Start program in GDB, set breakpoint

```
richter@bb-richter > gdb twoparticle
GNU gdb (GDB) Fedora 7.5.1-42.fc18
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from twoparticle...done.
(gdb) run 10
Starting program: twoparticle 10
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
event no: 0 999 particle(s)
event no: 1 971 particle(s)
event no: 2 1011 particle(s)
event no: 3 1020 particle(s)
event no: 4 937 particle(s)
^Z
Program received signal SIGTSTP, Stopped (user).
0x0000000000403246 in TParticle::Eta() const ()
Missing separate debuginfos, use: debuginfo-install freetype-2.4.10-5.fc18.x86_64 glibc-2.16-34.fc18.x86_64
(gdb) bt
#0 0x0000000000403246 in TParticle::Eta() const ()
#1 0x0000000000402e4e in Correlate(TParticle, TParticle, TH1*, TH2*) ()
#2 0x00000000004027fa in twoparticle(int) ()
#3 0x0000000000402f65 in main ()
(gdb)
```

Sample session - Examine the program stack

It's a hierarchy of functions.

- ▶ backtrace or bt: print the trace
- ▶ up/down: go one level up or down

```
(gdb) bt
#0  TParticle::Eta (this=0x7fffffffda60) at /home/richter/src/versions_Root/root_v5-33-02b/include/TParticle.
#1  0x000000000402dfd in Correlate (trigger=<incomplete type>, associated=<incomplete type>, hDeltaphiDist=0
#2  0x0000000004027fa in twoparticle (nevent=10) at twoparticle.C:49
#3  0x000000000402f65 in main (argc=2, argv=0x7fffffffdc88) at twoparticle.C:144
(gdb) up
#1  0x000000000402dfd in Correlate (trigger=<incomplete type>, associated=<incomplete type>, hDeltaphiDist=0
117      Float_t eta= trigger.Eta();
(gdb) bt
#0  TParticle::Eta (this=0x7fffffffda60) at /home/richter/src/versions_Root/root_v5-33-02b/include/TParticle.
#1  0x000000000402dfd in Correlate (trigger=<incomplete type>, associated=<incomplete type>, hDeltaphiDist=0
#2  0x0000000004027fa in twoparticle (nevent=10) at twoparticle.C:49
#3  0x000000000402f65 in main (argc=2, argv=0x7fffffffdc88) at twoparticle.C:144
(gdb) down
#0  TParticle::Eta (this=0x7fffffffda60) at /home/richter/src/versions_Root/root_v5-33-02b/include/TParticle.
126      Double_t pmom = P();
```

Hint: that's the way to understand complex programs, set a breakpoint in some module which is called during processing, than step through the trace and you can learn a lot about the program.

Sample session - The source code

- ▶ list: print the code around the current position

```
(gdb) list
121     Double_t      P              ()              const { return TMath::Sqrt(fPx*fPx+fPy*fPy+fPz*fPz);
122     Double_t      Pt              ()              const { return TMath::Sqrt(fPx*fPx+fPy*fPy);
123     Double_t      Energy          ()              const { return fE;
124     Double_t      Eta              ()              const
125     {
126         Double_t pmom = P();
127         if (pmom != TMath::Abs(fPz)) return 0.5*TMath::Log((pmom+fPz)/(pmom-fPz));
128         else return 1.e30;
129     }
130     Double_t      Y              ()              const
(gdb) up
#1 0x000000000402dfd in Correlate (trigger=<incomplete type>, associated=<incomplete type>, hDeltaphiDist=0)
117     Float_t eta= trigger.Eta();
(gdb) list
112     // fill two histograms
113     int Correlate(const TParticle trigger, const TParticle associated, TH1 *hDeltaphiDist, TH2* hDelt
114     {
115         Float_t phi = trigger.Phi();
116         if (phi<0) phi += 2*Pii;
117         Float_t eta= trigger.Eta();
118
119         Float_t associatedPhi = associated.Phi();
120         if (phi<0) associatedPhi += 2*Pii;
121         Float_t associatedEta= associated.Eta();
(gdb)
```

Sample session - Set breakpoint

```
(gdb) break twoparticle.C:115
Breakpoint 1 at 0x402db9: file twoparticle.C, line 115.
```

...and continue: Command: 'continue' or 'c'

```
(gdb) c
Continuing.

Breakpoint 1, Correlate (trigger=<incomplete type>, associated=<incomplete type>, hDeltaphiDist=0x8d4350, hDetaDist=0x8d4350) at twoparticle.C:115
115     Float_t phi = trigger.Phi();
(gdb) list
110
111     // correlate the trigger and associated particles in phi and eta
112     // fill two histograms
113     int Correlate(const TParticle trigger, const TParticle associated, TH1 *hDeltaphiDist, TH2 *hDetaDist)
114     {
115         Float_t phi = trigger.Phi();
116         if (phi<0) phi += 2*Pii;
117         Float_t eta= trigger.Eta();
118
119         Float_t associatedPhi = associated.Phi();
```

Sample session - Walk through the code line by line

- ▶ next or n: execute the next line
- ▶ step or s: execute the next step which might be stepping into a function

```
Breakpoint 2, 0x0000000004027fa in twoparticle (nevent=10) at twoparticle.C:49
49      Correlate(*trigger, *associated, hDeltaphiDist, hDeltaPhiVsDeltaEta);
(gdb) n
45      for (Int_t ca=0; ca<nofParticles; ca++) {
(gdb)
46          if (ct==ca) continue;
(gdb)
47      TParticle* associated=(TParticle*)theparticles->At(ca);
(gdb)
48      if (!associated) continue;
(gdb)
49      Correlate(*trigger, *associated, hDeltaphiDist, hDeltaPhiVsDeltaEta);
(gdb)
45      for (Int_t ca=0; ca<nofParticles; ca++) {
(gdb) s
46          if (ct==ca) continue;
(gdb)
47      TParticle* associated=(TParticle*)theparticles->At(ca);
(gdb)
48      if (!associated) continue;
(gdb)
49      Correlate(*trigger, *associated, hDeltaphiDist, hDeltaPhiVsDeltaEta);
(gdb)
Correlate (trigger=<incomplete type>, associated=<incomplete type>, hDeltaphiDist=0x8d4350, hDeltaPhiVsDelta
115      Float_t phi = trigger.Phi();
```

Sample session - Exiting a function

- ▶ finish: execute the current function to the end and jump one level up

```
Breakpoint 1, Correlate (trigger=<incomplete type>, associated=<incomplete type>, hDeltaphiDist=0x8d4350, hD
115         Float_t phi = trigger.Phi();
(gdb) finish
Run till exit from #0 Correlate (trigger=<incomplete type>, associated=<incomplete type>, hDeltaphiDist=0x8d
0x00000000004027fa in twoparticle (nevent=10) at twoparticle.C:49
49         Correlate(*trigger, *associated, hDeltaphiDist, hDeltaPhiVsDeltaEta);
Value returned is $$1 = 0
```

Sample session - More on breakpoints

- ▶ `info break`: list of breakpoints
- ▶ you can activate, deactivate or delete a breakpoint using its identifier

```
(gdb) info break
```

Num	Type	Disp	Enb	Address	What
2	breakpoint	keep	y	0x00000000004027fa	in twoparticle(int) at twoparticle.C:49
	breakpoint	already hit	3	times	
3	breakpoint	keep	y	0x0000000000402db9	in Correlate(TParticle, TParticle, TH1*, TH2*) at twopart

```
(gdb) disable 2
```

```
(gdb) info break
```

Num	Type	Disp	Enb	Address	What
2	breakpoint	keep	n	0x00000000004027fa	in twoparticle(int) at twoparticle.C:49
	breakpoint	already hit	3	times	
3	breakpoint	keep	y	0x0000000000402db9	in Correlate(TParticle, TParticle, TH1*, TH2*) at twopart

```
(gdb) enable 2
```

```
(gdb) delete 2
```

```
(gdb) info break
```

Num	Type	Disp	Enb	Address	What
3	breakpoint	keep	y	0x0000000000402db9	in Correlate(TParticle, TParticle, TH1*, TH2*) at twopart

Sample session - Examining Variables and Memory

- ▶ print or p: print a variable
- ▶ x: memory dump

```
(gdb) p associatedPhi
```

```
$6 = 0.333897293
```

```
(gdb) p hDeltaphiDist
```

```
$9 = (TH1 *) 0x8d4350
```

```
(gdb) x/32w 0x8d4350
```

0x8d4350:	-162449456	32767	0	50331656
0x8d4360:	-134942896	32767	1698981914	1885434988
0x8d4370:	1766091112	29811	-134942896	32767
0x8d4380:	1698981914	1885434988	1766091112	29811
0x8d4390:	-162447576	32767	66138	1
0x8d43a0:	-162447440	32767	65601536	0
0x8d43b0:	-162447312	32767	65537	1065353216
0x8d43c0:	182	0	-162441680	32767

```
(gdb) x/32b 0x8d4350
```

0x8d4350:	-48	55	81	-10	-1	127	0	0
0x8d4358:	0	0	0	0	8	0	0	3
0x8d4360:	80	-17	-12	-9	-1	127	0	0
0x8d4368:	26	104	68	101	108	116	97	112

```
(gdb) x/32x 0x8d4350
```

0x8d4350:	0xd0	0x37	0x51	0xf6	0xff	0x7f	0x00	0x00
0x8d4358:	0x00	0x00	0x00	0x00	0x08	0x00	0x00	0x03
0x8d4360:	0x50	0xef	0xf4	0xf7	0xff	0x7f	0x00	0x00
0x8d4368:	0x1a	0x68	0x44	0x65	0x6c	0x74	0x61	0x70

GDB command line training

⇒ Try it!

Some other important things you might want to try:

- ▶ Interactive split screen mode with source code display: Ctrl-x a
- ▶ Core Dumps (Set ulimit -c before program crashes), open core dump in gdb

GDB backends - DDD

start DDD:

```
ddd twoparticle
```

Intuitive GUI

Several Tool Windows

- ▶ Source Window
- ▶ GDB console window
- ▶ Data Window
- ▶ Run Control Console

GDB console allows to type commands directly to shell, GUI provides menus and buttons for all important functionality



Tasks

The END ... is just the beginning. So far a few examples.

Now: \Rightarrow Try it!

- ▶ Try different compiler optimization levels and compare
- ▶ Compare CINT interpreter with compiled program
- ▶ Identify possibilities for optimization in the code
- ▶ Do some optimization, check how effective it is in different compiler optimizations
- ▶ Try ddd and debug the program

Lecture XIV

Coding suggestions

Initial remarks

- ▶ Coding is often also a matter of taste, below we try to give some (personal) suggestions what could be considered during coding
- ▶ Some suggestions have a real impact on performance
- ▶ Some suggestions help in error prevention

Includes

- ▶ In header files only use include which **MUST** be there
 - ▶ Unnecessary includes increase the compilation time
- ▶ Most of the time all necessary includes can be moved to the implementation
- ▶ if you have class pointer types use forward declarations

```
class Bar;  
  
class Foo {  
    private:  
        Bar* myVar;  
}
```

Including namespaces, typedef

- ▶ NEVER globally import a namespace in a header

```
using namespace std;
```

- ▶ This means all files including your header will have the std namespace
 - ▶ You could do it in the implementation, not considered nice style
- ▶ Prefer importing only single function, only do it in your implementation

```
using std::cout;
```

- ▶ You could also define an alias, this can even be done locally in a function body

```
using cout = std::cout;
```

- ▶ consider using modern 'typedef'

```
using vecType = std::vector<int>; // modern since C++11  
typedef std::vector<int> vecType; // old way
```


Pass by const reference

- ▶ Complex objects should be passed by reference
- ▶ Do not use it for simple types (int, float, ...), it can be an overhead
- ▶ Prefer over pointer: This makes sure you get a valid object

```
void func(const myClass& c); // good
void func(const myClass* c); // ok, if you know why
void func(myClass c); // not so good, unless you know why
```

- ▶ Similarly, give read access to complex object members by const reference.

```
class Foo {
    myClass bar;
public:
    const myClass& getBar() const; // good
    myClass getBar() const; // not so good; will make a copy
};
```

Pointer

- ▶ Only use pointer if you have a strong reason!
- ▶ If you use objects on the stack you don't have to care about deletion
- ▶ If you need pointers, prefer smart pointers over plain pointers

```
int main() {  
    bool fail = true;  
    auto ptr = new int[9];  
    if (fail) exit(1);  
    delete [] ptr;  
}
```

```
int main() {  
    bool fail = true;  
    auto ptr = make_unique<int[]>(9);  
    if (fail) exit(1);  
}
```

- ▶ 'Convention': plain pointer show non-ownership

Type safety, const, override and final

- ▶ Try programming type safe
 - ▶ Prefer `enum class` over simple `enum`
 - ▶ Use `nullptr` instead of `0`, `0x0`, `NULL`
- ▶ Use `const` where ever possible
 - ▶ For local variables if you know they don't change
 - ▶ Declare functions `const` if they don't change data members

```
int getSomething() { return i; }  
int getSomething() const { return i; }
```

- ▶ use `override` in a function declaration if you intend to override a base class function

Micro optimisation

- ▶ Prefer multiplication over division
- ▶ In case you would need to divide several time by the same number calculate `1/number` once and multiply

```
const float oneOverB = 1./b;  
px *= b;  
py *= b;  
pz *= b;
```

- ▶ Similarly, consider creating class members caching the inverse instead of dividing by class data members.
- ▶ Prefer explicit square, triple, quad over using `pow(x,2)`

```
const float x2 = pow(x, 2); // bad  
const float x2 = x*x;       // good  
const float x3 = x2*x;  
const float x4 = x*x*x*x;    //ok  
const float x4 = x2*x2;      //better
```

- ▶ Prefer using `std::cos`, `std::sqrt` instead of `cos`, `sqrt` as they will dispatch to the optimized version based on the input type.

More considerations

- ▶ Inline short functions (declare them in the header)
- ▶ Prefer single over double precision (float over double)
- ▶ Switch on compiler warnings and fix them all
- ▶ Create complex objects once and re-use them (cache them as data members)
- ▶ Avoid memory allocations inside loops or functions often called.
 - ▶ `std::vector<T> v`; inside loops not good; move it out
- ▶ Define loop variables inside the loop rather than outside

```
int i;  
for (i=0; i<10; ++i) ... // bad  
for (int i=0; i<10; ++i) ... //good
```

- ▶ Prefer `++something` over `something++`

For further readings/videos

- ▶ Official code guidelines of C++
- ▶ Collection of tips for performance
- ▶ C++ Performance and Optimisation - Hubert Matthews
- ▶ Moving Faster: Everyday Efficiency in Modern C++

Try out the performance examples!

`example/performance/`

Lecture XV

Benchmarking and profiling

Intro: Benchmarking and Profiling

- ▶ In order to improve the speed of software, **we need to know about the runtime behaviour of our software!**
- ▶ **Therefore important to regularly**
 - ▶ identify major hotspots of the code consuming most CPU cycles
 - ▶ understand runtime behaviour and resource usage in general
 - ▶ match our expectations against reality
 - ▶ take actions

Rough definition of terms

- ▶ **Profiling** (or tracing) is the process of **inspecting and analysing the runtime behaviour** of software
 - ▶ which function is using most CPU time?
 - ▶ which code paths are taken?
 - ▶ who is actually calling whom?
 - ▶ how much memory is allocated by whom?
 - ▶ what is the instruction mix?
 - ▶ ...
- ▶ **Benchmarking** is the process of taking performance metrics of pieces of code
 - ▶ measure time taken in algorithm foo
 - ▶ measure instruction cache misses in a certain part of code
- ▶ Profiling and benchmarking are connected overlapping concepts

Overview of tools

Benchmarking

- ▶ timers
- ▶ hardware performance counters (perf tool)
- ▶ google benchmark

Basic Profiling

- ▶ perf
- ▶ Intel VTune
- ▶ valgrind/callgrind

Memory checkers

- ▶ valgrind
- ▶ massif

Custom Instrumentation /

Logging Beyond this course. Frameworks that allow you to develop your own inspection tools focusing on a particular questions.

- ▶ Intel PIN
- ▶ Dyninst
- ▶ LD_PRELOAD trick

This is very incomplete. Many specialized tools exist, especially for HPC (parallel architectures). See [here](#) for an overview.

A simple timer

- ▶ Whenever we are interested to get an idea how much time a certain algorithm needs (relative to each other) we can simply use `std::chrono` from C++11
- ▶ One can easily make a class stopwatch and do something like this

```
#include "stopwatch.h"

double kernel(double v);

int main() {
    // use stopwatch like this to measure time taken by some kernel
    {
        // gets initialized and started automatically
        precise_stopwatch watch;

        volatile double x = kernel(1.);

        auto t = watch.elapsed_time<unsigned int, std::chrono::nanoseconds>();
        std::cerr<< "This took" << t << "nano seconds \n";
    }
    return 0;
}
```

Benchmark example

- ▶ There are many benchmark examples in the code
- ▶ Demo with a particular one

`examples/performance`

`examples/performance/10_ROOTvsSTDContainer`

valgrind

- ▶ valgrind is a collection of profiling and debugging tools
 - ▶ memcheck (trace memory access)
 - ▶ callgrind (profiling tool)
 - ▶ massif (heap profiler), helgrind (thread correctness), ...
- ▶ valgrind executes a binary inside a processor emulator
 - ▶ 100x slower than native mode
- ▶ valgrind monitors/sees each single instruction
 - ▶ perfect to understand program flow (reverse engineering)
 - ▶ you see each single object instantiation and function call
 - ▶ easy to detect bugs
 - ▶ track why something is running slowly
 - ▶ can identify some things not possible using statistical tools such as perf/VTune

callgrind

Question I: Where do we “burn” CPU cycles?

Run from the command line:

```
valgrind --tool=callgrind ./twoparticle 10
```

while running: callgrind_control

```
richterm@power1 ~ callgrind_control -b  
PID 1172: ./twoparticle 10  
sending command status internal to pid 1172
```

```
Frame: Backtrace for Thread 1
```

```
[ 0] TMath::Sqrt(double) (4620743 x)  
[ 1] TParticle::P() const (4620743 x)  
[ 2] TParticle::Eta() const (2310372 x)  
[ 3] Correlate(TParticle, TParticle, TH1*, TH2*) (2310372 x)  
[ 4] twoparticle(int) (1 x)  
[ 5] main (1 x)  
[ 6] (below main) (1 x)  
[ 7] 0x0000000000014230 (1 x)
```

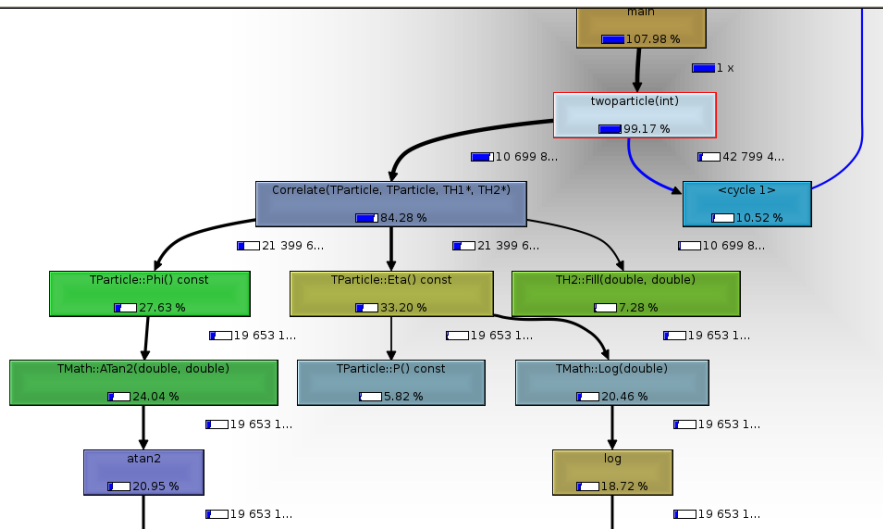
```
kcachegrind callgrind.out.xxxx
```



kcachegrind - Summary

Search:		(No Grouping)	
Incl.	Self	Called	Function
118.34	13.74	39 339 622	<cycle 1>
100.00	0.00	(0)	0x00000000000001590
99.28	0.00	789	0x00000000000014230 <c
99.28	0.00	1	0x0000000000004027d0
99.28	0.00	1	(below main)
99.17	0.00	1	main
99.17	2.72	1	twoparticle(int)
77.40	4.54	9 826 572	Correlate(TParticle, TParti.
33.20	5.07	19 653 144	TParticle::Eta() const
27.63	2.72	19 653 144	TParticle::Phi() const
24.04	3.09	19 653 144	TMath::ATan2(double, dou.
20.95	0.62	19 653 144	atan2
20.46	1.73	19 653 144	TMath::Log(double)
20.33	20.33	19 653 144	0x000000000000bde0
18.72	0.49	19 653 356	log
18.23	18.23	19 653 356	0x00000000000013990
10.52	7.55	19 653 144	TParticle::TParticle(TParti
7.28	4.22	9 826 572	TH2::Fill(double, double)
5.82	3.46	19 653 144	TParticle::P() const
4.76	2.91	9 826 572	TH1::Fill(double)
4.04	4.04	29 479 716	TAxis::FindBin(double)
2.97	2.97	19 653 691	TObject::TObject(TObject c
2.85	2.85	19 681 299	TObject::~~TObject() <cycl
2.85	2.35	19 653 144	TParticle::~~TParticle() <cy
2.35	1.73	19 663 059	TMath::Sqrt(double)
1.85	1.85	19 653 144	TMath::Abs(double)
0.95	0.00	1	TObject::Write(char const*
0.95	0.00	1	TObject::Write(char const*
0.95	0.00	1	TDirectoryFile::WriteTObjec
0.95	0.00	1	TFile::CreateKey(TDirectory
0.87	0.87	19 653 145	TMath::Pi()
0.75	0.75	9 862 372	TObjArray::At(int) const
0.62	0.62	19 663 059	sqrt
0.49	0.49	19 653 152	TAttLine::~~TAttLine()
0.46	0.00	1	0x000000000000e8e0

kcachegrind - the Call Graph



kcachegrind ...

⇒ Try it!

What is your conclusion?

An alternative to callgrind: perf

- ▶ Perf is a profiling tool included in the Linux kernel and quite powerful
- ▶ A tool reading special CPU performance counters (available on hardware)
 - ▶ instruction counter
 - ▶ cache miss counter
 - ▶ branch prediction miss counter
- ▶ A statistical sampling tool, therefore much faster than valgrind for hotspot analysis
- ▶ Very good to get quick hotspot (in time, cache miss, or other) analysis

<https://perf.wiki.kernel.org>

www.brendangregg.com/perf.html

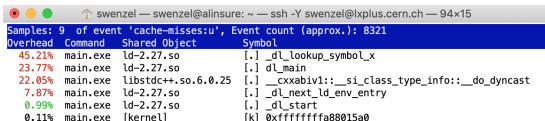
Cycles/time hotspots with Perf

- ▶ Run **perf record -g -e cycles:u [command]** to collect compute cycle statistics similar to callgrind.
- ▶ Run **perf report -g** to visualize them (in the console).
- ▶ Note: The -g defines the ordering (graph or fractal - I prefer graph)

```
swenzel — swenzel@alinsure: ~ — ssh -Y swenzel@lxplus.cern.ch — 77x19
Samples: 47 of event 'cycles:u', Event count (approx.): 29698925
Overhead Command Shared Object Symbol
39.62% main.exe libm-2.27.so [.] __ieee754_pow_sse2
38.20% main.exe libm-2.27.so [.] __exp1
7.03% main.exe libm-2.27.so [.] __pow
5.52% main.exe libc-2.27.so [.] cfree@GLIBC_2.2.5
1.75% main.exe libc-2.27.so [.] handle_intel.constprop.1
1.70% main.exe libstdc++.so.6.0.25 [.] operator new
1.59% main.exe ld-2.27.so [.] _dl_lookup_symbol_x
1.53% main.exe ld-2.27.so [.] _dl_relocate_object
1.46% main.exe ld-2.27.so [.] do_lookup_x
1.32% main.exe ld-2.27.so [.] _dl_next_tls_modid
0.27% main.exe ld-2.27.so [.] _dl_start
0.01% main.exe ld-2.27.so [.] _start
```

Cache-miss Hotspots with Perf

- ▶ Run **perf record -g -e cache-misses:u [command]** to collect cache-miss hotspots
- ▶ Run **perf report -g** to visualize them (in the console).
- ▶ Note: The -g defines the ordering (graph or fractal - I prefer graph)



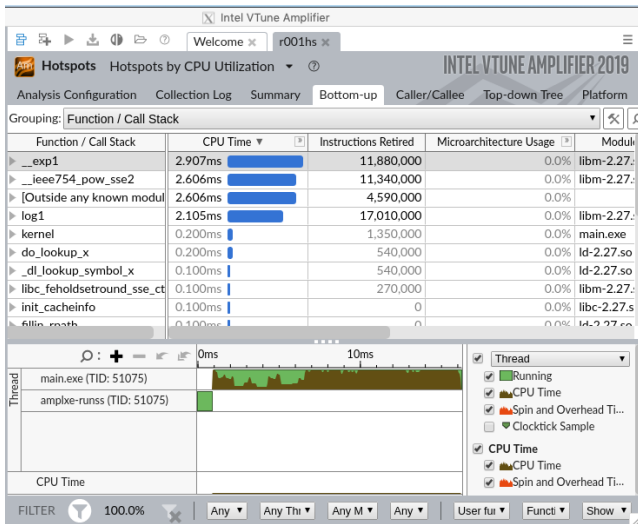
```
swenzel — swenzel@alinsure: ~ — ssh -Y swenzel@lxplus.cern.ch — 94x15
Samples: 9 of event 'cache-misses:u', Event count (approx.): 8321
Overhead Command Shared Object Symbol
45.21% main.exe ld-2.27.so [.] _dl_lookup_symbol_x
23.77% main.exe ld-2.27.so [.] dl_main
22.05% main.exe libstdc++.so.6.0.25 [.] __cxxabi1::__si_class_type_info::__do_dyncast
7.87% main.exe ld-2.27.so [.] _dl_next_ld_env_entry
0.99% main.exe ld-2.27.so [.] _dl_start
0.11% main.exe [kernel] [k] 0xfffffffffa88015a0
```

Bonus: Intel VTune

- ▶ performance tool from Intel with GUI and some useful tools to inspect parallel performance of your program (concurrency)
- ▶ also gives hotspot analysis
- ▶ CERN has a license available under cvmfs
`/cvmfs/projects.cern.ch/intelsw/`; source with

`. /cvmfs/projects.cern.ch/intelsw/psxe/linux/all-setup.sh`
- ▶ make a simple hotspot analysis:
`amplxe-cl -c hotspots ./main.exe`
- ▶ investigate result with `amplxe-gui`

Bonus: Intel VTune GUI



Some general profiling tips

- ▶ Never profile a pure debug build when detecting hotspots (unless you have a specific reason)
 - ▶ actually happens more often than you think
- ▶ Profile using typical data input
- ▶ Usually one profiles on an otherwise idle machine (not relevant for valgrind)
- ▶ When benchmarking pieces of code (microbenchmarks), one should try to
 - ▶ pin processes/threads to specific CPU nodes (to reduce effect of thread migrations)
`taskset -c 1 EXECUTABLE`
 - ▶ repeat measurement many times
 - ▶ (disable CPU turbo boost)

Taking actions from a profile

Some first questions to ask

- ▶ Does the profile make sense?
- ▶ Is function call sequence reasonable?
- ▶ Is the number of object instantiations reasonable?

Attack problems and algorithmic hotspots, some tips

- ▶ look for functions which allocate/deallocate memory (are they areally necessary?)
- ▶ look for functions with large call-count (inlining could reduce cost of calling them)
- ▶ look for prominent occurance of expensive math functions (exp, pow, log, atan2). Are they really necessary or could you use cached values or better algorithms?

Never trust the compiler !! Always verify !!

Instructive callgrind exercise

Can you spot problems with this program in callgrind? Compare the outcome before and after fixing them.

```
// some kernel cubing all elements in a vector
__attribute__((noinline))
double kernel(std::vector<double> const v) {
    double accum = 0.;
    for (auto &e : v) {
        accum += pow(e, 3);
    }
    return accum;
}

int main() {
    std::vector<double> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
    double accum = 0.;
    // loop just to generate some work
    for (size_t i = 0; i < 10000; ++i)
        accum += kernel(v);

    std::cerr << "result " << accum << "\n";
    return 0;
}
```

massif - Heap Profiler

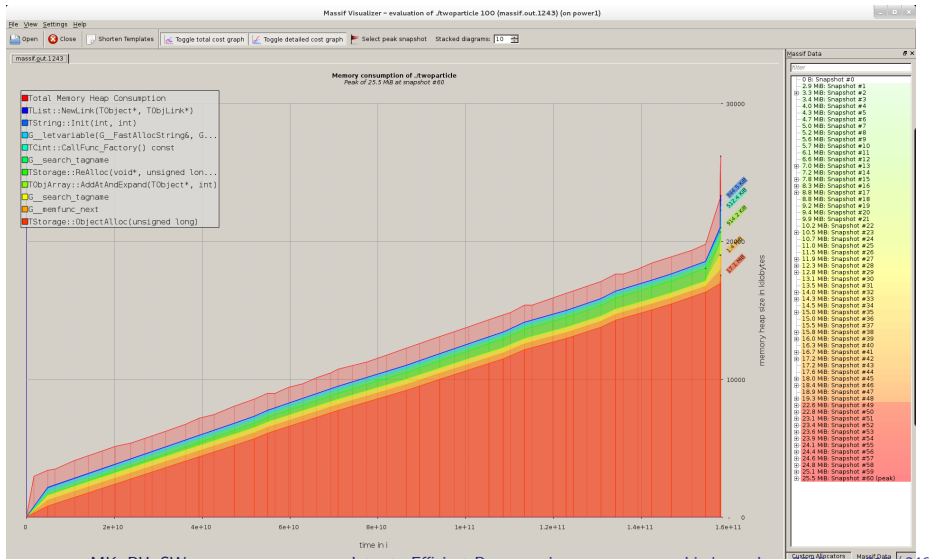
Run from the command line:

```
valgrind --tool=massif ./twoparticle 100
```

Result written to separate output file `massif.out.xxxx`

Examine the massif result - massif-visualizer

massif-visualizer massif.out.xxxx



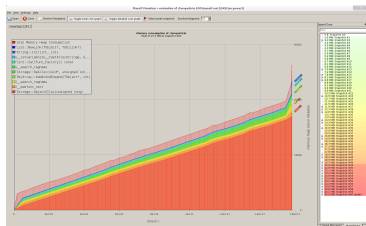
massif and massif-visualizer ...

⇒ Try it!

Anything interesting to observe?

Whoops ... a memory leak

That you never want to observe!



Keep an eye on the memory profile in top

```
top - 02:16:07 up 4 days, 6:37, 1 user, load average: 0.41, 0.20, 0.11
Tasks: 194 total, 2 running, 192 sleeping, 0 stopped, 0 zombie
%Cpu(s): 6.3 us, 0.0 sy, 0.0 ni, 93.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 24688140 total, 3793656 used, 20894484 free, 272976 buffers
KiB Swap: 9974780 total, 0 used, 9974780 free, 2869704 cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1227	richterm	20	0	153680	28468	13500	R	100.0	0.115	0:22.84	twoparticle
1230	richterm	20	0	54028	2788	1908	R	0.333	0.011	0:00.03	top
1	root	20	0	4220	712	612	S	0.000	0.003	0:02.42	init
2	root	20	0	0	0	0	S	0.000	0.000	0:00.00	kthreadd
3	root	20	0	0	0	0	S	0.000	0.000	0:01.73	ksoftirqd/0
5	root	0	-20	0	0	0	S	0.000	0.000	0:00.00	kworker/0:0H

memcheck

- ▶ default 'tool' mode
- ▶ result written to standard output
 - ⇒ pipe it to a file

```
valgrind --show-reachable=yes --leak-check=full \  
./twoparticle 10 2>&1 | tee memcheck.log
```

Output is rather lengthy
→ need strategy to read
the file

Example: less command
less memcheck.log

- ▶ search: /
- ▶ next hit: n
- ▶ prev hit: N

Other options

```
--log-file=xxx.txt  
--suppressions=$ROOTSYS/etc/valgrind-root.supp  
--leak-resolution=high  
--error-limit=no
```

sanitizers

- ▶ valgrind simulates the processor, execution is slow
- ▶ valgrind has only access to information in the binary code
- ▶ the source code might have additional information
- ▶ new compilers can instrument the code to spot problems
- ▶ instrumentation slows down the application, not suited for production, but for debugging runs
- ▶ Two exemplary GCC sanitizer options:
- ▶ Must be passed to compiler and linker (CXXFLAGS and LDFLAGS)
- ▶ Just run the application, the sanitizer prints the problems it finds to console

```
-fsanitize=address  
-fsanitize=undefined
```


Lecture XVI

Design patterns

Design patterns

- ▶ many tasks recur frequently
- ▶ patterns for solutions evolved,
good to know some of them
- ▶ sometimes direct usage,
sometimes adapt the idea to your own need
- ▶ we will discuss some examples,
this is by far non-exhaustive

Singleton

- ▶ consider a manager component which should be accessible from many places in the code
- ▶ you could use a global variable but there is a better solution
- ▶ you want a class for which
 - ▶ exactly one instance is created (when it's needed)
 - ▶ this instance can be accessed from anywhere in the program
 - ▶ nobody can create further instances
- ▶ but think twice whether this is really what you want, e.g. for parameters it might not be

Singleton – code

```
class manager {  
    private:  
        manager();  
};
```

- ▶ a class for which
 - ▶ no instances can be created except from within the class

Singleton – code

```
class manager {  
public:  
    static manager& instance() {  
        static manager man;  
        return man;  
    }  
private:  
    manager() = default;  
    static manager *inst;  
};
```

- ▶ a class for which
 - ▶ no instances can be created except from within the class
 - ▶ entry point which is usable without an existing instance
 - ▶ instance is created when it is asked for

examples/singleton

Factory

- ▶ assume you need an object of a certain base class
- ▶ how to create an instance of the proper derived class without relying on all implementations?
- ▶ you all know and probably have used:
`auto f = TFile::Open(filename);`
why is it better than:
`auto f = new TFile(filename);`
- ▶ what happens if you specify “http://power.week/file” as filename?

Factory (cont'd)

- ▶ the idea is to use a static method of the base class to create the derived class
- ▶ file.h only contains the declaration of this static method, e.g.

```
class file {  
    ...  
    static file* open(const std::string& filename);  
    ...  
};
```

and that's all you have to know when you compile your program, i.e. your program stays independent of the actual implementation

- ▶ the implementation of this method can decide which derived class is created depending on the file name

Lecture XVII

Parallelization and OpenMP

Parallel computation

- ▶ There are three ways a computer can perform multiple calculations in parallel.
- ▶ Vectorization: Also called Single-Instruction-Single-Data (SIMD).
- ▶ **Parallelization**: MIMD: Multiple Instruction Multiple Data.
- ▶ Instruction level parallelism

We will focus on parallelization.

Parallel computation

▶ Vectorization

- ▶ Performs the same calculation on multiple data values (usually components of a vector).
- ▶ Cheap to implement in the hardware, no additional control flow needed.
- ▶ Some limitations on the applicability, suited code and in particular suited data structures needed.

▶ Parallelization

- ▶ Totally independent instructions streams operate on different data sets.
- ▶ principle two independent processors run in the computer, having access to the same address space.
- ▶ Much higher flexibility, but more complicated in hardware: The processor has two full cores.
- ▶ Simpler to write parallel code.

▶ Instruction level parallelism

- ▶ Lowest level of parallelization: The processor executes multiple instructions of the same instruction stream that do not depend directly on each other in parallel:

```
int a, b, c, d; a = 2 * b; c = 2 * d;
```
- ▶ The compiler can optimize for this.

Parallel computation: std::thread

- ▶ Since c++11: thread support in std.

```
void thread_main() {  
    std::cout << "Hello, World (thread)" << std::endl;  
}  
int main() {  
    std::thread t1(thread_main);  
    std::cout << "Hello, World (main)" << std::endl;  
    t1.join();  
    return 0;  
}
```

Parallel computation: OpenMP

- ▶ A more "accessible" approach is OpenMP basing on compiler pragmas (needs `-fopenmp` compiler flag!).
- ▶ Pragmas are "optional" hints for the compiler, to pass additional information or flags outside the C++ standard.

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv)
{
    printf("Start\n");
    #pragma omp parallel num_threads(2)
    {
        printf("Thread %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
    }
    printf("End\n");
}
```

```
Start
Thread 1 / 2
Thread 0 / 2
End
```

Parallel computation: Critical sections

- ▶ One problem: shared global variables.

```
class ParticleGenerator gen;  
std::vector<Particle> particles;  
int i;  
#pragma omp parallel num_threads(2)  
{  
    for (i = 0; i < 10; i++)  
        particles.push_back(gen.Generate());  
}
```

- ▶ The two threads collide accessing the variable `i`, the generator, and the `std::vector`.

Parallel computation: Critical sections

- ▶ Three solutions:
- ▶ Private variables: do only use local variables, or mark them as `omp private`.
- ▶ Make functions **thread-safe**, i.e. they can run multiple times concurrently. They must not use static variables or write to global / member variables.
- ▶ Use critical sections: OpenMP blocks the threads such that only one thread runs through the section in parallel.

```
class ParticleGenerator gen;
std::vector<Particle> particles;
int i;
#pragma omp parallel num_threads(2) private(i)
{
    for (i = 0; i < 10; i++) {
        Particle tmp = gen.Generate(); //Must be thread safe!
        #pragma omp critical
        {
            particles.push_back(tmp);
        }
    }
}
```

Parallel computation: Mutexes, Semaphores, Spinlocks, Atomics

There are some more advanced concepts for critical sections.

- ▶ Mutexes - An object that can be locked by only one thread at a time. If two threads try to lock the object, the operating system makes the second one wait until the first one unlocks - available as e.g. `pthread_mutex_t`.
- ▶ Spinlocks - Similar to a mutex, but the thread performs active wait polling the variable. Good for very short term locks (nanoseconds) - available as e.g. `pthread_spinlock_t`.
- ▶ Semaphores - Like a mutex, but not binary. A resource that can be decremented and incremented. When a thread decrements it below zero, it automatically waits for another thread to increment it again - available e.g. via POSIX `sem_t`;
- ▶ Atomic instructions / variables - Operations are atomic and cannot be interrupted by another thread. A counter as `"int i; i += 1;"` is not thread-safe, but e.g. C++11 atomic integers are.

Parallel computation: More OpenMP constructs

- ▶ OpenMP has plenty of predefined constructs for many cases.
- ▶ The most prominent one is the **parallel for** for loops.
- ▶ You can also look into: **#pragma omp simd**, **#pragma omp parallel sections**, **#pragma omp task**, the **reduction** option, or even **#pragma omp target** for offloading e.g. to GPUs.

```
#pragma omp parallel for
for (int i = 0; i < n; i++)
{
    data[i] = complicated_function(i);
}
```

- ▶ Automatically splits the iterations of the for-loop among multiple threads.
- ▶ If no "num_threads" option is given, the number of threads is determined automatically (or via the `omp_set_num_threads()` library function).

Mutex example instead of critical section

- Note: A critical section is basically an automatic mutex.

```
class ParticleGenerator gen;
std::vector<Particle> particles;
pthread_mutex_t lock;
if (pthread_mutex_init(&lock, NULL) != 0) {...}
#pragma omp parallel for
for (int i = 0; i < 10; i++) {
    Particle tmp = gen.Generate();
    pthread_mutex_lock(&lock);
    particles.push_back(tmp); //critical section
    pthread_mutex_unlock(&lock);
}
pthread_mutex_destroy(&lock);
```

- Mutex ensures that only 1 thread enters the critical section.

Mutex example instead of critical section

- ▶ Today, you can use `std::mutex`, without calling `create` / `destroy`.

```
std::mutex lock;
#pragma omp parallel for
for (int i = 0; i < 10; i++) {
    Particle tmp = gen.Generate();
    std::lock_guard<std::mutex> guard(lock); //automatically unlocks when it goes
    particles.push_back(tmp);
}
```

Parallel computation: Reduction

- ▶ Sometimes you might want to compute a large sum (or similar) distributedly.
- ▶ Each thread can execute a part of the loop..
- ▶ One needs to aggregate the results at the end.

```
std::vector<int> results(nThreads);  
for (int i = 0; i < nThreads; i++) results[i] = 0;  
#pragma omp parallel for num_threads(nThreads)  
for (int i = 0; i < n; i++)  
{  
    results[omp_get_thread_num()] += complicated_function(i);  
}  
for (int i = 1; i < nThreads; i++) results[0] += results[i];
```

- ▶ Code does not look too nice.

Parallel computation: More OpenMP constructs

- ▶ OpenMP has a reduction construct for this.

```
int result = 0;
#pragma omp parallel for reduction( + : result)
for (int i = 0; i < n; i++)
{
    result += complicated_function(i);
}
```

Parallel computation: `std::async`

- ▶ Since C++11: `std::async`.
- ▶ Launches a function (e.g. a lambda) in the background.
- ▶ immediately returns a `std::future` - value can be queried at a later time; e.g. `future::get()` will block until result available.

```
#include <future>
int main() {
    using namespace std;
    auto f = async([]() -> int { cout << "first\n"; return 1; });
    auto g = async([]() -> int { cout << "second\n"; return 2; });
    int i = f.get();
    int j = g.get();
    return 0;
}
```

Parallel computation: coming to your compiler soon: parallel stl

- ▶ Since c++17: parallel stl algorithms
- ▶ execution policy can be set in the arg list to e.g. sequenced or parallel
- ▶ Available in gcc9.1, but library support experimental (disabled by default as of summer 2019)

e.g.:

```
vector<float> data(state.range(0.));  
iota(data.begin(), data.end(), decltype(data)::value_type(0.));  
  
for_each(execution::par, data.begin(), data.end(), [](auto& element) {  
    element = complexFunction(element);  
});
```

Parallel computation: Summary

- ▶ There are low-level approaches like threads (Windows / Linux API, pthreads, std::thread).
- ▶ Some high level approaches are easier to use.
- ▶ In particular **openmp parallel for** is often an easy to use black box (if the code is thread safe).
- ▶ For the project, I would suggest not to spend too much time with pthreads, etc., but simply attempt to use OpenMP.
- ▶ Can you find a place in you code, where you can easily plug it in?
- ▶ Be aware: many root functions are not thread safe!
- ▶ check out the examples in `code/examples/openmp` and some benchmarked (google benchmark) examples in `code/examples/benchmark`