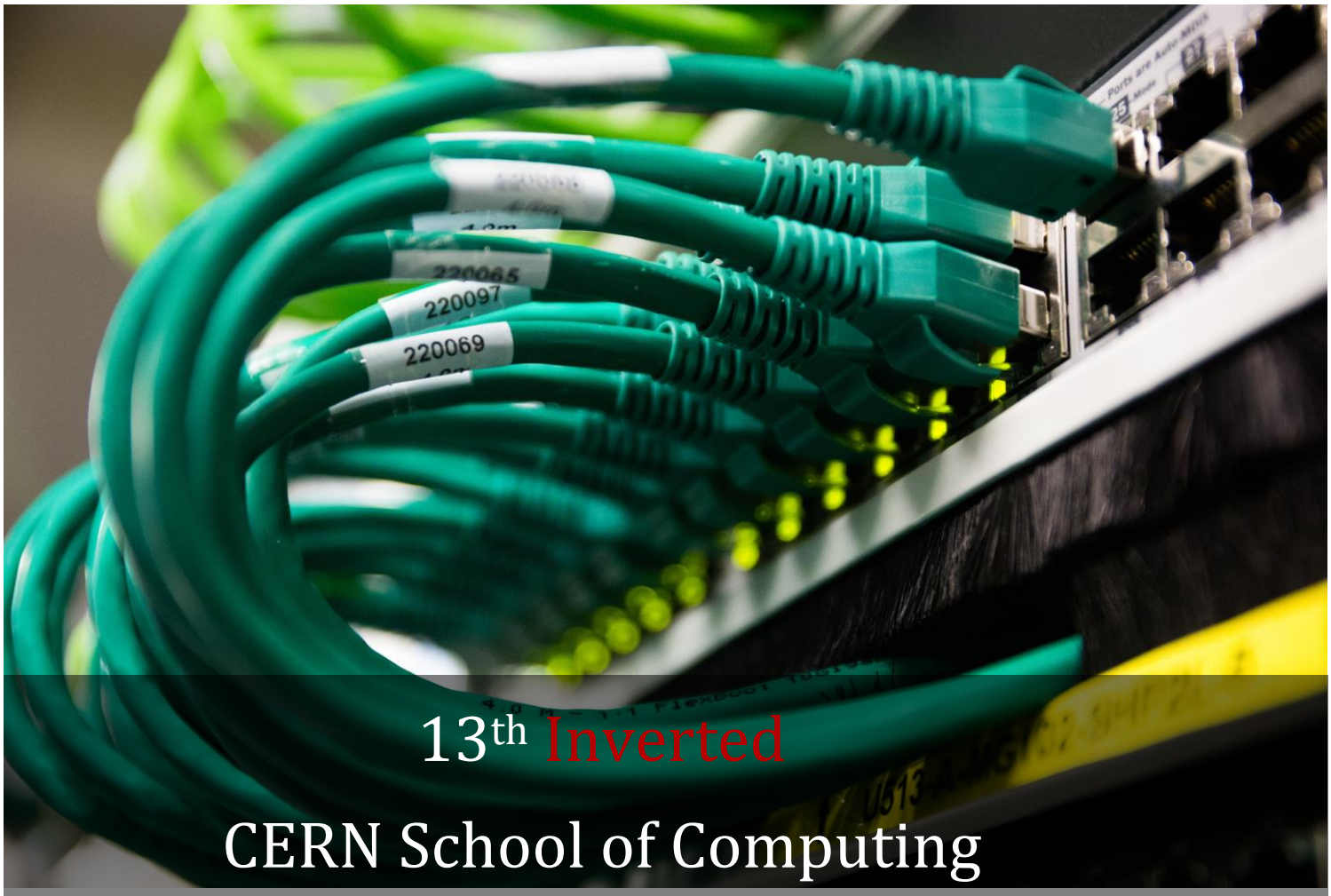


13th Inverted CERN School of Computing

28 September to 2 October 2020 – online school

Live webcast, slides & recording at **https://indico.cern.ch/e/iCSC-2020**

Lecturers

Miguel ASTRAIN ETXEZARRETA	*Universidad Politécnica de Madrid, Spain*
Emil KLESZCZ	*CERN*
Kilian LIERET	*Ludwig Maximilian Univers. of Munich, Germany*
Nis MEINERT	*German Aerospace Center (DLR), Germany*
Ruchi MISHRA	*Nicolaus Copernicus Astronomical Center, Poland*
Rita ROQUE	*LIBPhys, University of Coimbra, Portugal*

CERN

iCSC CERN School of Computing

iCSC 2020 booklet compiled and edited by Joelma Tolomeo.

Front page image: CERN computing facilities views from 2018 (IT-PHO-CCC-2018-001-25)

It is with great pleasure that I warmly welcome you all, lecturers and attendees alike, to this Inverted CERN School of Computing (iCSC) 2020.

As many of you know, this school was initially scheduled to take place in March 2020 at CERN. For the obvious reasons, we had to postpone it – hoping that we will be able to have a normal school with physical attendance. However, the ongoing COVID-19 pandemic forced us to switch to the online format. While we would all prefer to attend this school in person, I hope that this online school will live up to your expectations.

You will certainly appreciate the very rich program of the school, based on excellent proposals received from students of the three past schools: CSC 2019 (Cluj-Napoca, Romania), Thematic CSC 2019 (Split, Croatia) and CSC 2018 (Tel-Aviv, Israel). This school consists of 10 hours of lectures, as well as 4 hours of instructions and demo sessions for the hands-on exercises, covering a big range of topics, including Programming Paradigms and Design Patterns, Heterogeneous Programming with OpenCL, Computational Fluid Dynamics, Reconstruction and Imaging, Modern C++ features, Big Data processing with SQL, and more. I'm sure you will find the classes both relevant and very interesting!

This is already the 13th edition of the Inverted CSC. The initial idea was to give the floor to former CSC students, so that they could share their knowledge and expertise with their colleagues. The fact that we've reached the 13th edition this year, and that we actually received more proposals for classes than we were able to accommodate, proves that this idea is still valid. On a more personal note, the Inverted CSC remains close to my heart. I was one of the lecturers of its first edition back in 2005, giving a lecture about software security - even though I worked on something else at that time. I still remember how this experience had pushed me to develop further my knowledge and passion in security, and - consequently - heavily influenced my professional life. I hope this year's iCSC will have a similar positive impact on the careers of the lecturers!

I wish to thank and congratulate the lecturers: Miguel Astrain, Emil Kleszcz, Kilian Lieret, Nis Meinert, Ruchi Mishra and Rita Roque, for their significant work put in preparing the lectures; the mentors for their invaluable input and feedback; Joelma Tolomeo (the School's Administrative Manager) and Jarek Polok (the Technical Manager) for the work behind the scenes; and finally, you - the attendees - for your interest and presence.

Please join me in enjoying this great learning and knowledge-sharing experience!

Sebastian Łopieński
Director
CERN School of Computing

CONTENTS

ORGANISERS AND MENTORS

Director	**Sebastian ŁOPIEŃSKI**
Administrative Manager	**Joelma TOLOMEO**
Technical Manager	**Jarek POLOK**
Mentors	**Nikos KASIOUMIS**
	Sebastian ŁOPIEŃSKI
	Alberto PACE
	Danilo PIPARO
	Sebastien PONCE
	Ivica PULJAK
	Arnulf QUADT
	Are STRANDLIE
	Enric TEJEDOR

CERN School of Computing

The inverted School of Computing (iCSC) is part of the annual series of schools organized by the CERN School of Computing **http://cern.ch/csc**

Miguel Astrain Etxezarreta

Universidad Politécnica de Madrid, Spain

I studied Physics and Electronics Engineering at Universidad del País Vasco (UPV), where I am from. I moved to Madrid where I completed my Msc in Nuclear Science and Technology and I am currently a PhD student at Universidad Politécnica de Madrid (UPM).

I have been working at the I2A2 Instrumentation laboratory at UPM in different positions since 2014. Among others, I develop solutions based on FPGAs for data acquisition and control systems. My thesis is focused on the development of such systems using OpenCL.

Lecture(s)

Heterogeneous computing: Introduction to OpenCL for FPGAs

This seminar introduces OpenCL as a heterogeneous programming language. We will analyze the structure of an OpenCL program and how to handle the different elements of OpenCL. Examples of parallel computing are presented to illustrate how to write computing programs in OpenCL. Finally, we discuss how these concepts have to be translated into the FPGA context to achieve high performance.

The exercises expand the concepts with example programs. The examples help to understand the role of the host program to allocate memory, schedule tasks, and execute kernels in the OpenCL device. More advanced examples explain optimization decisions made due to hardware particularities.

Mentor(s): Danilo Piparo

Emil Kleszcz

CERN

I am a software engineer currently developing a new generation Hadoop and Spark service at CERN.

I hold an MSc diploma in Computer Science from Warsaw University of Technology. During the studies, I completed two student exchanges, one at the Technical University of Delft and the other at the Nanyang Technological University in Singapore. I was working for almost three years as a Java Software Developer at CERN, and at two other IT companies.

My main areas of professional interest are web development, distributed computing, and big data. In my free time I enjoy reading, skiing, running and traveling.

Lecture(s)

Big Data technologies and distributed data processing with SQL

The interest of many users communities in solutions based on the Big Data ecosystem such as Hadoop is constantly increasing at CERN, including physics experiments, monitoring, and accelerator controls.

This lecture will introduce the participant to some of the Big Data technologies that CERN offers for distributed processing. It will cover some history, architecture, and specifics of selected technologies such as the Hadoop, Spark, and Presto for SQL-like data processing.

Moreover, this talk will cover the example of using the Big Data tools to speedup some computations on hundreds of terabytes of events coming from the Atlas experiment at CERN.

Mentor(s): Sebastian Łopieński, Alberto Pace

Kilian Lieret

Ludwig Maximilian University of Munich, Germany

I'm a PhD student for the Belle II experiment and part of its software group. Starting this year I am also one of the convenors of the software training WG of the HEP software foundation.

After Bachelor's degrees in mathematics and physics I did a Master's in theoretical and mathematical physics, while gaining software experience on various work and spare time projects. I found that reading relevant computer science literature as well as thinking and talking about design decisions more consciously and formally greatly improved my understanding of software and coding.

With my course I would like to share some of these insights and provide a platform for further discussion.

Lecture(s)

Programming Paradigms and Software Design Patterns

Ever been to the point where even little improvements require large refactoring efforts or dirty hacks? If only one had made the right choices on the way!

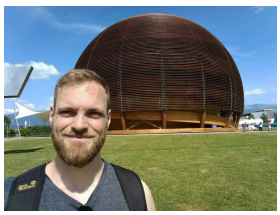
This course discusses programming paradigms (functional vs object oriented, imperative vs declarative programming, ...) and introduces common design patterns (reusable solutions to common problems). While theory alone is unlikely to make you a better developer, having an overview of common principles paired with the vocabulary to describe them will make it easier to come up with the right solutions.

In the exercises, we refactor various code snippets using the ideas of the lecture. Different approaches and their applicability to HEP problems are being discussed. Basic familiarity with python or C++ is advised for the exercises.

This series is aimed at scientists with little formal training in software engineering, but veterans are always welcome to join for additional input in the discussions!

Mentor(s): Enric Tejedor, Sebastien Ponce

Nis Meinert



German Aerospace Center (DLR), Germany

I have been studying physics at Rostock University and recently handed in my doctoral thesis. In the large data sample of the LHCb experiment at CERN I searched for rare b to open-charm two-body decays using physical intuition and statistical / ML methods. The results of this study are the first discovery of a decay by observing it with a statistical significance above five standard deviations, and estimations of upper limits of another, back then not yet observed, decay.

During my PhD time I had the fantastic opportunity to visit and participate at various international HEP, IT and ML orientated schools and conferences. These thrilling meetings boosted my passion for high performance computing and AI research and granted insights into leading edge developments.

Since August I am working at the German Aerospace Center (DLR) where I search for anomalies in safety-critical maritime traffic movements using non-linear Kalman Filters and AI/ML techniques.

I am a Linux fanboy, a C++ enthusiast, founder of a Python User Group, coffee lover and proud father of a lovely son.

Lecture(s)

Modern C++

1) Modern C++ vs. its legacy: when stability is more important than performance

“Leaving no room for a lower-level language”, this has always been the declared mantra of the C++ language development. At the same time C++ is an old language and its legacy affects possible performance improvements in the STL due to ABI stability concerns. ABI stands for “Application Binary Interface” and its stability is often put over performance on the one hand while on the other hand, “ABI” or even its stability is not mentioned a single time in the entire C++ standard.

In this talk I will explain what ABI stability means, why it is not mentioned in the standard but is still lethal to many proposals, and discuss some recent implications. I will not be able to present solutions to the dilemma; the hope is to encourage attendees to make their mind and to attend ongoing discussions about this vital topic.

2) Demystifying Value Categories in C++ (L1, L2)

Value categories are omnipresent in today’s C++ code bases. Since the advent of move semantics the field continuously becomes wider and knowing terms such as lvalue and rvalue only from compiler error messages is not enough anymore.

I will present an overview about C++ value categories and decay rules, the subtle difference between pointers and references, explain why neither std::move moves, nor std::forward forward values and talk about implication on related topics such as RVO.

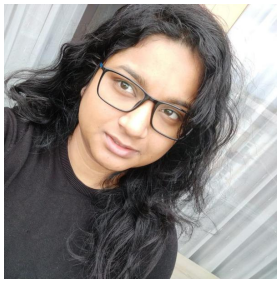
3) Everything you (n)ever wanted to know about C++'s Lambdas

From a syntactical point of view, the Lambda expression of C++ is nothing but syntactic sugar of a struct with an appropriate call operator overload. On the other hand, this simple syntax is shockingly flexible and allows powerful abstractions in a functional way, while providing elegant and easy to read code in a language that is notoriously famous for being unnecessary clunky and verbose.

I will give an overview about the basic syntax and best practices. I will then talk about stateful Lambdas, Lambda inheritance and their real-world applications.

Mentor(s): Sebastien Ponce, Enric Tejedor

Ruchi Mishra



Nicolaus Copernicus Astronomical Center of Polish Academy of Sciences, Warsaw, Poland

Originally from India where I received my masters degree in Physics, I moved to Warsaw to follow my passion.

I am currently in my 2nd year of doctoral studies in Astrophysics . My PhD work focuses on computational simulation of environment around Neutron star and black hole. I use codes like PLUTO and KORAL for my work.

I like to explore new places, their culture and food habits. I also like to paint in my free time.

Lecture(s)

Computational fluid dynamics for physicists and engineers

Computational fluid dynamics(CFD) is an interdisciplinary approach involving Physics and Computer Science which deals with solving the governing equation of fluid flow. With the increasing advancement of technology, CFD has gained a lot of popularity in wide ranging topics of Physics and Engineering.

In this lecture we will look into the basic idea behind CFD, the conservation system of equations governing fluid dynamics, physical domain and equation discretization, use of numerical methods such as Finite difference and Finite volume methods, different boundary conditions and Reimann solvers.

Mentor(s): Ivica Puljak, Arnulf Quadt, Sebastian Łopieński

Rita Roque

LIBPhys, University of Coimbra, Portugal

I was born in 1995 in Coimbra, Portugal, where I have lived all my life.

I am a second year PhD student in Physics Engineering at University of Coimbra, where I am currently working on a novel optical imaging gaseous detector. In 2016 (my last bachelor year) I joined my current research group and have been working on X-ray imaging gaseous detectors ever since. At the end of the day, I always try to find an hour or two to work on my short stories and novels.

Lecture(s)

From the electric pulse to image quality — the analysis chain of imaging detectors

Imaging gaseous detectors are the base-technology for many applications, such like Medical Imaging and Airport Security, as they are an upgrade to our biologic eyes.

The objective of this lecture is to join physics and computing concepts to fully understand the analysis chain of an imaging gaseous detector. We will discuss the physics inside the detector, the engineering that processes the electric pulses, the computing skills behind image reconstruction and come back to the physical concepts by extracting meaningful parameters from the final image.

In the exercise session, you will follow these steps yourself by analyzing electric pulses from an oscilloscope, reconstructing an image from a data file and quantifying some important parameters like position resolution and noise.

Mentor(s): Are Strandlie, Ivica Puljak

Weekly view

https://indico.cern.ch/event/853710/timetable/

SLIDES

TOWARDS “WRITE ONCE, RUN ANYWHERE”

Miguel Astrain Etxezarreta
Universidad Politécnica de Madrid (UPM)

Acknowledgements:
Mariano Ruiz, Antonio Carpeño and Sergio Esquembri of the Universidad Politécnica de Madrid (UPM)

INSTRUMENTATION & APPLIED ACOUSTICS RESEARCH GROUP
miguel.astrain@i2a2.upm.es

Agenda

Introduction (1h)
- Context
- The OpenCL Models
- OpenCL Host Programming
- OpenCL Kernel Programming
- FPGA Oriented Kernel Design

Advanced (1h)
- More on OpenCL host programming
- Example: Matrix Multiplication
- More on FPGA Oriented kernel design
- Memory Hierarchy
- Synchronization in OpenCL
- The OpenCL Event Model
- Building OpenCL

Additional materials

In addition to these slides, and the set of examples, it is useful to have:
- **OpenCL in Action. Matthew Scarpino. Manning 2011. (Very recomended)**
- Heterogeneous Computing with OpenCL. Gaster, Howes, et al Morgan Kaufmann 2011.
- OpenCL Parallel Programming Development CookBook. Raymond Tay. PACKT 2013.

Vendor specific documentation for each platform:
- Xilinx VITIS Softwate platform
  - https://www.xilinx.com/html_docs/xilinx2019_2/vitis_doc/index.html
- Intel FPGA SDK for OpenCL
  - https://www.intel.com/content/www/us/en/programmable/products/design-software/embedded-software-developers/opencl/support.html

Before we start…

Three main profiles that will benefit from this course:
1. **Scientific background**: Scientific experiments, Data Acquisition, some hardware…
2. **Programmer background:** Programming frameworks, some parallelization, GPUs….
3. **Hardware background:** HDL, FPGAs, DSP, some programming….

No knowledge is required from the other fields.

The goal… understanding the heterogeneous programming model.

1. Lower learning curve to write high-performance algorithms for hardware.
2. Apply programming concepts to hardware. Integrate many hardware in a unified framework.
3. Optimizations thanks to hardware knowledge, provide tools for the above.

The very basics

Very recommended to review the CSC2019 lectures on these topics:

1. Software Design in the Many-Cores era; A. Gheata, E. Tejedor.
2. Base Concepts of Parallel Programming: A Pragmatic Approach; A. Gheata, E. Tejedor.

Over-summarized, the frameworks for high-performance computing help on:

1. Dividing tasks into smaller problems.
2. Managing the tasks (and memory!) to execute efficiently.

But OpenCL offers one more perk, managing heterogeneous hardware:

1. Additional to parallelism I can select optimal hardware! → More performance!
2. A single computing language can manage many devices! → Ease of maintenance, portability!
3. An open standard, which is manufacturer agnostic →Write once, run anywhere!

It's a Heterogeneous world

- OpenCL lets Programmers write a single portable program that uses ALL resources in the heterogeneous platform

A modern computing platform includes:

- One or more CPUs
- One or more FPGAs
- One of more GPUs
- DSP processors
- FPGA with AI Cores + DSP + PCIe +DDR
- ADCs?! (Xilinx RFSOC) …

Industry Standards for Programming Heterogeneous Platforms

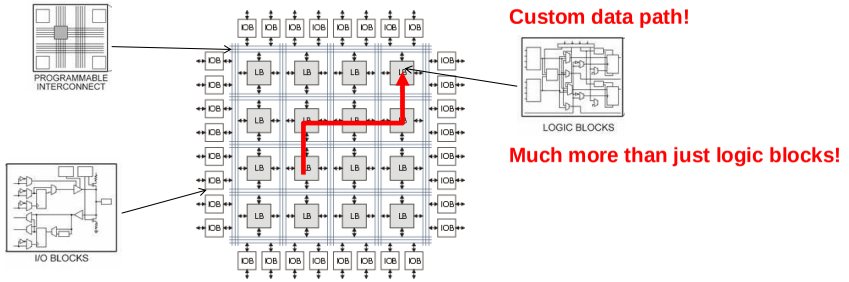
OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, FPGAs, DSPs,…

OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group.

FPGAs?

- Very interesting hardware, with real-time capabilities.
- While often losing to GPUs in raw computation muscle, FPGAs are more energy efficient!
- The learning curve for HDLs is steep, what do I need to know about FPGAs to use OpenCL?
  - Actually, very little! Knowledge helps, of course, but the tools provided with OpenCL SDKs help us!



THE OPENCL MODELS

There are quite a few.

OpenCL Models

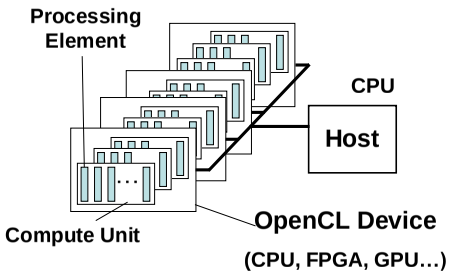
OpenCL defines a set of models to organize the core ideas:

- Platform Model
- Execution Model
- Memory Model
- Programming Model

from: Khronos registry, OpenCL specification 2.2
276 pages of condensed information!

OpenCL Platform Model

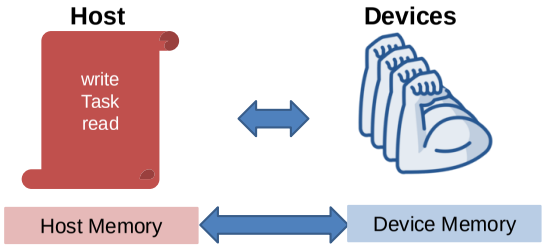
- One *Host* and one or more OpenCL *Devices.*
  - Each OpenCL Device is composed of one or more *Compute Units.*
    - Each Compute Unit is divided into one or more *Processing Elements.*



OpenCL keywords are high-lightened in RED hereinafter.

Host and Device

- The host has the recipe on how to perform the computation
  - It uses commands to the device to do so.
- The device has the power to perform the computation.
  - Can only understand kernel code.
- Memory divided into *Host Memory* and *Device Memory.*



OpenCL Execution Model

- Define a problem domain and execute an instance of a kernel for each point in the domain.
  - The smallest unit is called a work-item
- If the problem needs synchronization or has dependencies, manage them into work-groups

C/C++ code:

```
void times_two( float* input,
float* output)
{
   int i =
some_identifier_routine();
   output[i] = 2.0f * input[i];
}
```

OpenCL code:

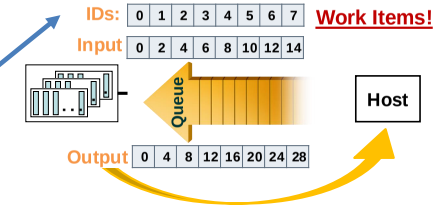
```
__kernel void times_two(
    __global float* input,
    __global float*
output)
{
   int i =
get_global_id(0);
   output[i] = 2.0f *
input[i];
}
```

OpenCL

OpenCL qualifiers

OpenCL Execution Model

```
__kernel void times_two(
    __global float* input,
    __global float*
output)
{
   int i = get_global_id(0);
   output[i] = 2.0f *
input[i];
}
```



- Note there is no for loop.
- This is very GPU like programming.
- No imposed order of execution

The idea behind OpenCL

- Computation divided into simpler functions (a kernel) executing at each point in a problem domain.
- Most of the computing muscle is usually needed in a few lines of code.
- Typical example: 1024x1024 image with one kernel invocation per pixel or 1024x1024 = 1,048,576 kernel executions

Traditional loops

```
void multiply(const int n, const
float *a,
        const float *b, float
*c)
{
  int i;
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
}
```

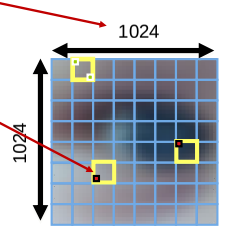
Data Parallel OpenCL

```
__kernel void multiply (__global
const float *a,
                __global const float
*b,
                        __global
float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
// many instances of the kernel,
// called work-items, execute
```

in parallel, but how much parallelism? ...

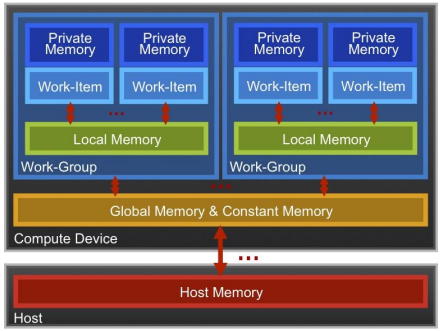
Example: N-dimensional domain

- Typical example, an image:
- Global Dimensions:
  - 1024x1024 (whole problem space)
- Local Dimensions:
  - 128x128 (**work-group**, executes together)
- The work-item is the pixel.
- You can tune the dimensions that are "best" for your hardware.
- Remember Platform Model "CU" and "PE".



OpenCL Memory model

- *Private Memory*
  - Per work-item
- *Local Memory*
  - Shared within a work-group
- *Global Memory*
  - Visible to all work-groups
- *Host Memory / Constant*
  - On the CPU

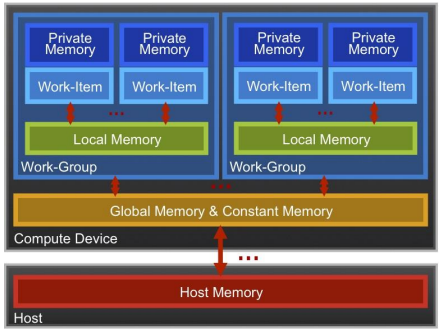


https://www.khronos.org/

OpenCL Memory model

Memory management is **explicit!**.

More on this and SYCL later…



https://www.khronos.org/

OpenCL Programming Model

Definitions, lots of definitions… What does it all mean?

- They help keep ideas clear.
- Divide and manage the problem the OpenCL way!
- The programming model is … the rest …
- Your problem, your algorithm, your hardware

OpenCL Programming Model

- Data Parallelism
  - kernels and indexes
    - Work-items
- Task Parallelism
  - kernels and queues
    - Work-Group
- Single Instruction Multiple Data
  - OpenCL C
    - Vector instructions.
- Single Program Multiple Data
  - Platforms and devices
    - Deploy to multiple devices

OPENCL HOST PROGRAMMING

What is your favorite language?

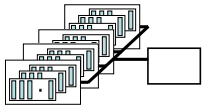
Host Programming

- There is now a full specification in C++.
- Host bindings are available for C, C++, Java, Python.
- Kernels are written in **OpenCL C subset of C99** with specific extensions and restrictions.

- **Recommend using C/C++. Most examples are written in C.**
- **Lots of development effort in C++, SYCL,…**

Host Programming

- The host program is the code that runs on the host to:
  - Setup the environment for the OpenCL program
  - Create and manage kernels



- 5 simple steps in a basic host program:
  1. Define the ***platform*** … platform = devices + context + queues
  2. Create the ***program*** (dynamic library for kernels)
  3. Setup ***memory*** objects
  4. Define the ***kernel*** (attach arguments to kernel functions)
  5. Submit ***commands*** … transfer memory objects and execute kernels

Host Program

```
// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(0,
                     CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetConte                                          &cb);

cl_device_id[] devices = malloc(cb);
clGetContextInfo(context,CL_CONTEXT_DEVICES,cb,devices,NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context,devices[0],0,NULL);
```

Define platform and queues

```
// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
       CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA, NULL);
memobjs[1]                                          cb, NULL);

memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                            sizeof(cl_float)*n, NULL, NULL);
```

Define memory objects

```
// create the program
program =                                           NULL, NULL);
```

Create the program

```
// build the progra
err = clBuildProgra                                 L,NULL);
```

Build the program

```
// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                      sizeof(cl_mem));
err |= clSetKe                                      1],
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2],
                      sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;
```

Create and setup kernel

```
// execute kernel
err = clEnqueueNDR                                  1, NULL,
                    global_work_size, NULL,0,NULL,NULL);
```

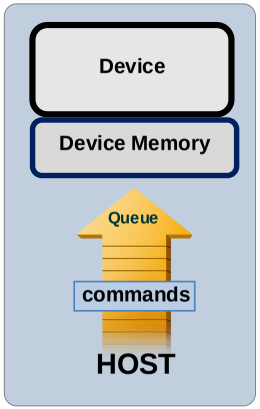
Execute the kernel

```
// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
                                                    0, NULL, NULL);
```

Read results on the host

Command-Queues

- The computation “recipe” is scheduled through the command-queue.
- ***Commands*** for a device include kernel execution, synchronization, and memory transfer operations.



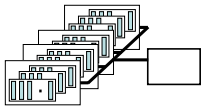
Example: step by step

Lets analyse a host program:

1. **Platform:**

Ex:
1.
2.
3.
4.
5.

Device:
CPU, Intel x86, OpenCL C



Host:
CPU, Linux, C language

****The code ahead might be simplified or wrong to keep it shorter and readable.***

Example: step by step

Ex:
1.
2.
3.
4.
5.

```
cl_uint num_of_platforms = 0;
clGetPlatformIDs(0, 0, &num_of_platforms);
cl_platform_id* platforms = new cl_platform_id[num_of_platforms];
clGetPlatformIDs(num_of_platforms, platforms, 0);
---------------------------------------------------------------------------------------------
clGetPlatformInfo( platforms[i], CL_PLATFORM_NAME, 0, 0,  platform_name_length);
clGetPlatformInfo(platforms[i], CL_PLATFORM_NAME, platform_name_length,
platform_name, 0);
cl_platform_id platform = platforms[selected_platform_index];
```

```
Number of available platforms: 1
Platform names:[0] Intel(R) OpenCL [Selected]
```


Example: step by step

Ex:
1.
2.
3.
4.
5.

```
cl_uint cur_num_of_devices;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 0, 0, &cur_num_of_devices);

cl_device_id* devices_of_type = new cl_device_id[cur_num_of_devices];
clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, cur_num_of_devices,
devices_of_type, 0);
```

```
CL_DEVICE_TYPE_CPU: 1
```

```
cl_uint device_index = 0;
cl_device_id device = devices_of_type[device_index];
```

```
CL_DEVICE_TYPE_CPU[0]
CL_DEVICE_NAME: Genuine Intel(R) CPU @ 2.60GHz
CL_DEVICE_AVAILABLE: 1
CL_DEVICE_VENDOR: Intel(R) Corporation
```

Example: step by step

- Create a simple context with a single device:

```
cl_context clCreateContext(cl_context_properties *properties, cl_uint
num_devices, cl_device_id *devices, (void CL_CALLBACK
*notify_func)(…), void user_data, cl_int *error);
```

Ex:
1. ✓
2.
3.
4.
5.

```
context = clCreateContext(NULL, 1, &device_id, NULL,
NULL, &err);
```

- Create a simple command-queue to feed our device:

```
cl_command_queue clCreateCommandQueue(cl_context context,
cl_device_id device_id, cl_command_queue_properties properties,
cl_int *error);
```

```
q_commands = clCreateCommandQueue(context, device_id, 0,
&err);
```


Example: step by step

```
const char* raw_text = &program_text_prepared[0];
cl_int err;
cl_program program = clCreateProgramWithSource(context,
1, &raw_text, 0, &err);

clBuildProgram(program, (cl_uint)num_of_devices, devices,
build_options.c_str(), 0, 0);
```

Ex:
1. ✓
2. ✓
3.
4.
5.

```
Build program options: "-DT=float -DTILE_SIZE_M=1 -DTILE_GROUP_M=16
-DTILE_SIZE_N=128 -DTILE_GROUP_N=1 -DTILE_SIZE_K=8"
```


Example: step by step

```
cl_kernel krnl = 0;
string kernel_name = "Multiply"
krnl = clCreateKernel(program, kernel_name.c_str(),
&err);
```

Ex:
1. ✓
2. ✓
3.
4. ✓
5.

- As we will see later, kernels are really like functions.
- They have arguments. But must return void.
- They are identified by name for OpenCL.
- Remember that kernels are compiled for the device architecture.

Example: step by step

- HOST MEMORY BUFFER:

```
cl_float* data_ptr = (cl_float *) malloc(sizeof(cl_float) * count);
cl_mem array_of_floats = clCreateBuffer(context, CL_MEM_READ_WRITE,
sizeof(cl_float)*count, data_ptr, NULL);
```

- KERNEL ARGUMENTS :

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), &array_of_floats);
```

Ex:
1. ✓
2. ✓
3. ✓
4. ✓
5.

Example: step by step

```
Remember the variables:
cl_command_queue q_commands;
cl_kernel krnl;
cl_mem array_of_floats;

Ex:
1. ✓
2. ✓
3. ✓
4. ✓
5. ✓

1. clEnqueueWriteBuffer(q_commands, array_of_floats, CL_FALSE, 0,
   sizeof(cl_float)*count, data_ptr,0, NULL, NULL);

2. clEnqueueTask(q_commands, krnl, 0, NULL,NULL); NO dimensions!

3. clEnqueueReadBuffer(q_commands, array_of_floats, CL_TRUE,
   sizeof(cl_float)*count, data_ptr, 0, NULL, NULL);
```


OPENCL KERNEL PROGRAMMING

OpenCL Kernel Programming

- Derived from **ISO C99 + ISO C11**
  - A few ***restrictions***: no recursion, function pointers,...
  - Preprocessing directives defined by C99 are supported (#include etc.)
- Built-in data types
  - Scalar and vector data types, pointers
  - Image types:
    - image2d_t, image3d_t and sampler_t
- OpenCL #pragmas added to guide the compiler.
- The return type of a kernel function must be void

OpenCL Kernel Programming

- Function qualifiers
  - **__kernel** qualifier declares a function as a kernel
    - I.e. makes it visible to host code so it can be enqueued.
  - Kernels can call other kernel-side functions
- Address space **QUALIFIERS**
  - **__global, __local, __constant, __private**
  - Pointer kernel arguments must be declared with an address space qualifier
- Work-item functions
  - get_work_dim(), get_global_id(), get_local_id(), get_group_id()
- Synchronization functions
  - **Barriers** - all work-items within a work-group must execute the barrier function before any work-item can continue
  - **Memory fences** - provides ordering between memory operations

OpenCL Kernel Programming

- Pointers to functions are ***not*** allowed
- Pointers to pointers allowed ***within*** a kernel, but not as an argument to a kernel invocation
- Bit-fields are not supported
- Variable length arrays and structures are not supported
- Recursion is not supported (yet!)
- Double types are optional in OpenCL v1.1, but the key word is reserved
  (note: most implementations support double)

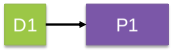
OpenCL Kernel Programming

- Built-in functions — ***mandatory***
  - Work-Item functions, math.h, read and write image
  - Relational, geometric functions, synchronization functions
  - printf ()
- Built-in functions — ***optional*** (called “extensions”)
  - Double precision, atomics to global and local memory
  - Selection of rounding mode, writes to image3d_t surface

FPGA ORIENTED KERNEL DESIGN

FPGA Oriented Kernel Design

- Some general rules for FPGAs:
- Work-Item and a kernel.

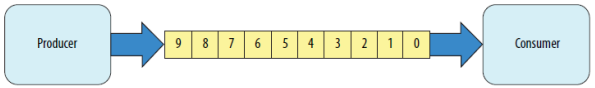


Strategy	Scheme	AREA	FREQ	THROUGHPUT	LATENCY
Parallelizing	D1 → P1 D2 → P1	++	=	++	=
Pipelining	D1 P1.1 P1.2 D2 P1.1	+	++	+	+
Complex op.	D1 → P1+P2+P3	=	--	++	--
Divide op.	D1 → P1 → P2 → P3	++	++	=	++

FPGA Oriented Kernel Design

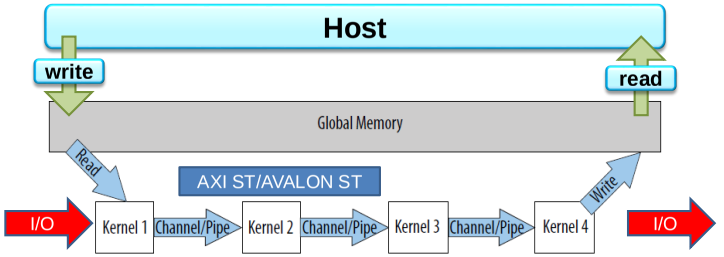
The most important FPGA design patern

```
#pragma OPENCL EXTENSION cl_intel_channels : enable
channel int c0;

__kernel void producer() {
    for (int i = 0; i < 10; i++) {
        write_channel_intel (c0, i);
    }
}

__kernel void consumer (__global uint * restrict dst) {
    for (int i = 0; i < 5; i++) {
        dst[i] = read_channel_intel(c0);
    }
}
```



from IntelFPGA programming guide

FPGA Oriented Kernel Design



- Send Data from one kernel to another without host intervention
- Send Data from I/O to kernel or from kernel to I/O
- Send Data from host to kernel and vice versa without using global memory
- Data remains in a channel as long as the kernel remains loaded on the FPGA device, persistence among NDRange invocations and among work-groups
- Blocking and Non-Blocking behavior
- Pipes are OpenCL 2.0 standard, contrary to channels

Comments on SYCL

- SYCL. cross-platform abstraction C++ programming model for **OpenCL**.
  - Adding much of the ease of use and flexibility of single-source C++.
- SYCL implements a single-source multiple compiler-passes (SMCP).
  - Simplifying the **Device-Host** separation of OpenCL.
- Easier to handle for programmers. But OpenCL concepts remain, i.e.

• The SYCL Platform Model • SYCL Execution Mode (command groups) • Memory Model (same 4 layers) • The SYCL programming model	○ cl::sycl::platform ○ cl::sycl::context ○ cl::sycl::device ○ cl::sycl::program ○ cl::sycl::queue ○ class kernel

As a summary, is just simpler …
for programers.

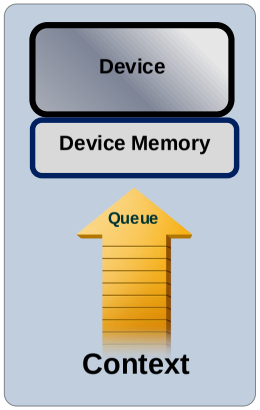
END OF INTRODUCTION

ADVANCED:

MORE ON OPENCL HOST PROGRAMMING

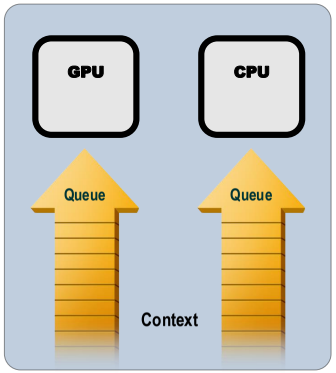
Context and Command-Queues

- ***Context***:
  - The environment within which kernels execute and in which synchronization and memory management is defined.
- The ***context*** includes:
  - One or more devices
  - Device memory
  - One or more command-queues
- All ***commands*** for a device (kernel execution, synchronization, and memory transfer operations) are submitted through a ***command-queue***.
- Each ***command-queue*** points to a single device within a context.



Command-Queues

- Commands include:
  - Kernel executions
  - Memory object management
  - Synchronization
- The only way to submit commands to a device is through a command-queue.
- Each command-queue points to a single device within a context.
- Multiple command-queues can feed a single device.
  - Independent streams of commands that don't require synchronization.



Command-Queue execution details

Command queues can be configured in different ways to control how commands execute

- ***In-order queues***:
  - Commands are enqueued and complete in the order they appear in the program (program-order)
- ***Out-of-order queues***:
  - Commands are enqueued in program-order but can execute (and hence complete) in any order.
- Execution of commands in the command-queue are guaranteed to be completed at synchronization points
  - Discussed later

What do we put in device memory?

Memory Objects:

- A handle to a reference-counted region of global memory.

There are two kinds of memory object

- ***Buffer*** object:
  - Defines a linear collection of bytes (“*just a C array*”).
  - The contents of buffer objects are fully exposed within kernels and can be accessed using pointers
- ***Image*** object:
  - Defines a two- or three-dimensional region of memory.
  - Image data can only be accessed with read and write functions, i.e. these are opaque data structures. The read functions use a sampler.
- ***Pipe*** (Channel) object:
  - A pipe is a memory object that stores data organized as a FIFO.
  - Pipe objects are not accessible from the host.

Memory Object Options

Flag value	Meaning
CL_MEM_READ_WRITE	The memory object can be read from and written to.
CL_MEM_WRITE_ONLY	The memory object can only be written to.
CL_MEM_READ_ONLY	The memory object can only be read from.
CL_MEM_USE_HOST_PTR	The memory object will access the memory region specified by the host pointer.
CL_MEM_COPY_HOST_PTR	The memory object will set the memory region specified by the host pointer.
CL_MEM_ALLOC_HOST_PTR	A region in host-accessible memory will be allocated for use in data transfer.

- These are from the point of view of the **device.**

Conventions for naming buffers

- **It can get confusing about whether a host variable is just a regular C array or an OpenCL buffer**
- **A useful convention is to prefix the names of your regular host C arrays with “h_” and your OpenCL buffers which will live on the device with “d_”**

Tasks and NDRange

- Enqueue the kernel for execution:

```
clEnqueueTask(cl_command_queue commands, cl_kernel kernel, cl_uint
num_events, const cl_event *wait_list, cl_event event);

clEnqueueNDRangeKernel (cl_command_queue commands, cl_kernel kernel,
cl_uint work_dims, size_t *global_work_offset, size_t
global_work_size, size_t *local_work_size,
 0, NULL, NULL);

clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global, &local,
 0, NULL, NULL);
```

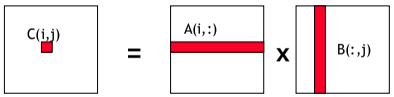
Events later.

MATRIX MULTIPLICATION EXAMPLE

Example: Linear Algebra

Analize C=A·B

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} & B_{02} \\ B_{10} & B_{11} & B_{12} \\ B_{20} & B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{00} & C_{01} & C_{02} \\ C_{10} & C_{11} & C_{12} \\ C_{20} & C_{21} & C_{22} \end{pmatrix}$$



Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
              // C(i, j) = sum(over k) A(i,k) * B(k,j)
              C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```

Matrix multiplication: OpenCL kernel (1/2)

```
__kernel void mat_mul(
 const int N,
__global float *A, __global float *B, __global float *C)
    int i, j, k;
    for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
        // C(i, j) = sum(over k) A(i,k) * B(k,j)
        for (k = 0; k < N; k++) {
          C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```

Mark as a kernel function and specify memory qualifiers

Matrix multiplication: OpenCL kernel (2/2)

```
__kernel void mat_mul(
  const int N,
  __global float *A, __global float *B, __global float *C)
{
    int i, j, k;
    i = get_global_id(0);
    j = get_global_id(1);
        for (k = 0; k < N; k++) {
            // C(i, j) = sum(over k) A(i,k) * B(k,j)
            C[i*N+j] += A[i*N+k] * B[k*N+j];
        }
    }
}
```

Remove outer loops and set work-item co-ordinates

Matrix multiplication: OpenCL kernel

```
__kernel void mat_mul(
 const int N,
 __global float *A, __global float *B, __global float *C)
{
    int i, j, k;
    i = get_global_id(0);
    j = get_global_id(1);
    // C(i, j) = sum(over k) A(i,k) * B(k,j)
    for (k = 0; k < N; k++) {
      C[i*N+j] += A[i*N+k] * B[k*N+j];
    }
}
```


Matrix multiplication: OpenCL kernel improved

- Rearrange and use a local scalar for intermediate C element values (a common optimization in Matrix Multiplication functions)

```
__kernel void mmul(
    const int N,
    __global float *A,
    __global float *B,
    __global float *C)
{
  int k;
  int i = get_global_id(0);
  int j = get_global_id(1);
  float tmp = 0.0f;
  for (k = 0; k < N; k++)
   tmp += A[i*N+k]*B[k*N+j];
  }
  C[i*N+j] += tmp;
}
```

This Accumulation is recognized by the compiler !

MORE ON FPGA ORIENTED KERNEL DESIGN

Single Work-Item vs NDRange Kernel

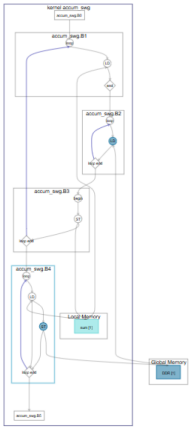
- Intel recommends single work-Item kernels, when possible
- Use NDRange when the code does not have memory dependencies and loops. If data must be shared among WI this structure is not efficient.
- High throughput achieved by using multiple pipelines stages at any time. Parallelism by pipelining the loop iterations.

- Some strategies are common to FPGAs, others depend on the family of FPGA, consult the programming guide for each one.

Single Work-Item vs NDRange Kernel

Single Work-Item kernel

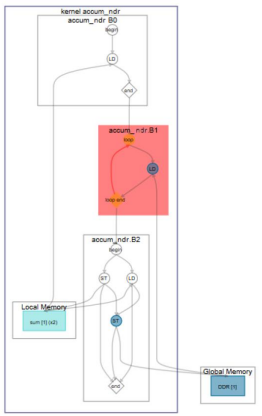
```
1 kernel void accum_swg (global int* a,
                         global int* c,
                         int size,
                         int k_size) {
2     int sum[1024];
3     for (int k = 0; k < k_size; ++k) {
4        for (int i = 0; i < size; ++i) {
5            int j = k * size + i;
6            sum[k] += a[j];
7        }
8     }
9     for (int k = 0; k < k_size; ++k) {
10       c[k] = sum[k];
11    }
12 }
```



Single Work-Item vs NDRange Kernel

NDRange Kernel

```
kernel void accum_ndr (global int* a,
                       global int* c,
                       int size) {
   int k = get_global_id(0);

   int sum[1024];
   for (int i = 0; i < size; ++i) {
     int j = k * size + i;
     sum[k] += a[j];
   }
   c[k] = sum[k];
}
```



Optimizing Data Processing Efficiency

Strategy 1: Unrolling a Loop

#pragma unroll <N>

```
#pragma unroll 2
for(size_t k = 0; k < 4; k++)
{
   mac += data_in[(gid * 4) + k] * coeff[k];
}
```


Optimizing Data Processing Efficiency

Strategy 2: Coalescing Nested Loops

#pragma loop_coalesce <loop_nesting_level>

The OpenCL compiler hates nested loops. Try to avoid its use!!

```
#pragma loop_coalesce
for (int i = 0; i < N; i++)
  for (int j = 0; j < M; j++)
    sum[i][j] += i+j;
```

```
int i = 0;
int j = 0;
while(i < N){

  sum[i][j] += i+j;
  j++;

  if (j == M){
    j = 0;
    i++;
  }
}
```


Optimizing Data Processing Efficiency

Strategy 3: Specifying a Loop Initiation Interval

#pragma ii <desired_initiation_interval>

Define the number of clock cycles to wait among successive loop iterations

Strategy 4: Loop Concurrency

#pragma max_concurrency <N>

Define the number of iterations to be in progress at one time

Optimizing Data Processing Efficiency

Strategy 5: Specifying the work group size

```
__attribute__((max_work_group_size(512,1,1)))
__kernel void sum (__global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict answer)
{
    size_t gid = get_global_id(0);
    answer[gid] = a[gid] + b[gid];
}
```

```
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void sum (__global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict answer)
{
    size_t gid = get_global_id(0);
    answer[gid] = a[gid] + b[gid];
}
```

Optimizing Data Processing Efficiency

Strategy 6: Specifying the Number of Compute Units

```
__attribute__((num_compute_units(2)))
__kernel void test(__global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict answer)
{
   size_t gid = get_global_id(0);
   answer[gid] = a[gid] + b[gid];
}
```

Strategy 7: Specifying the Number of SIMD Work-Items

However. Use vectors explicitly as frequently as possible

```
__attribute__((num_simd_work_items(4)))
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void test(__global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict answer)
{
   size_t gid = get_global_id(0);
   answer[gid] = a[gid] + b[gid];
}
```

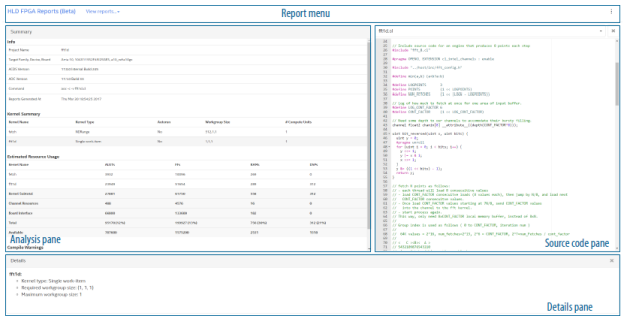

Optimizing Data Processing Efficiency

Strategy 8: Removing Loop-Carried Dependencies

#pragma ivdep

When each loop access different parts of the array there may be fictitious dependencies. This directive commands the compiler to forget the dependencies and remove the extra initiation cycles in the loop immediately after the pragma directive.

Compilation Report



Report about code structures that prevent the loops from being fully pipelined.
Report about area usage.
Report about wrong memory management.

More…

Strategy 9: Implementing Arbitrary Precision Integers

Sometimes, optimizing the code when working with FPGAs demands adjusting the size of data to the size strictly needed.

Strategy 10: Inferring Registers and Shift Registers.

When variables are defined as “private” and the access to arrays are statically inferable, they are implemented as FFs in LEs, or in blocks RAM (if their size is larger than 64 bytes). They are the fastest hardware for loop execution.

Strategy 11: Inferring Single-Cycle Floating-Point Accumulator.

Only for Arria10 devices.

… and more strategies

Create good and efficient code for FPGA kernels is a complex task. The results depends heavily on the designer's expertise.

However, it's a perfect beginning to read two very useful manuals.

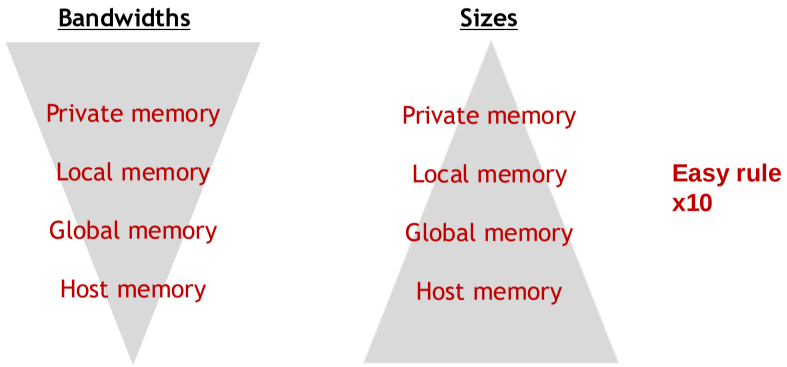
"Intel FPGA SDK for OpenCL Programming Guide" and
"Intel FPGA SDK for OpenCL Best Practices Guide"

They are hard to deal with though they will give you priceless help for optimizing the area and speed of your application parting from the information given by the compilation report.

They will show you how to work with loop-carried dependency, how to carry out proper memory management to improve access, how to use channels or pipes when needed, etc.

THE OPENCL MEMORY HIERARCHY

The Memory Hierarchy



Private Memory

- Managing the memory hierarchy is one of ***the*** most important things to get right to achieve good performance.
- Remember memory transfers are explicit!

- Private Memory:
  - A **very scarce** resource, only a few thousands of 32-bit words per Work-Item at most
  - If you use too much it spills to global (or local) memory or reduces the number of Work-Items that can be run at the same time, potentially harming performance
  - These is the closest-to-hardware memory. The actual realization varies from one to another. (CPU registers, FPGA registers,…)

Local Memory

- The memory for the work-groups:
  - Close to the hardware, but shared between work-items. Each device realizes it in a different way.
  - Your kernels are responsible for transferring data between Local and Global/Constant memories
- Access patterns to Local Memory affect performance in a similar way to accessing Global Memory.
- Due to their architecture, managing local memory is most important to GPUs.
- FPGA local memory is still very fast (private), but being shared means access patterns affect it.
- CPUs do not have specialized hardware for this….

Global Memory

- The host accessible memory.
- The access pattern to global memory should be the easiest possible.
  - Move data to faster memories, think about dependencies of the algorithm.
- Constant memory is an specialization of global.
- In FPGAs global memory is RAM outside the chip.
  - Constant memory might get replicated or cached to chip memory to satisfy reading needs.
  - While the bandwidth is good, the FPGA can easily overwhelm it with read and write operations.

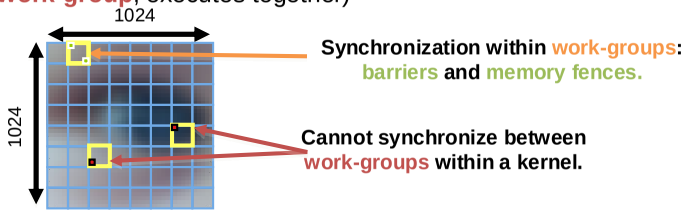
Memory Consistency

- OpenCL uses a **relaxed consistency** memory model; i.e.
  - The state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.
- Within a work-item:
  - Memory has load/store consistency to the work-item's private view of memory, i.e. it sees its own reads and writes correctly
- Within a work-group:
  - Local memory is consistent between work-items at a barrier.
- Global memory is consistent within a work-group at a barrier, **but *not* guaranteed across different work-groups!!**
  - This is a common source of bugs!
- Consistency of memory shared between commands (e.g. kernel invocations) is enforced by synchronization (barriers, events, in-order queue)

SYNCHRONIZATION IN OPENCL

Consider N-dimensional domain of work-items

- Global Dimensions:
  - 1024x1024 (whole problem space)
- Local Dimensions:
  - 64x64 (**work-group**, executes together)



Synchronization: when multiple units of execution (e.g. work-items) are brought to a known point in their execution. Most common example is a barrier … i.e. all units of execution "in scope" arrive at the barrier before any proceed.

Work-Item Synchronization

- Use barrier to synchronize work items inside a work-group.
- **barrier**(CLK_LOCAL_MEM_FENCE) **or barrier**(CLK_GLOBAL_MEM_FENCE)
- Careful with branching! All the work items must take the same branch.
- Across work-groups
  - No guarantees as to where and when a particular work-group will be executed relative to another work-group
  - Cannot exchange data, or have barrier-like synchronization between two different work-groups! (Critical issue!)
  - Only solution: finish the kernel and start another
- The FPGA being hardware, does have other means to synchronize (pipes).

Where might we need synchronization?

- Consider a reduction … reduce a set of numbers to a single value
  - E.g. find sum of all elements in an array
- Sequential code

```
int reduce(int Ndim, int *A)
{
  int sum = 0;
  for (int i = 0; i < Ndim; i++)
    sum += A[i];
  return sum;
}
```


Simple parallel reduction

- A reduction can be carried out in three steps:
  1. Each work-item sums its private values into a local array indexed by the work-item's local id
  2. When all the work-items have finished, one work-item sums the local array into an element of a global array (indexed by work-group id)
  3. When all work-groups have finished the kernel execution, the global array is summed on the host

Again, the dimensionality of the problem regarding performance depends on the hardware!

THE OPENCL EVENT MODEL

OpenCL Kernel life cycle

- An event is an object that communicates the status of commands in OpenCL … legal values for an event:
  - CL_QUEUED: command has been enqueued.
  - CL_SUBMITTED: command has been submitted to the compute device
  - CL_RUNNING: compute device is executing the command
  - CL_COMPLETE: command has completed
  - ERROR_CODE: a negative value indicates an error condition occurred.
- Can query the value of an event from the host … for example to track the progress of a command.

```
cl_int clGetEventInfo (
          cl_event event,     cl_event_info param_name,
          size_t param_value_size, void *param_value,
          size_t *param_value_size_ret)
```

Examples:
- CL_EVENT_CONTEXT
- CL_EVENT_COMMAND_EXECUTION_STATUS
- CL_EVENT_COMMAND_TYPE

Generating and consuming events

- Consider the command to enqueue a kernel. The last three arguments optionally expose events (NULL otherwise).

```
cl_int clEnqueueNDRangeKernel (
      cl_command_queue command_queue,
      cl_kernel kernel,
      cl_uint work_dim,
      const size_t *global_work_offset,
      const size_t *global_work_size,
      const size_t *local_work_size,
      cl_uint num_events_in_wait_list,
      const cl_event *event_wait_list,
      cl_event *event)
```

Number of events this command is waiting to complete before executing

Array of pointers to the events being waited upon … Command queue and events must share a context.

Pointer to an event object generated by this command

Event: basic event usage

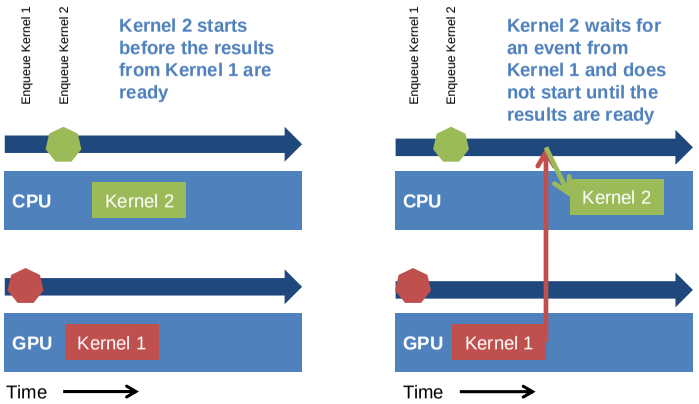
- Events can be used to impose order constraints on kernel execution.
- Very useful with out-of-order queues.

```
cl_event     k_events[2];

err = clEnqueueNDRangeKernel(commands, kernel1, 1,
 NULL, &global, &local, 0, NULL, &k_events[0]);

err = clEnqueueNDRangeKernel(commands, kernel2, 1,
 NULL, &global, &local, 0, NULL, &k_events[1]);

err = clEnqueueNDRangeKernel(commands, kernel3, 1,
 NULL, &global, &local, 2, k_events, NULL);
```

Enqueue two kernels that expose events

Wait to execute until two previous events complete

OpenCL synchronization: queues & events



Profiling with Events

- OpenCL is a performance oriented language … Hence performance analysis is an essential part of OpenCL programming.
- The OpenCL specification defines a portable way to collect profiling data.
- Can be used with most commands placed on the command queue … includes:
  - Commands to read, write, map or copy memory objects
  - Commands to enqueue kernels, tasks
- Profiling works by turning an event into an opaque object to hold timing data.

Using the Profiling interface

- Profiling is enabled when a queue is created with the CL_QUEUE_PROFILING_ENABLE flag set.
- When profiling is enabled, the following function is used to extract the timing data

Profiling data to query (see next slide)

```
cl_int clGetEventProfilingInfo(
        cl_event event,
        cl_profiling_info param_name,
        size_t param_value_size,
        void *param_value,
        size_t *param_value_size_ret)
```

Expected and actual size of profiling data.

Pointer to memory to hold results

cl_profiling_info values

- CL_PROFILING_COMMAND_QUEUED
  - the device time in nanoseconds when the command is enqueued in a command-queue by the host. (cl_ulong)
- CL_PROFILING_COMMAND_SUBMIT
  - the device time in nanoseconds when the command is submitted to compute device. (cl_ulong)
- CL_PROFILING_COMMAND_START
  - the device time in nanoseconds when the command starts execution on the device. (cl_ulong)
- CL_PROFILING_COMMAND_END
  - the device time in nanoseconds when the command has finished execution on the device. (cl_ulong)

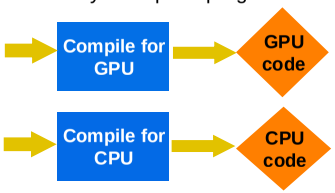
BUILDING OPENCL FOR FPGA

Building Program Objects

- The program object encapsulates:
  - A context
  - The program kernel source or binary
  - List of target devices and build options
- The C API creates a program object:
  - clCreateProgramWithSource()
  - clCreateProgramWithBinary()

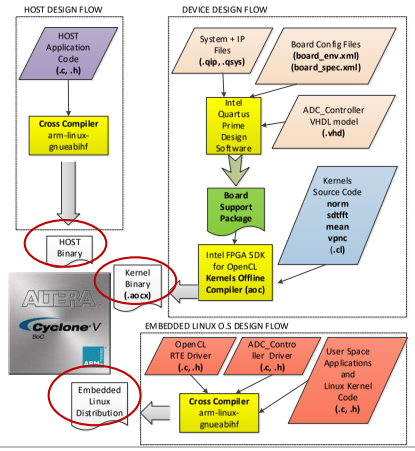
FPGAs:
OpenCL DOESN'T use runtime compilation

CPUs and GPUs:
OpenCL uses runtime compilation … because in general you don't know the details of the target device when you ship the program



OpenCL FPGAs Development

- Example of heterogeneous system:
- SoC with:
- FPGA (Altera)
- CPU (arm).



Final remarks

- Remember the models to divide the computation the OpenCL way!
- The Host is controlling the computation of one or multiple heterogeneous devices.
- The host communicates using commands in command-queues.
- The 4 layers of memory. Memory transfers are explicit!
- There are lots of layers of parallelism
- Synchronize your work-items.
- Use events and profile them to monitor performance.
- Building an OpenCL application requires multiple compilations (min. 2)

Thank you!

Special thanks to Dr. Antonio Carpeño and Professor Mariano Ruiz for their guidance.

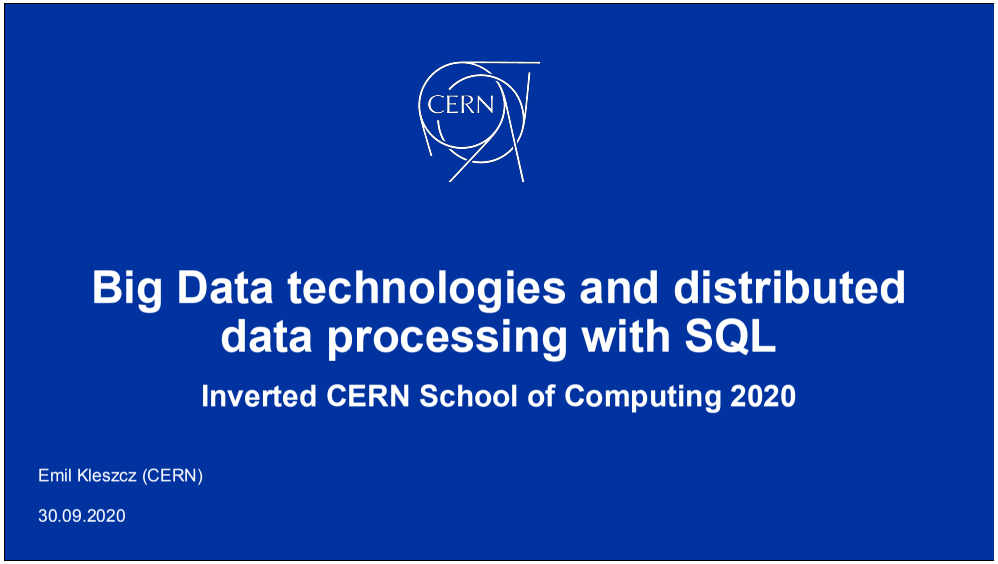
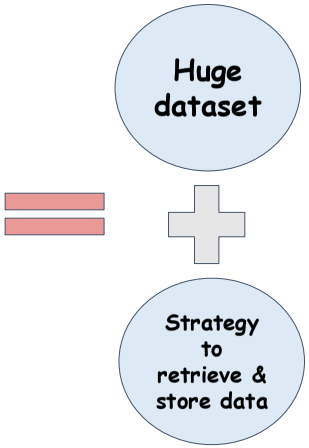
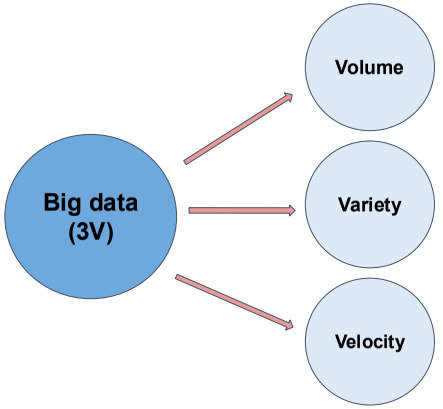


Table of contents

Introduction to Big Data



What is Big Data?



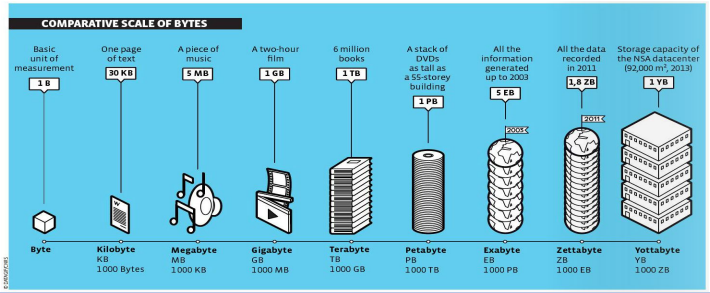
- Scale of data
- Large volume: TB,PB, etc.
- Size, records, transactions, tables, etc.

- Different forms of data
- Multiple data sources
- Type of data: structured, unstructured, etc.

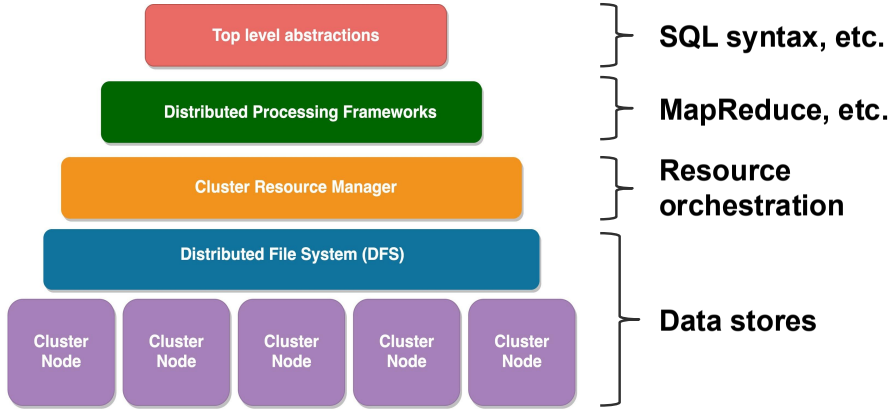
Frequency of updates:
- Batch processing
- Stream processing
- Real-time processing

Big Data history & facts

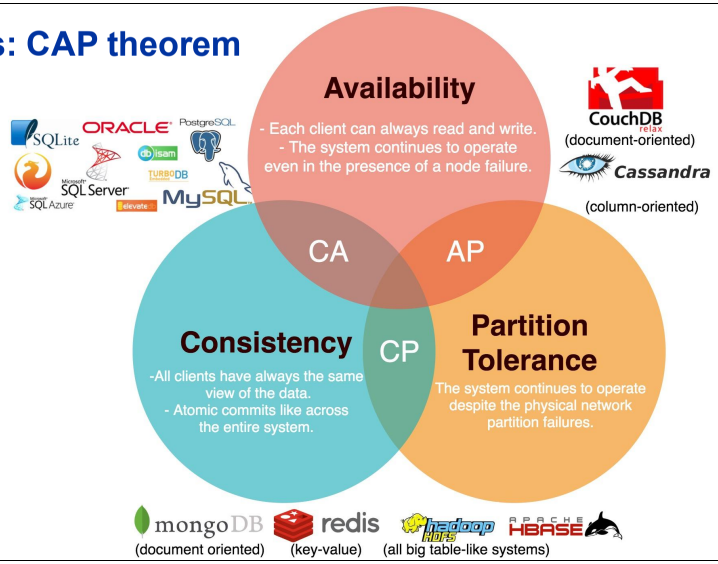
- 2004 - **MapReduce**: Simplified Data Processing on Large Clusters by Google.
- 2005 - **Hadoop** created by Yahoo & built on top of Google's MapReduce.
- 2008 - Google **processes 20PB of data in one day**.
- **90% of data created in last 2 years**.
- 4.4ZB in 2013, **now ~15ZB yearly**, expected.
- 44ZB in 2020 (1ZB = 10^21B).
- The whole universe can contain ~10^124 objects (entropy of black holes).



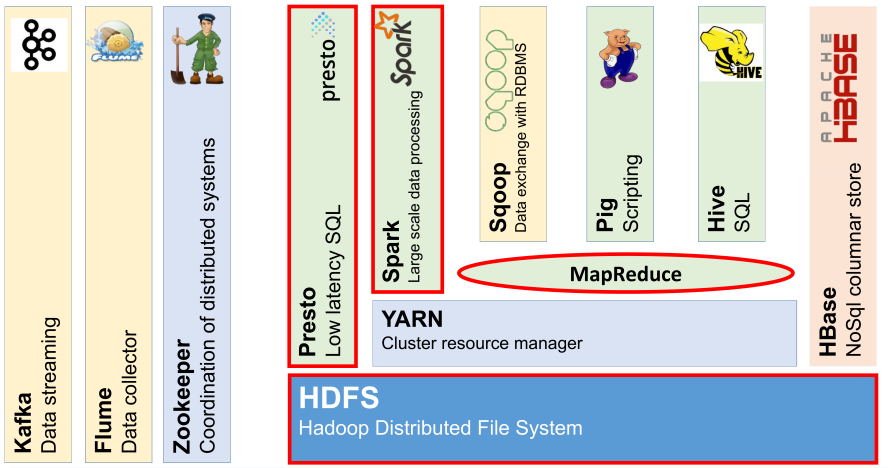
Architecture overview



Data models: CAP theorem

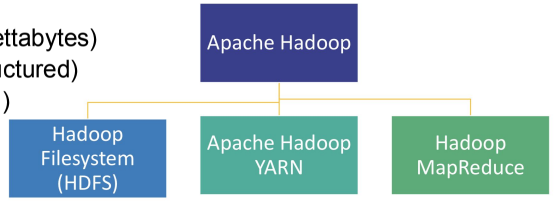


Big Data ecosystem



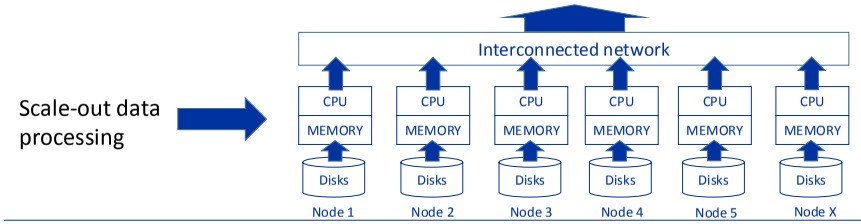
Hadoop ecosystem

- Started at Yahoo in 2006 based on **Google File System and MapReduce** from 2003-2004
- A framework for **large scale data processing**
  - Open source
  - Written in **Java**
  - To be run on a **commodity hardware**
- **3Vs of Big Data:**
  - Data **V**olume (Terabytes, … , Zettabytes)
  - Data **V**ariety (Structured, Unstructured)
  - Data **V**elocity (Batch processing)



Distributed system for data processing

- Split and distribute data across many machines **(sharding)**
- Storage with multiple data processing interfaces
- Operates at scale by design **(shared nothing - scales out)**
- Typically on clusters of **commodity-type servers/cloud**
- Well established in the industry (**open source**)
- **Distributed data processing**
  - Fast parallel data scanning
  - Profit from **data locality** - high throughput between storage, CPU & Memory

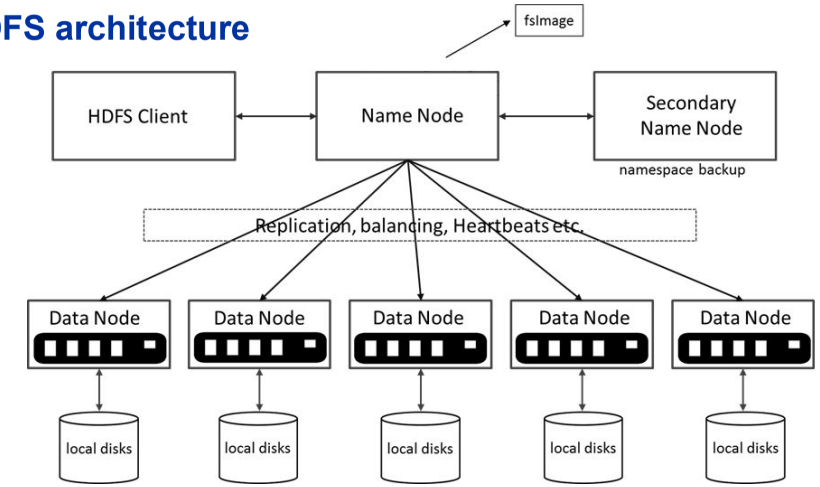


Hadoop Distributed File System (HDFS)

- **HDFS characteristics**
  - **Fault-tolerant:** multiple copies of data, or Erasure Coding (RAID 5/6, XOR-like)
  - **Scalable** - design to deliver high throughputs, sacrificing access latency
  - Files cannot be modified in place (**Write once - Read Many**)
  - **Permissions** on files and folders like in **POSIX**, also additional ACLs can be set
  - **Minimal data motion** and rebalance
- **HDFS architecture:**
  - Cluster with **master-slave architecture**
    - **Name Node**(s) (1 or more per cluster) - maintains & manages file system metadata (in RAM)
    - **Data Nodes** (many per cluster) - store & manipulate the data (blocks)
- **Ways of accessing and processing data**
  - Can be mounted with Fuse (with fstab entry)
  - Programming bindings: Java, Scala, Python, C++
  - HDFS has web UI where its status can be tracked
    - http://namenode:50070

```
hdfs dfs -ls                #listing home dir
hdfs dfs -ls /user          #listing user dir…
hdfs dfs -du -h /user       #space used
hdfs dfs -mkdir newdir      #creating dir
hdfs dfs -put myfile.csv .  #storing a file on HDFS
hdfs dfs -get myfile.csv .  #getting a file from HDFS
```

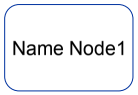

HDFS architecture



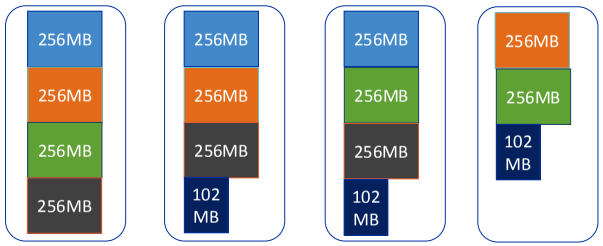
How HDFS stores the data

1. File to be stored on HDFS of size 1126MB (split into 256MB blocks)

2. Ask Name Node where to put the blocks



3. Blocks with their replicas (by default 3) are distributed across Data Nodes



What to use Hadoop for?

- **Big Data storage** with **HDFS** and **big data volumes** with **MapReduce**
- Strong for **batch processing at scale**
  - Data exploration (ad-hoc), reporting, statistics, aggregations, correlation, ML, BI
- **Hadoop is On-Line Analytical Processing** (OLAP)
  - no real-time data but historical or old data moved in batches
- **Write once - read many**
  - no data modifications allowed only appends
- **Typical use cases:**
  - Storing and analysing systems' logs, time series data at big scale
  - Building data warehouses/lakes for structured data
  - Data preparation for Machine Learning

… and not use Hadoop for:

- **Weak for Online Transaction Processing** system (OLTP)
  - No data updates (only appends and overwrites)
  - Typically response time in minutes rather milliseconds
- **Not optimal for systems with complex relational data**

Typical system based on Hadoop ecosystem

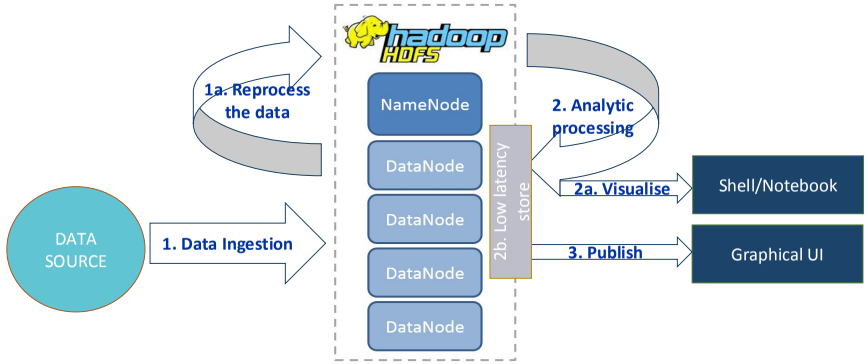
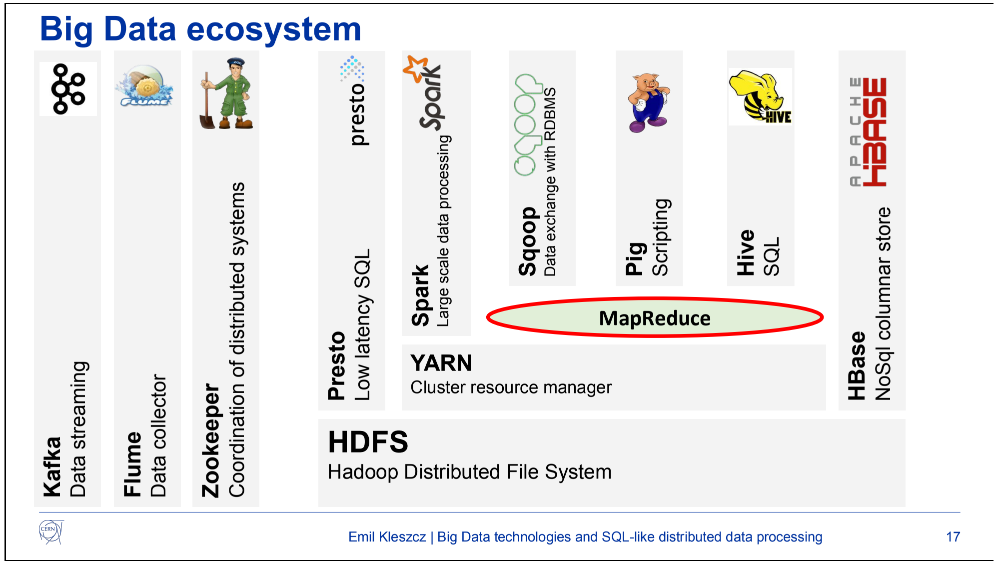
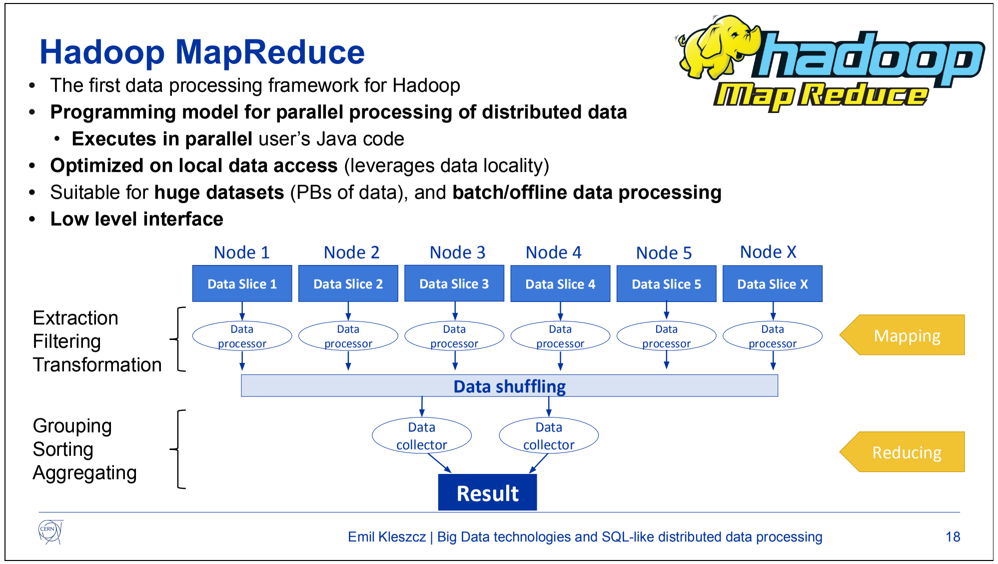
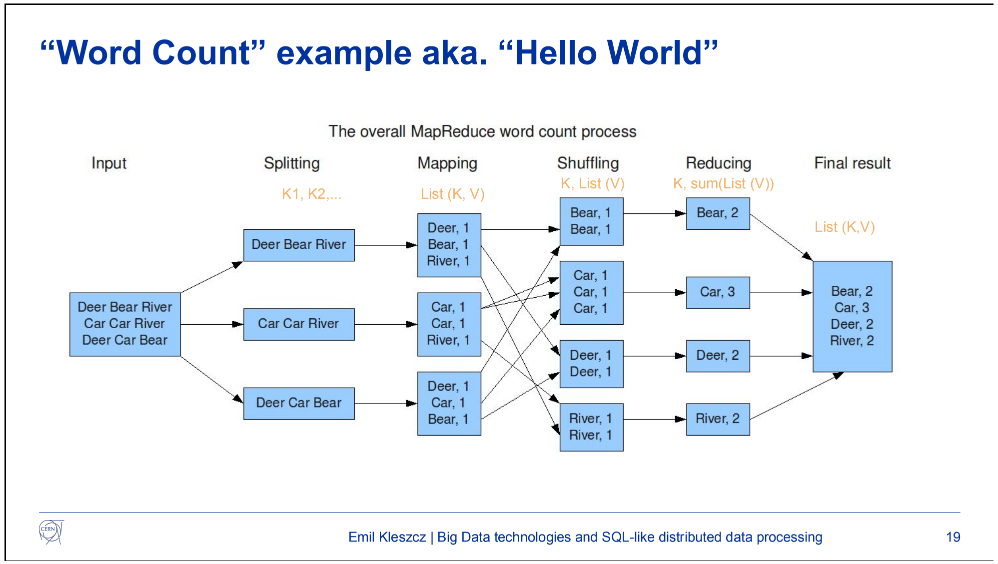
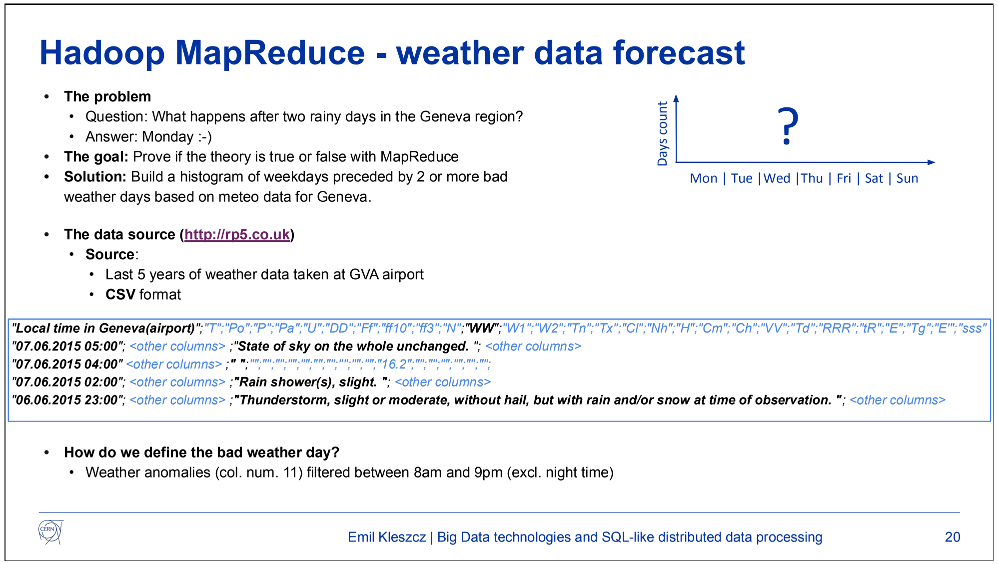


Table of contents


Big Data ecosystem
Kafka
Data streaming
Flume
Data collector
Zookeeper
Coordination of distributed systems
Presto
Low latency SQL
Spark
Large scale data processing
Sqoop
Data exchange with RDBMS
Pig
Scripting
Hive
SQL
HBase
NoSql columnar store
MapReduce
YARN
Cluster resource manager
HDFS
Hadoop Distributed File System
Emil Kleszcz | Big Data technologies and SQL-like distributed data processing
17


Hadoop MapReduce
The first data processing framework for Hadoop
Programming model for parallel processing of distributed data
Executes in parallel user's Java code
Optimized on local data access (leverages data locality)
Suitable for huge datasets (PBs of data), and batch/offline data processing
Low level interface
Node 1
Node 2
Node 3
Node 4
Node 5
Node X
Data Slice 1
Data Slice 2
Data Slice 3
Data Slice 4
Data Slice 5
Data Slice X
Extraction
Filtering
Transformation
Data processor
Mapping
Data shuffling
Grouping
Sorting
Aggregating
Data collector
Reducing
Result
Emil Kleszcz | Big Data technologies and SQL-like distributed data processing
18


"Word Count" example aka. "Hello World"
The overall MapReduce word count process
Input
Splitting
Mapping
Shuffling
Reducing
Final result
K1, K2,...
List (K, V)
K, List (V)
K, sum(List (V))
List (K,V)
Deer Bear River
Car Car River
Deer Car Bear
Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1
Bear, 1
Bear, 1
Car, 1
Car, 1
Car, 1
Deer, 1
Deer, 1
River, 1
River, 1
Bear, 2
Car, 3
Deer, 2
River, 2
Emil Kleszcz | Big Data technologies and SQL-like distributed data processing
19

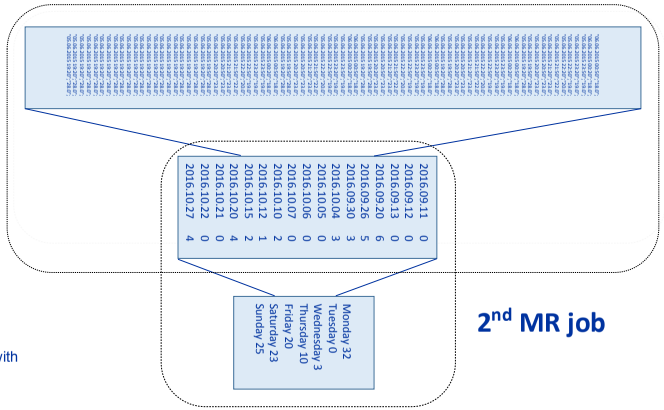

Hadoop MapReduce - weather data forecast
The problem
Question: What happens after two rainy days in the Geneva region?
Answer: Monday :-)
The goal: Prove if the theory is true or false with MapReduce
Solution: Build a histogram of weekdays preceded by 2 or more bad weather days based on meteo data for Geneva.
Days count
?
Mon | Tue | Wed | Thu | Fri | Sat | Sun
The data source (http://rp5.co.uk)
Source:
Last 5 years of weather data taken at GVA airport
CSV format
"Local time in Geneva(airport)";"T";"Po";"P";"Pa";"U";"DD";"Ff";"ff10";"ff3";"N";"WW";"W1";"W2";"Tn";"Tx";"Cl";"Nh";"H";"Cm";"Ch";"VV";"Td";"RRR";"tR";"E";"Tg";"E'";"sss"
"07.06.2015 05:00"; <other columns> ;"State of sky on the whole unchanged. "; <other columns>
"07.06.2015 02:00"; <other columns> ;"Rain shower(s), slight. "; <other columns>
"06.06.2015 23:00"; <other columns> ;"Thunderstorm, slight or moderate, without hail, but with rain and/or snow at time of observation. "; <other columns>
How do we define the bad weather day?
Weather anomalies (col. num. 11) filtered between 8am and 9pm (excl. night time)
Emil Kleszcz | Big Data technologies and SQL-like distributed data processing
20

Hadoop MapReduce - weather data forecast

Input Data:
Record: Weather report every hour

Reduced data:
Record: Date of good weather preceded by days of bad weather

Reduced data:
Record: Day of a week with counter of occurrences



1st MR job

2nd MR job

Weather forecast - 2nd MapReduce

Mapper

```
public static class ByDayMapper extends Mapper<LongWritable, Text,
IntWritable, IntWritable> {
  private IntWritable rKey = new IntWritable();
  private IntWritable rValue = new IntWritable();
  private Calendar c = Calendar.getInstance();
  private SimpleDateFormat dt = new SimpleDateFormat("yyyy.MM.dd");

  @Override
  protected void map(LongWritable key, Text value, Context context)
    throws Exception {
  // Splitting the line into columns by tab
   String[] split = value.toString().split("\t");
   try {
     // Only 2 columns expected
     if (split.length==2)
      {
        // Get a day of the week (num.) out of date (1st column)
        c.setTime(dt.parse(split[0]));
        rKey.set(c.get(Calendar.DAY_OF_WEEK));

        // Value is optional for our case
        rValue.set(1);

       // Emit kv for good weather day if preceded by 2>= bad days
       if (Integer.parseInt(split[1])>=2){
          context.write(rKey, rValue);
       }
     } catch (Exception e) {// ...}
   }
}
```

Reducer

```
public static class ByDayReducer<KEY> extends Reducer<KEY,
IntWritable, KEY, LongWritable>
{
  private LongWritable result = new LongWritable();
  public void reduce(KEY key, Iterable<IntWritable> values,
          Context context) throws Exception {
   // Counting all mapped pairs for given days of a week
   long sum = 0;
   for (IntWritable val : values) {
     ++sum; // or += val.get(); always 1
   }
   result.set(sum);
   // Emit the result
   context.write(key, result);
  }
}
```

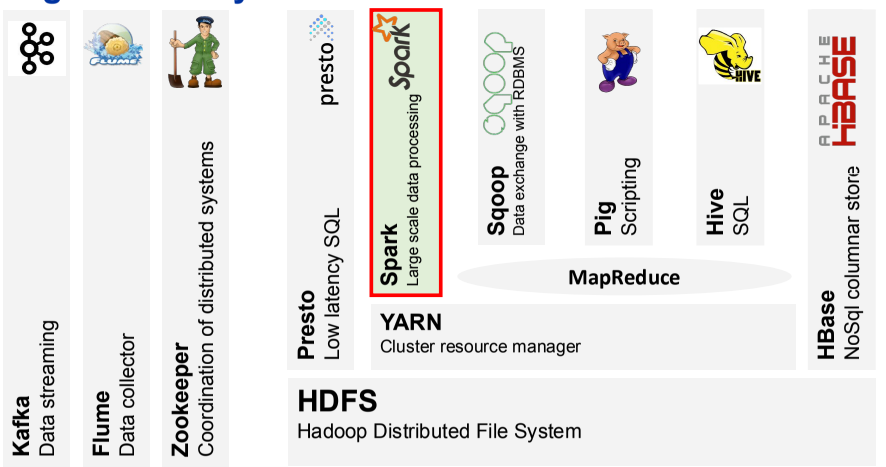
MapReduce run

```
public int run(String[] args) throws Exception {
  // Init the job
  Job job = Job.getInstance(getConf());
  job.setJarByClass(getClass());
  job.setJobName("Aggregating by week days");
  // Setting input/output paths
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  // Setting mapper and reducer class
  job.setMapperClass(ByDayMapper.class);
  job.setReducerClass(ByDayReducer.class);
  // Setting output types/classes
  job.setOutputKeyClass(IntWritable.class);
  job.setOutputValueClass(IntWritable.class);
  return job.waitForCompletion(true) ? 0 : 1;
}
```


Limitations of MapReduce

- **Not interactive**
  - Process of scheduling job takes significant amount of time
    - Negotiation with YARN, sending client code, application master has to setup (start JVM, etc.)
  - Typically separate executor for each data unit (e.g. HDFS block)
    - A lot of executors have to be started (JVM & local environment have to be setup), short life-time
- **Complex processing** requires to launch **multiple MR jobs**
  - Only 2 stages per job
  - Intermediate results have to be dumped to HDFS and it takes time
- **Each data processing task has to be implemented by a user**
  - Time consuming process especially for data exploration cases
- **What are the other more user friendly approaches?**

Big Data ecosystem



Spark as the next generation MapReduce

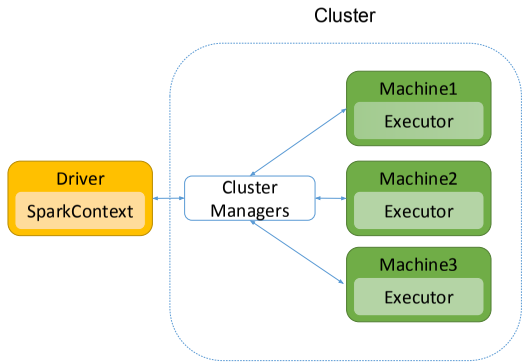
- **A framework for performing distributed computations**
- **Scalable** - applicable for processing TBs of data
- **User-friendly API**
- Supports **Java, Scala, Python, R** and **SQL**

- **Optimized for complex processing**
  - **Not using MapReduce**
  - Allows complex **Directed-Acyclic-Graph** of stages
  - **Staged data kept in memory**
  - **Long living executors**
    - processing multiple stages and jobs

- Varied APIs: **DataFrames, SQL, MLib, Streaming**
- Multiple computing resource schedulers supported
  - YARN, Kubernetes, Mesos
- Many **deployment modes** on Hadoop – **local,** and **cluster** on YARN
- **Multiple data sources:** HDFS, HBase, S3, JDBC...
- Various integrations available such as notebooks

Driver and executor concept in Spark

```
import scala.math.random

val slices = 3 # num of parallel executors
val n = 100000 * slices
val rdd = sc.parallelize(1 to n, slices)
val sample = rdd.map { i =>
  val x = random
  val y = random
  # Check if inside the circle
  if (x*x + y*y < 1) 1 else 0
}
val count = sample.reduce(_ + _)
# Geometric probability of a point inside the
square to lie inside the circle
println("Pi is roughly " + 4.0 * count / n)
```



SQL for the Big Data processing

- **SQL** is a **well-defined language standard** that exists since 1970s
  - Everyone is familiar with
  - Minimizes the learning curve of using different data processing tools
- It's a **syntax that is converted to the natively optimised code**
  - It's just **a way of expressing what you want to get** and not how you want to get it
- **Reduces the amount of code users need to write**
- **Allows performance optimizations transparent to the users**
  - SQL planner and query optimizer
- **Opens the door for leveraging & integrating lots of existing tooling**
- **Structured data are easy to understand and maintain**

```
UPDATE country

                    expression
SET population = population + 1
                 expression
WHERE name = 'USA' ;
             predicate
```

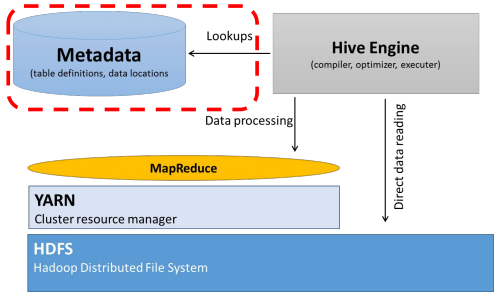
} statement

```
select count(*) from phoenix_hadoop3.aei.sevents;
select * from AEI.EVENTS limit 10;
select * from AEI.EVENTS where EVENTNUMBER=852298541;
```


SQL on HDFS needs Metastore

- Problem: **SQL needs tables but on HDFS we have only directories & files**
- Hive Metastore is **a relational database containing metadata about objects**
- Contains:
  - **Table definitions**
    - column names, data types, comments
  - **Data locations** - partitions
- Acts as a **central schema repository**
- Can be used by other access tools such as Spark, Presto, MapReduce etc.
- **Supports multiple file formats:**
  - Parquet, ORC, Text file, etc.
- **Tables can be partitioned**
  - each partition is a single HDFS directory

- **In practice** - 3 steps:
  - Create your own **Hive Metastore - database as a container for tables**
  - Define a table on top of your HDFS data
  - Run queries on tables with Spark, etc.



Spark SQL module

- **Module for structured data processing**
- **There are two ways to run Spark SQL:**
  - Spark SQL CLI (*./bin/spark-sql*) (easy to use SQL)
  - or DataFrame API with JDBC/Thrift Server
- **Spark SQL CLI**
  - Convenient tool to run the Hive Metastore service in local mode and execute queries input from the command line :-)
  - cannot talk to the Thrift JDBC server :-(
- **Limitation: Natively the data can only be read from Hive Metastore** (using *SparkSession*)
  - For other databases one needs to use JDBC protocol and Thrift server

Mixing SQL queries with Spark programs

```
# Apply functions to results of SQL queries
results = spark.sql("SELECT * FROM my_table")
names = results.map(lambda p: p.column_name)
```

Uniform data access: querying and joining different data sources

```
# Defining dataframe with schema from parquet files stored on hdfs
> val df = spark.read.parquet("/user/ekleszcz/datasets/")

# Counting the number of pre-filtered rows with DF API
> df.filter($"l1trigchainstap".contains("L1_TAU4")).count

# Counting the number of pre-filtered rows with SQL
> df.registerTempTable("my_table")
> spark.sql("SELECT count(*) FROM my_table where l1trigchainstap like '%L1_TAU40%'").show
```


Spark SQL - weather example

Read weather data from csv

```
val data = spark.read.format("csv").
           option("sep", ";").
           option("inferSchema", "true").
           option("header", "true").
           load("data/*")
```

Create a temporary table

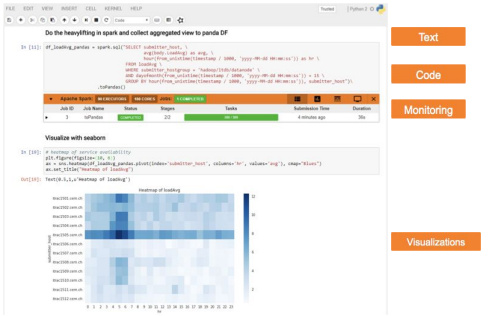
```
data.registerTempTable("weatherTable")
```

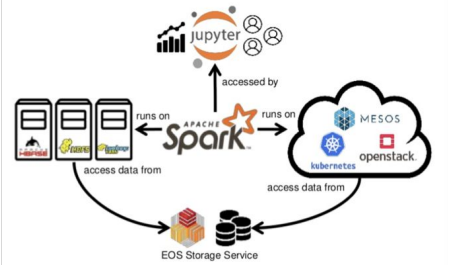
Query to compute sunny days after two rainy days

```
sql("
with source as (select [...] as time, ww as weather from weatherTable),
weather as (select time,[...] then 0 else 1 end bad_wather from source where hour(time) between 8 and 20),
bad_days as (select [...] as time, sum(bad_wather) bad from weather [...],
checked as (select time, bad, lag(bad,1) over (order by time) bad1, [...] bad2 from bad_days)
select [...] as day_of_a_week, count(*) from checked where bad=0 and bad1>0 and bad2>0 [...]
").show(100,false)
```


Running Spark in Jupyter Notebook

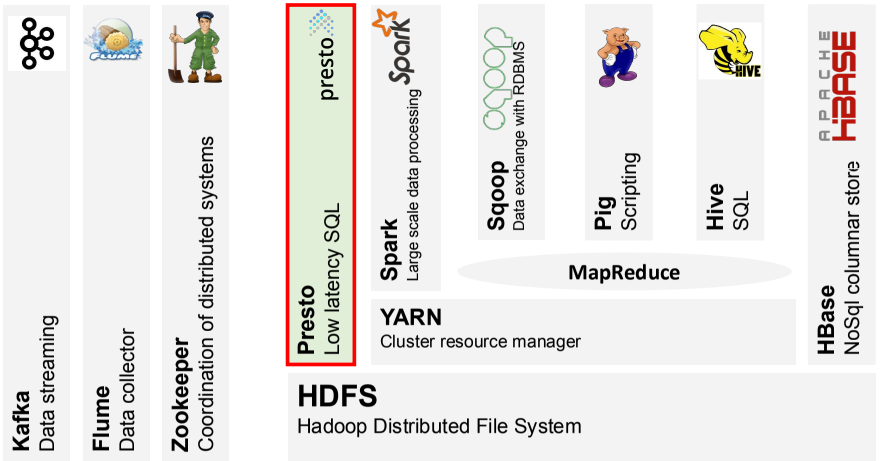
- Service for Web based ANalysis (SWAN) platform for interactive data analysis in the cloud developed @ CERN
- SWAN Platform: https://swan.web.cern.ch/
- Exercise to run on the workshop, Jupyter Notebook: http://cern.ch/go/X6Kj





Analytics platform outlook with HDFS, Spark and Jupyter

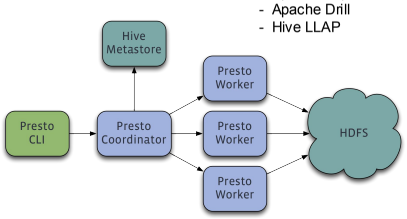
Big Data ecosystem

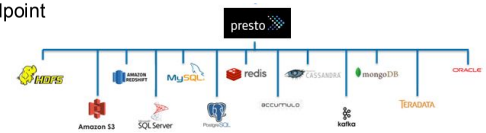


Presto - Massively Parallel Processing (MPP)

- **MPP SQL (on-anything) query engine for multiple datastores/databases** initiated by Facebook
- **Characteristics**:
  - **Low latency SQL queries** (query start up time **<100ms**)
  - Typically much **faster than Spark and MapReduce**
    - Executing daemons/workers are up all the time
    - **Platform agnostic, can run anywhere**
      - doesn't use Yarn
  - Typically **run on top of the Hadoop** cluster
- **Main benefits:**
  - Offers easy-to-use SQL (no other integration/code required),
  - Multiple connectors to data storages with one endpoint
  - Connectors are pluggable (ad-hoc adding)
  - Low latency thanks to:
    - Cost-Based Query Optimizer
    - Leveraging data locality in Hadoop

Similar frameworks:
- Apache Impala
- Apache Drill
- Hive LLAP

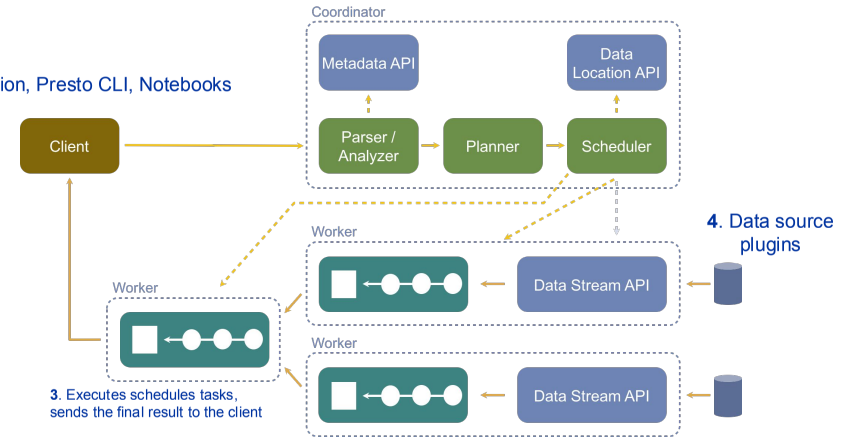




Presto Architecture

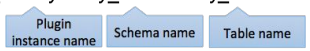
2. Receives a query from the client, analyzes, parses, plans, and schedules to the workers

1. Application, Presto CLI, Notebooks

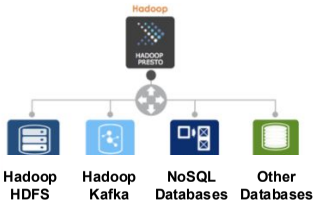


Presto for Hadoop in practice

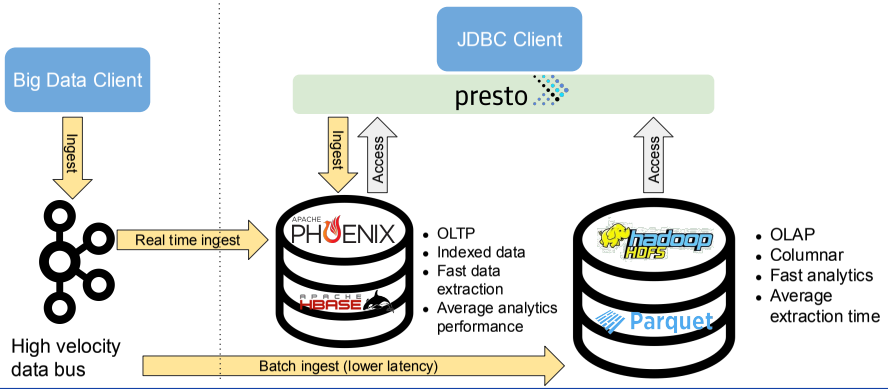
- **Dedicated connector for HDFS**
  - **Only** the data mapped **via Hive Metastore tables** can be accessed from HDFS
  - Existing HDFS folders can be easily mapped to Hive tables (if schema is coherent)
- **Each connector can have multiple instances (called catalogs)**
  - Multiple hives (Hadoop clusters) can be accessed simultaneously
  - *select * from hive_hadalytic.my_schema.my_table*



- **Interfaces:**
  - Presto shell (CLI)
  - JDBC/ODBC for binding with applications
  - Web: http://coordinator-addr:8080/ui/



Big Data scale-out database example with Presto



Presto SQL - weather forecast example

Actual query to compute sunny days after two rainy days in Geneva


```
[...] // Cleaning data

weather as (select time, case when weather in ('',' ') then 0 else 1 end bad_weather
from interesting_data where extract (hour from time) between 8 and 20),

bad_days as(select date_trunc('day',time) as time, sum(bad_weather) bad from weather [...]),

checked as (select time,bad,lag(bad,1) over (order by time) bad1, [...] bad2 from bad_days),

select date_format(time,'%W') as day_name, count(*) from checked
where bad=0 and bad1>0 and bad2>0 group by [...];
```


Table of contents

1. Brief introduction to Big Data and Hadoop ecosystem.
2. Distributed Data processing on Hadoop:
   a. MapReduce
   b. Spark SQL
   c. Presto
3. **Comparison of the processing frameworks.**
4. An example: Atlas EventIndex project.

Comparison of the 3 frameworks

- **MapReduce**
  - Requires complex coding of jobs - **time consuming**,
  - Intended mainly for **batch processing**
- **Spark SQL**
  - Covers most of the use cases (batch, long running ETLs)
  - Only one native connector to the Hive Metastore
  - The data from other sources can be queried only by writing some spark code and using 3rd party connectors as jars
- **Presto**
  - For interactive data access (low latency queries)
  - Cluster starts on-demand
  - Declared resources that are available all the time
  - Used for:
    - Generation of reports from big datasets
    - Complex analytics with multiple data sources
    - Querying: OLAP (HDFS/Parquet) and OLTP (HBase+Phoenix) systems

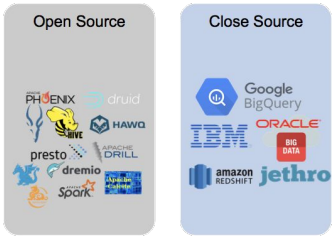


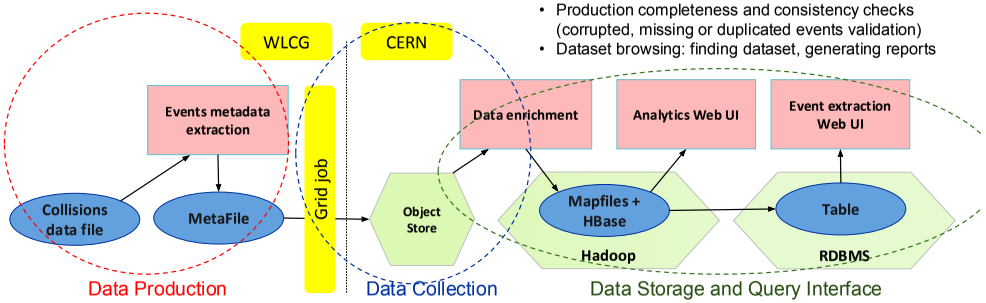
Table of contents

1. Brief introduction to Big Data and Hadoop ecosystem.
2. Distributed Data processing on Hadoop:
   a. MapReduce
   b. Spark SQL
   c. Presto
3. Comparison of the processing frameworks.
4. **An example: Atlas EventIndex project.**

The Atlas EventIndex

- **Catalogue of all collisions in the ATLAS detector**
  - Over 185 billion of records, 200TBs> of data
  - Current ingestion rates: 5kHz, 60TB/year
  - One record has size of ~1.5kB
  - Each indexed event is stored in a MapFile

- **EventIndex information**
  - Event identifiers:
    - Run and event number
    - Trigger Stream
    - Luminosity block
    - Bunch Crossing ID
- **Main use-cases**
  - Event picking
  - Count or select events based on trigger decisions
  - Production completeness and consistency checks (corrupted, missing or duplicated events validation)
  - Dataset browsing: finding dataset, generating reports



Instruction to execute exercises (self-guided)

- To access materials and documentation (available for everyone):
  - **$** *git clone https://gitlab.cern.ch/db/BigDataTraining-iCSC2020.git*
- Steps to run exercises on the CERN machines (requires CERN account):
  - Access CERN client machines (with configuration and hadoop binaries)
    - **$** *ssh it-hadoop-client.cern.ch # ithdp-client0[1-6].cern.ch # Requires connection to the CERN network*
    - More details in Hadoop guide: http://hadoop-user-guide.web.cern.ch/hadoop-user-guide/getstart/client_edge_machine.html#connecting
  - Set the environment (to point to the cluster configuration in order to interact with the CERN cluster):
    - Use either Analytix or Hadoop QA cluster depending on the exercise
    - **$** *source hadoop-setconf.sh analytix # or hadoop-qa*
- Execute jupyter notebooks using SWAN service - the first example: http://cern.ch/go/X6Kj
  - Check how to connect to the cluster with SWAN: http://spark-user-guide.web.cern.ch/spark-user-guide/spark-yarn/inter_user_guide.html
- The basic exercises to follow in the order: HDFS, MapReduce, Spark and YARN
- More advanced exercises (require executing first the basic ones): HBase, Parquet, Phoenix, Hive (metastore)

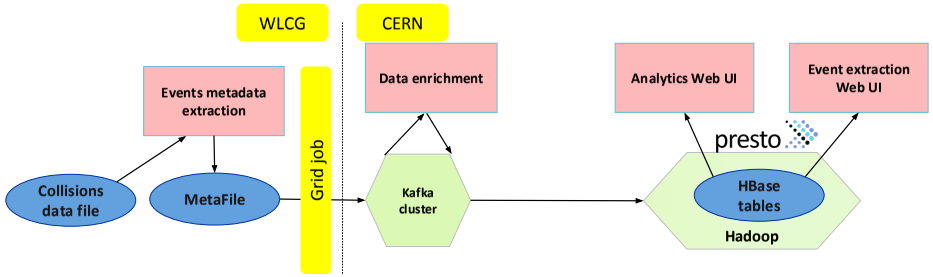
References

- https://blog.cloudera.com/big-data-processing-engines-which-one-do-i-use-part-1/ - comparison of Big Data Processing Engines (including SQL processing for OLAP & OLTP)
- phoenix.apache.org
- https://prestodb.io/blog/2019/08/05/presto-unlimited-mpp-database-at-scale
- A study of data representation in Hadoop to optimize data 2 storage and search performance for the ATLAS EventIndex, ref. http://cds.cern.ch/record/2244442/files/ATL-SOFT-PROC-2017-043.pdf
- A prototype for the evolution of ATLAS EventIndex based on Apache Kudu storage, ref. https://www.epj-conferences.org/articles/epjconf/pdf/2019/19/epjconf_chep2018_04057.pdf
- The ATLAS EventIndex: Full chain deployment and first operation, https://cds.cern.ch/record/1711821/files/ATL-SOFT-SLIDE-2014-360.pdf
- The ATLAS EventIndex for LHC Run 3, CHEP 2019 https://indico.cern.ch/event/868327/contributions/3660042/attachments/1975427/3287701/Barberis-EI3-CHEP2019v3.pdf
- Introduction to Presto, CERN, Hadoop and Spark User Forum 12.2019 https://indico.cern.ch/event/869037/contributions/3663775/attachments/1960650/3258410/Introduction_to_Presto.pdf

Thank you for your attention!

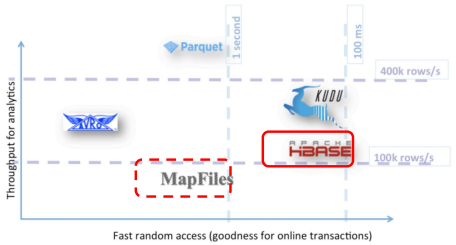
The Atlas EventIndex - new architecture proposal

- **Proposed changes:**
  - Replacing RDBMS with HBase/Phoenix and **Presto layer for SQL queries**
  - Replacing MapFiles with HBase data storing
  - In the future could be also Object Store replacement with Apache Kafka cluster



The Atlas EventIndex - performance comparison

- **Data ingestion speed** improved by rate of 2-10x.
- **Storage efficiency** improved by factor of 10
  - by using HBase + snappy compression on the data
- **Random data access** using HBase
  - typical random data lookup speed is below 500 ms
  - for the MapFile-based solution was around 4s
- **Data analytics** - fast and scalable with rate of 300k records per CPU core (300kHz)
- **Updates are possible** and not only appends
- Combining with Phoenix/Presto allows **querying data from multiple data sources with SQL**
- **Random lookup test is suboptimal for HBase** as a significant amount of time is spent to set up a query before it really gets executed ~200ms
- **Salting improves parallelism** by distributing data (regions) between different servers (regionservers)



The Atlas EventIndex - some queries and data structure

```
> show tables from phoenix_hadoop3.aei;
 datasets
 events
 sdatasets
 sevents

> use phoenix_hadoop3.aei;
> describe sdatasets;

# Typical AEI queries to find GUID of a file in Castor (with the event information)
> select * from datasets where runnumber=280753;

# Find dspid for the run # dspid = < project, runnumber, streamname,
processingStep, version >
# Example: dspid = <data15_13TeV, 00281385, physics_Main, deriv,
r9264_p3083_p3213>

> select * from events where dspid in (283,170) and eventnumber=4317812;
# To find the reference to the file and more metadata

# The worst scenario (scanning the whole dataset)
> select count(*) from events;
```

Table 1. *Datasets* table schema

column name	column type	encoding	compression	primary key
runnumber	int	BIT_SHUFFLE	LZ4	X
project	string	DICT_ENCODING	SNAPPY	X
streamname	string	DICT_ENCODING	SNAPPY	X
prodstep	string	DICT_ENCODING	SNAPPY	X
datatype	string	DICT_ENCODING	SNAPPY	X
version	string	DICT_ENCODING	SNAPPY	X
dspid	int	BIT_SHUFFLE	LZ4	
rgid	int	BIT_SHUFFLE	LZ4	
insert_start	timestamp	BIT_SHUFFLE	LZ4	
insert_end	timestamp	BIT_SHUFFLE	LZ4	
backup_start	timestamp	BIT_SHUFFLE	LZ4	
backup_end	timestamp	BIT_SHUFFLE	LZ4	
validated	timestamp	BIT_SHUFFLE	LZ4	
count_events	bigint	BIT_SHUFFLE	LZ4	
uniq_dspl_events	bigint	BIT_SHUFFLE	LZ4	
num_duplicates	bigint	BIT_SHUFFLE	LZ4	
trigger_counted	int	BIT_SHUFFLE	LZ4	
ds_overlaps	int	BIT_SHUFFLE	LZ4	
amt_count	bigint	BIT_SHUFFLE	LZ4	
amt_raw_count	bigint	BIT_SHUFFLE	LZ4	
amt_date	timestamp	BIT_SHUFFLE	LZ4	
amt_upd_date	timestamp	BIT_SHUFFLE	LZ4	
amt_state	string	DICT_ENCODING	SNAPPY	
inconsistainer	int	BIT_SHUFFLE	LZ4	
state	string	DICT_ENCODING	SNAPPY	
smk	int	BIT_SHUFFLE	LZ4	

Table 2. *Events* table schema

column name	column type	encoding	compression	primary key
dspid	int	BIT_SHUFFLE	LZ4	X
eventnumber	bigint	BIT_SHUFFLE	LZ4	X
rgid	int	BIT_SHUFFLE	LZ4	X
hlpsk	int	BIT_SHUFFLE	LZ4	
l1psk	int	BIT_SHUFFLE	LZ4	
lumiblocknr	int	BIT_SHUFFLE	LZ4	
bunchid	int	BIT_SHUFFLE	LZ4	
eventtime	int	BIT_SHUFFLE	LZ4	
eventtimens	int	BIT_SHUFFLE	LZ4	
lvl1id	bigint	BIT_SHUFFLE	LZ4	
l1trigmask	string	DICT_ENCODING	SNAPPY	
l1trigchainsnaw	string	DICT_ENCODING	SNAPPY	
l1trigchainswap	string	DICT_ENCODING	SNAPPY	
l1trigchainsPtp	string	DICT_ENCODING	SNAPPY	
efhtgmask	string	DICT_ENCODING	SNAPPY	
efhtgchainsph	string	DICT_ENCODING	SNAPPY	
efhtgchainspt	string	DICT_ENCODING	SNAPPY	
efhtgchainsrs	string	DICT_ENCODING	SNAPPY	
dbrow	string	DICT_ENCODING	SNAPPY	
tkraw	string	DICT_ENCODING	SNAPPY	
dhead	string	DICT_ENCODING	SNAPPY	
tkead	string	DICT_ENCODING	SNAPPY	
dhaed	string	DICT_ENCODING	SNAPPY	
tkaod	string	DICT_ENCODING	SNAPPY	
db	string	DICT_ENCODING	SNAPPY	
tk	string	DICT_ENCODING	SNAPPY	

Programming Paradigms

Kilian Lieret[1,2]

Mentors:
Sebastien Ponce[3], Enric Tejedor[3]

[1]Ludwig-Maximilian University
[2]Excellence Cluster Origins
[3]CERN

28 September 2020

Overview

- Lecture 1: Programming Paradigms (PPs): Monday 9:15 – 10:15
- Lecture 2: Design Patterns (DPs): Tuesday 14:30 – 15:30
- Exercises: Wednesday 09:00 – 11:00

All material at: `github.com/klieret/icsc-paradigms-and-patterns`

The goal of this course

- This course does not try to make you a better programmer
- But it **does** convey basic concepts and vocabulary to make your design decisions more **consciously**
- Thinking while coding + reflecting your decisions after coding ⟶ Experience ⟶ Great code!

Programming Paradigms

What **is** a programming paradigm?

- A classification of programming languages based on their features (but most popular languages support multiple paradigms)
- A programming style or way programming/thinking
- Example: Object Oriented Programming (thinking in terms of objects which contain data and code)
- Many common languages support (to some extent) multiple paradigms (C++, python, ...)

Why should I care?

- Discover new ways of thinking ⟶ challenge your current believes about how to code
- Choose the right paradigm for the right problem or pick the best of many worlds

Programming Paradigms

Some problems

- Too formal definitions can be hard to grasp and sometimes impractical, too loose definitions can be meaningless
- Comparing different paradigms requires experience and knowledge in both (if all you [know] is a hammer, everything looks like a nail)
- A perfect programmer might write great software using any PP

My personal approach

- Rather than asking "How to define paradigm X?", ask "How would I approach my problems in X?".
- Try out "academic languages" that enforce a certain paradigm ⟶ How does it *feel* to program in X
- Get back to your daily programming and rethink your design decisions

Good code: Objectives

Key objectives

- Testability: Make it easy to ensure the software is working correctly
- Maintainability: Make it easy to keep the software working (debugging, readability, ...)
- Extendibility: Make it easy to add new functionality
- Flexibility: Make it easy to adapt to new requirements
- Reusability: Make it easy to reuse code in other projects

$\longrightarrow$ How do I achieve all this?

Modularity

Perhaps the most important principle of good software

Split up code into parts, e.g. functions, classes, modules, packages, ...

You have done **well** if the parts are
- independent of each other
- have clear responsibilities

You have done **badly** if the parts
- are very dependent on each other (changes in one part require changes in many others)

This has benefits for almost all of your goals:
- Easier and more complete testability by using unit tests, better debugging
- Confidence from unit tests allows for better maintainability and flexibility
- Allowing to split responsibilities for different "modules" enhances collaboration and thereby maintainability
- Code reusability (obvious)

Modularity

Perhaps the most important principle of good software

A related principle: Isolate what changes!
- Which parts of your code will likely have to change in the future?
  $\longrightarrow$ These parts should be isolated (you should be able to change them in one place, without having to change anything else)
- This also leads to the concept of a separation of
  - interface (used by other "modules", stays untouched) and
  - implementation (only used by the module itself, can change easily)

Complex vs Complicated

From the Zen of python:

Simple is better than complex.
Complex is better than complicated.

- The more *complicated* something is, the harder it is to understand
- The more *complex* something is, the more parts it has

- Complicated problems might not have simple solutions
- But it is often still possible to modularize to have several simple components

- For example, using classes and objects will make your code more *complex*, but still easier to understand

OOP: Idea

- Before OOP: Two **separate** entities: *data* and *functions* (logic)
- Inspiration: In the real world, objects have a "state" (data) and "behaviors" (functions)

OOP

- **Think** in terms of *objects* that contain data and offer *methods* (functions that operate on objects) $\longrightarrow$ Data and functions form a unit
- **Focus** on object structure rather than manipulation logic
- **Organize** your code in *classes* (blueprints for objects): Every object is *instance* of its class

A basic class in python

```python
class Rectangle:
    def __init__(self, width, height):  # <-- constructor
        # 'self' represents the instance of the class
        self.width = width               # <-- attribute = internal variable
        self.height = height

    def calculate_area(self):            # <-- method (function of class)
        return self.width * self.height


r1 = Rectangle(1, 2)          # <-- object (instance of the class)
print(r1.calculate_area())    # <-- call method of object
print(r1.width)               # <-- get attribute of object
r1.width = 5                  # <-- set attribute of object
```


Encapsulation and data hiding

- Do not expose object internals that may change in the future $\longrightarrow$ Make certain attributes and methods private (*data hiding*)
- Rephrased: Separate interface (won't be touched because it's used by others) from implementation (might change)
- In some languages this is "enforced" (e.g. using the `private` keyword), in others it is denoted by naming conventions (e.g. leading underscore)

Subclasses and Inheritance

Subclasses are specializations of a class

- inherit attributes/methods of their superclass
- can introduce new attributes/methods
- can override methods of superclass

```python
class Person:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f"Hello, I'm {self.name}")


class Child(Person):
    def __init__(self, name, school):
        super().__init__(name)
        self.school = school

    def learn(self):
        print(f"I'm learning a lot at {self.school}")


c1 = Child("john", "iCSC20")
c1.greet()
c1.learn()
```


Abstract methods

- An *abstract method* is a method that has to be implemented by a subclass
- An *abstract class* (*abstract type*) is a class that cannot be instantiated directly but it might have *concrete subclasses* that can
- Use abstract classes to enforce interfaces for the concrete classes

```python
from abc import ABC, abstractmethod


class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        pass

    @abstractmethod
    def draw(self):
        pass


class Rectangle(Shape):
    def __init__(self, ...):
        ...

    def calculate_area(self):
        # concrete implementation here
```

Strenghts and Weaknesses

Strengths

- Easy to read and understand if done well (very natural way of thinking if classes model real world objects)
- Natural way to structure large projects (e.g. taking classes as components)
- Very wide spread way of thinking
- Especially applicable to problems that center around data and bookkeeping with logic that is strongly tied to the data

Weaknesses

- Wrong abstractions can lead to less code reusability
- Lasagna code: Too many layers of classes can be hard to understand
- Can be hard to parallelize if many entangled and interdependent classes with shared mutable states are involved ($\longrightarrow$ if required, should be design requirement from the start; *parallel patterns* address some difficulties)

Functional programming

Functional programming
- expresses its computations in the style of mathematical functions
- emphasizes
  - expressions ("is" something: a series of identifiers, literals and operators that reduces to a value)

  over
  - statements ("does" something, e.g. stores value, etc.)

  $\longrightarrow$ declarative nature
- Data is immutable (instead of changing properties, I need to create copies with the changed property)
- Avoids side effects (expressions should not change or depend on any external state)

Examples

Languages made for FP (picture book examples):
- `Lisp` and derivatives: `Common Lisp`, `Clojure`, …
- `Haskell`
- `OCaml`
- `F#`
- `Wolfram Language` (Mathematica etc.)
- …

With emphasis on FP:
- `JavaScript`
- `R`
- …

Not designed for, but offering strong support for FP:
- `C++` (from `C++11` on)
- `Perl`
- …

Pure functions

A function is called pure if

1. Same arguments $\implies$ same return value ($x = y \implies f(x) = f(y)$)
2. The evaluation has no side effects (no change in non-local variables, ...)

Which of the following functions are pure?

```
def f1(x):
    return x**2


def f2(x):
    print(x)
    return x**2


global y = 0


def f3(x):
    y += 1
    return x + y
```

```
def f4():
    return int(input()) + 1


def f5(lst: List):
    lst[0] = 3
    return lst
```

Answer: `f1` is pure; `f1`, `f3`, `f5` violate rule 2; `f4` violates rule 1.

Non strict evaluation

- Some functional programming languages use *non-strict evaluation*: The arguments of a function are *only* evaluated once the function is called.

  Example: `print(sqrt(sin(a**2)))`

  In a strict language (e.g. `Python`, `C++`), we evaluate inside out:

  $$a \longmapsto a^2 \longmapsto \sin a^2 \longmapsto \sqrt{\sin a^2}$$

  In a non-strict language, the evaluation of the inner part is **deferred**, until it is actually needed.

- But `Python` actually has something similar in the concept of `generators`:

```
%time a = range(int(1e8))
>>> Wall time: 7.63 µs

%time b = list(a)
>>> Wall time: 2.33 s
```

- This allows for infinite data structures (which can be more practical than it sounds)

Memoization

- Non strict evaluation together with *sharing* (avoid repeated evaluation of the same expression) is called *lazy evaluation*
- Generally, functional programming can get cheap performance boosts by very simple *memoization*: Storing the results of expensive **pure** function calls in a cache

```
import time
from functools import lru_cache


@lru_cache()
def expensive(x):
    time.sleep(1)
    return x+42


%time expensive(2)
>>> Wall time: 1 s

% time expensive(2)
>>> Wall time: 6.2 µs
```


Higher order functions

A *higher order function* does one of the following:

- returns a function
- takes a function as an argument

Opposite: *first-order function*.

Mathematical examples (usually called *operators* or *functionals*): differential operator, integration, ...

Higher level functions are the FP answer to template methods in OOP ("configuring" object behavior by overriding methods in subclasses).

Classic example of a higher order function: `map` (applies function to all elements in list):

```
def map(function, iterator):
    """ Our own version of map (returns a list rather than a generator) """
    return [function(item) for item in iterator]


map(lambda x: x**2, [1, 2, 3])
>>> [1, 4, 9]
```

Higher order functions II

A function that also returns a function:

```
def get_map_function(function):
    """ Takes a function f and returns the function map(f, *) """
    def _map_function(iterator):
        return map(function, iterator)

    return _map_function


mf1 = get_map_function(lambda x: x**2)
mf2 = get_map_function(lambda x: x+1)

mf2(mf1([1, 2, 3]))
>>> [2, 5, 10]
```


Type systems

Types:

- In OOP, type and class are often used interchangeably (e.g. `"abc"` is of type *string* = is an instance of the `str` class)
- In FP we talk about *type*s
- Complex types can be built from built in types (e.g. `List[Tuple[str, int]]`, we can also use `structs`)
- In many languages, types of variables, arguments, etc. have to be declared (e.g. `def len(List[float]) -> int`)
- Real FP languages usually have very powerful type systems

Polymorphism

In FP, the type system allows to bring back some OOP thinking but is more flexible. Usually you can do some of the following:

Single/multiple dispatch/ad hoc polymorphism:

- Can overload function definitions (e.g. define `def print(i: int)` differently from `def print(string: str)`)
- The right function is resolved based on the type at compile- or runtime

Parametric polymorphism:

- Parameterize types in function signatures (e.g. `def first(List[a]) -> a`; `a` represents an arbitrary type)

Type classes:

- Define a "type" by what functions it has to support (e.g. define `Duck` as anything that allows me to call the `quack` function on it)
- Similar to a class with only abstract methods (=*interface*) and no encapsulated data

Looping in functional programming

Let's consider a function that calculates $\sum_{i=0}^{N} i^2$:

```
def sum_squares_to(n):
    result = 0
    for i in range(n+1):
        result += i^2
    return result
```

This is a function, but does not follow the FP paradigm:

- More statements (assignments, loops, ...) than expressions
- The for loop segment is not free of side effects (value of `result` changes)
- Repeated reassignments of `result` are frowned upon (or impossible)

How to change this? $\longrightarrow$ Use recursion

```
def sum_squares_to(n):
    return 0 if n == 0 else n^2 + sum_squares_to(n-1)
```

Looping in functional programming

The previous example is called a head recursion (recursion before computation); using a tail recursion (recursion after computation) is preferable due to better compiler optimization:

```
def sum_squares_to(n, partial_sum=0):
    return partial_sum if n == 0 else sum_squares_to(n-1, partial_sum + n^2)
```

Another FP way is to use the higher level functions `map` and `reduce` together with anonymous functions (lambda):

```
from functools import map, reduce


def sum_squares_to(n):
    return reduce(
        lambda x, y: x+y,
        map(lambda x: x**2, range(n+1))
    )
```

This also opens the door for concurrency ($\longrightarrow$ parallel versions of `map` and `reduce`)

Strengths and Weaknesses

Strengths

- Proving things mathematically (referential transparency, ...)
- Testability (no object initializations and complex dependencies, pure functions)
- Easy debugging (no hidden states)
- Can be very short and concise $\longrightarrow$ easy to verify
- Sophisticated logical abstractions (using high level functions) $\longrightarrow$ modularity, code reuse
- Easy parallelization (no (shared) mutable states)

Strengths and Weaknesses

Weaknesses

- Structuring code in terms of objects can feel more intuitive if logic (methods) are strongly tied to data
- Imperative algorithms might be easier to read and feel more natural than declarative notation
- FP might have a steeper learning curve (e.g. recursions instead of loops, ...)
- Performance issues: Immutable data types and recursion can lead to performance problems (speed and RAM), whereas many mutable data structures are very performant on modern hardware
- Pure FP has still only a small user base outside of academia, but FP support more and more wide spread in common languages

Object oriented vs functional programming

Some key aspects to keep in mind:

- FP $\neq$ OOP – classes
- FP is **not** the opposite of OOP: Both paradigms take opposite stances in several aspects: declarative vs imperative, mutable vs immutable, ... $\Longrightarrow$ Not everything can be classified into one of these categories
- Rather: Two different ways to think and to approach problems $\longrightarrow$ see caveats at the beginning

In a multi-paradigm language, you can use the best of both worlds!

- OOP has its classical use cases where there is strong coupling between data and methods and the bookkeeping is in the focus (especially of "real-world" objects)
- FP instead focuses on algorithms and *doing* things
- Some people advocate "OOP in the large, FP in the small" (using OOP as the high level interface, using FP techniques for implementing the logic)

For example:

- Many complicated class structures implementing manipulations can be made more flexible with a system of high level functions, anonymous functions etc.

Declarative vs imperative programming

Declarative programming:
- Program describes logic rather than control flow
- Program describes "what" rather than "how"
- Aims for correspondence with mathematical logic
- FP is usually considered a subcategory

Opposite: imperative programming:
- Algorithms as a sequence of steps
- Often used synonymously: procedural programming (emphasizing the concept of using procedure calls (functions) to structure the program in a modular fashion)
- OOP is usually considered a subcategory

Examples

Picture book examples:
- SQL (Structured Query Language – language to interact with databases ⟶ Emil Kleszcz):

```
SELECT * FROM Customers WHERE Country='Mexico';
```

- Markup languages, like HTML, CSS (Cascading Style Sheets – language to describe styling of e.g. HTML pages), ...

```
<h1 style="color:blue;">This is a Blue Heading</h1>
```

- Functional programming languages like Haskell (even though they allow some "encapsulated" imperative parts)
- ...

Powerful backends I

Idea:
- Split up your code into application/analysis specific code (describing the problem) and a backend/library (implementing solution strategies)
- The application specific code starts to *feel* very declarative
- The backend can use different strategies depending on the nature/scale of the problem

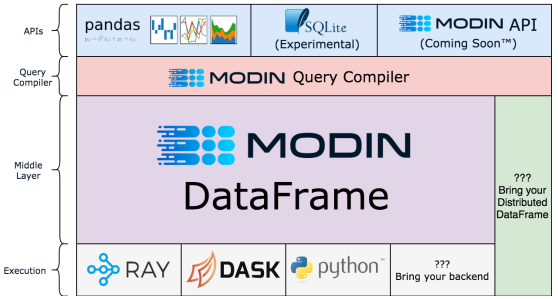
Powerful backends II

Example:

```python
# "Chi2 distance" using plain python
def chi2(data, theory, error):
    err_sum = 0
    for i in range(len(data)):
        if data[i] == theory[i] and error[i] == 0:
            continue
        err_sum += (data[i] - theory[i])**2 / (error[i]**2)
    return err_sum


# Using DataFrames: Table contains columns experiment, theory, error

# ROOT RDataFrame example:
chi2 = ROOT.ROOT.RDataFrame(...)  # initialize
        .Filter("!(data==theory & error==0.)")  # filter rows
        .Define("sqd", "pow(data-theory, 2) / pow(error, 2)")  # new col
        .Sum("sqd").GetValue()  # sum it up

# Pandas example:
chi2 = pd.DataFrame(...)  # initialize
_df = df.query("~(data==theory & error==0)")  # filter
chi2 = (_df["data"] - _df["theory"]).pow(2) / _df["error"].pow(2)).sum()
```


Powerful backends III

- We might want even more of our backend, e.g. delayed or distributed execution
- pandas can also be viewed as a "declarative language" describing the problem $\longrightarrow$ have a more sophisticated backend handle all operations $\longrightarrow$ `modin pandas`



Powerful backends IV

Belle II steering file:

```
path = create_path()

# Load data
inputMdstList("default", "/path/to/input/file", path=path)

# Get final state particles

# Fill 'pi+:loose' particle list with all particles that have pion ID > 0.01:
fillParticleList("pi+:loose", "piid > 0.01", path=path)
# Fill 'mu+:loose' particle list with all particles that have muon ID > 0.01:
fillParticleList("mu+:loose", "piid > 0.01", path=path)

# Reconstruct decay
# Fill 'K_S0:pipi' particle list with combinations of our pions and muons
reconstructDecay(
    "K_S0:pipi -> pi+:loose pi-:loose", "0.4 < M < 0.6", path=path
)
```


Powerful backends V

Many more high level tools available:

- LINQtoROOT: Uses C# with LINQ (SQL like) queries to describe problem

```
events
  .Select(e => e.Data.eventWeight)
  .FuturePlot("event_weights", "Sample EventWeights",100, 0.0, 1000.0)
  .Save(hdir);
```

- The FAST HEP toolkit: Uses yaml config files to describe problem; using pandas, numpy, etc. in the backend

```
stages:
 - BasicVars: Define
 - DiMuons: cms_hep_tutorial.DiObjectMass
 - NumberMuons: fast_carpenter.BinnedDataframe
 - EventSelection: CutFlow
 - DiMuonMass: BinnedDataframe
```

- Many more…

Other paradigms

- Logic programming (LP) (subset of declarative programming): Automatic reasoning by applying inference rules
  - LP languages: `Prolog`, `Datalog`
  - LP can be made available with libraries, e.g. for `Python`: `Pyke` (inspired by `prolog`), `pyDatalog` (inspired by `Datalog`)
  - Example:

```
% X, Y are siblings if they share a parent
sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).

% Father, mother implies parent
parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).

% Introduce some people
father_child(tom, sally).
father_child(tom, erica).

% Ask:
?- sibling(sally, erica).
Yes
```

- Symbolic programming
- Differentiable programming

Outlook

Next lecture: Software design patterns
- Focus on OOP
- Introduce some “golden rules” of OOP
- Patterns: Reusable solutions to common problems

Software Design Patterns

Kilian Lieret[1,2]

Mentors:
Sebastien Ponce[3], Enric Tejedor[3]

[1]Ludwig-Maximilian University
[2]Excellence Cluster Origins
[3]CERN

29 September 2020

Slides + material available at
github.com/klieret/icsc-paradigms-and-patterns

Repetition: Object Oriented Programming

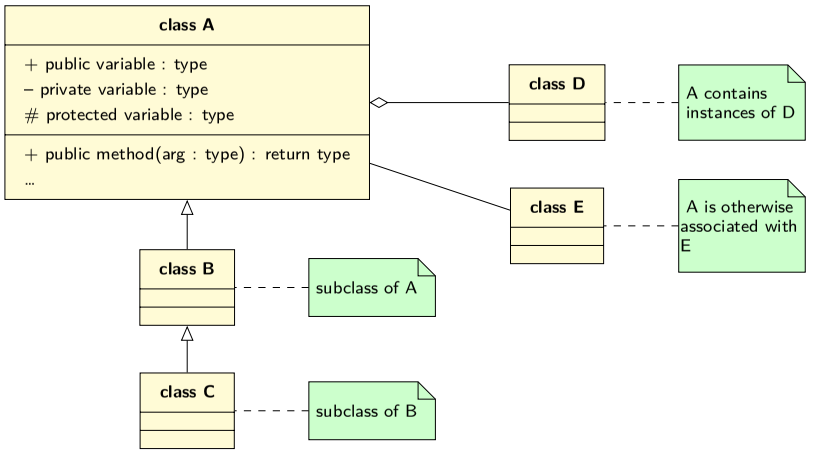
- Inheritance: Subclasses inherit all (public and protected) attributes and methods of the base class
- Methods and attributes can be public (anyone has access), private (only the class itself has access) or protected (only the class and its subclasses have access)
- Abstract methods of an abstract class are methods that have to be implemented by a subclass (concrete class)

New: class methods: Methods of the class, rather than its instances

```python
class Class:
    def method(self):
        # needs to be called from INSTANCE and can access instance attributes
    @classmethod
    def classmethod(cls):
        # no access to instance attributes

# This won't work:
Class.method()  # <-- needs an instance, e.g. Class(...).method()

# But this does:
Class.classmethod()
```


Class diagrams I

UML (Unified Markup Language) class diagrams visualize classes and the relationships between them. We will use the following subset of notations:



The SOLID rules of OOP: Single responsibility principle

SOLID

Commonly (mis-)quoted as:

A class should only have one responsibility.

More accurate:

A class should only have one reason to change.

Better still:

Gather together the things that change for the same reasons.
Separate those things that change for different reasons.

So this actually proposes a balance!

- Avoid classes that do too much ("god class")
- But also avoid having changes always affect many classes ("shotgun surgery")

The SOLID rules of OOP: Open Closed Principle

SOLID

You should be able to extend the behavior of a system without having to modify that system.

- Writing a library, modifying functionality means that all users have to be informed (not backwards compatible) $\longrightarrow$ Avoid!
- In your own code: Modifying one functionality (also by overriding methods of the super class, etc.) poses the danger of breaking other parts (though tests can help with that)
- Extending code by providing additional methods, attributes, etc. does not have this danger $\longrightarrow$ preferred!
- Requires thinking ahead: What parts have to be flexible, what remains constant?
- Again a balance is required:
  - Be too generic (avoid modifications) and your code won't do anything
  - Be too concrete and you will need to modify (and potentially break things) often

The SOLID rules of OOP: Liskov Substitution Principle

SOLID

If S is a subtype (subclass) of T, then objects of type T can be replaced with objects of type S without breaking anything

This can be expanded to a series of properties that should be fulfilled:

- Signature of methods of the subclass:
  - Required type of arguments should be supertype (*contravariance*)
    Violation: Supermethod accepts any `Rectangle`, submethod only `Square`
  - Return type of method should be a subtype (*covariance*)
    Violation: Supermethod returns `Square`, submethod returns `Rectangle`
- Behavior:
  - Preconditions (requirements to be fulfilled before calling method) cannot be strengthened in the subtype
    Violation: Only in subclass `prepare()` must be called before `method()`
  - Postconditions (conditions fulfilled after calling a method) cannot be weakened by the subtype
  - Invariants (properties that stay the same) of supertype must be preserved in the subtype
  - History contraint: Subtypes cannot modify properties that are not modifyable in supertype
    Violation: `VariableRadiusCircle` as subtype of `FixedRadiusCircle`

The SOLID rules of OOP: Interface segregation principle (ISP)

SOLID

Clients should not be forced to depend on methods they do not use

- "Thin" interfaces offering a reasonably small number of methods with *high cohesion* (serve similar purposes; belong logically together) are **preferred** over "fat" interfaces offering a large number of methods with low cohesion
- Sometimes we should therefore split up (segregate) fat interfaces into thinner *role interfaces*
- This leads to a more decoupled system that is easier to maintain
- Example: Even if all data is contained in one (e.g. SQL) database, the ISP asks to write *different* interfaces to do different things, e.g. have a `CustomerDb`, `OrderDb`, `StoreDb`, ...

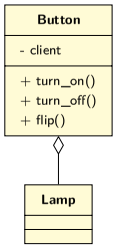
The SOLID rules of OOP: Dependency Inversion Principle

SOLID

This is about decoupling different classes and modules:

1. *High-level modules should not depend on low-level modules. Both should depend on abstractions (interfaces).*

Let's consider a very simple example: A button controlling a lamp. One way to implement this:



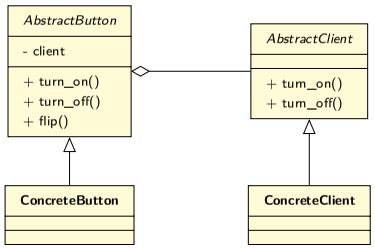
This violates the DIP, because `Button` (high-level) depends on `Lamp` (detail).

What if we have multiple consumers (`Motor`, `Lamp`, ...) and multiple types of buttons (swipe button, switch, push button, ...)? How can we force them to behave consistent? What methods does a consumer have to implement to work together with the button?

$\longrightarrow$ Enter abstractions (interfaces)

The SOLID rules of OOP: Dependency Inversion Principle

SOLID



Now it's clear which methods the concrete client has to implement. Both high level and low level modules only depend on abstractions.

This also fulfills the second part of the DIP:

2. *Abstractions should not depend on details. Details (i.e. concrete implementations) should depend on abstractions.*

Performance considerations

Some patterns will advocate:

- Classes that only act as interfaces and pass on calls to other (worker) classes
- Using separate classes to facilitate communication between classes
- Accessing attributes (only) through methods
- Prefer composition over inheritance

However, when writing performance critical (`C++`, ...) code, you should avoid unnecessary "detours":

- Avoid unncessary interfaces
- Consider inlining simple, often-called functions (e.g. getters and setters)
- Inheritance > composition > `if` statements

Modern compilers will try to apply some optimization techniques automatically (automatic inling, return value optimization, ...)

General rule: Profile before Optimizing

Patterns

Software design patters try to offer general and reusable solutions for commonly occuring problems in a given context.

Commonly categorized as:

- Creational patterns: How are instances of classes instantiated? (What if I have a class that can create instances in different ways?)
- Structural patterns: Concerned with relationships between classes. (How can classes form flexible larger structures?)
- Behavioral patterns: Concerned with algorithms and communication between classes. (How are responsibilities assigned between classes?)
- Parallel patterns: Parallel processing and OOP $\longrightarrow$ only mentioned briefly

Factory method

If there are multiple ways to instantiate objects of your class, use factory methods rather than adding too much logic to the default constructor.

Bad:

```python
class Uncertainty:
    def __init__(self, absolute_errors=None,  relative_error=None,
        data=None, config=None, ...):
        if config is not None:
            # load from config
        elif absolute_errors is not None:
            # add absolute errors
        elif relative_errors is not None and data is not None:
            # add relative errors
        ...


instance = Uncertainty(config="path/to/my/config")
```

Factory method

If there are multiple ways to instantiate objects of your class, use factory methods rather than adding too much logic to the default constructor.

Good:

```python
class Uncertainty:
    def __init__(self, absolute_errors):
        # construct from absolute errors

    @classmethod  # <-- doesn't need instance to be called (cf. first slide)
    def from_config(cls, config):
        # get absolute errors from config file
        return cls(absolute_errors)

    @classmethod
    from relative_errors(cls, data, relative_errors):
        return cls(data * relative_errors)


instance = Uncertainty.from_config("path/to/my/config")
```

Alternatively, you can also have subclasses that provide (implementations to) factory methods.

Builder Pattern

If you build a very complex class, try to instantiate (build) it in several steps.

Bad:

```python
class Data:
    def __init__(
        data: array,
        data_error: array
        mc_components: List[array],
        mc_errors: List[array],
        mc_float_normalization: List[bool],
        mc_color: List[string],
        ...
    )

    def fit(...)

    def plot(...)
```

You will probably consider different fits and plots; violates Single Responsibility Principle $\longrightarrow$ Rather have `Fit` and `Plot` classes

Builder Pattern

If you build a very complex class, try to instantiate (build) it in several steps.

Bad:

```python
class Data:
    def __init__(
        data: array,
        data_error: array
        mc_components: List[array],
        mc_errors: List[array],
        mc_float_normalization: List[bool],
        mc_color: List[string],
        ...
    )
```


Builder Pattern

If you build a very complex class, try to instantiate (build) it in several steps.

Better:

```python
class Data:
    def __init__(data: array, data_error: array)
        pass

    def add_mc_component(data, errors, floating=False, color="black", ...):
        pass


data = Data(...)
data.add_mc_component(...)
...
data.add_mc_component(...)
```

- What if we have multiple ways to build of the object?
- Do I want to have the `add_mc_component` method after I start using the data?

$\longrightarrow$ Have a separate `Data` and `Builder` hierarchy.

Builder Pattern

If you build a very complex class, try to instantiate (build) it in several steps.

Best:

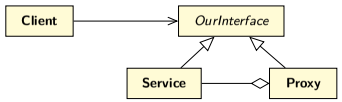
```
class Data:
    pass

class Builder:
    def __init__(...)

    def add_mc_component(...)

    def create(...) -> Data


builder = Builder(...)
builder.add_mc_component(...)
...
builder.add_mc_component(...)
data = builder.create()
```

And of course I could now create `AbstractData` and `AbstractBuilder` etc.

Proxy, Adapter, Facade

Three patterns that deal with interfaces:

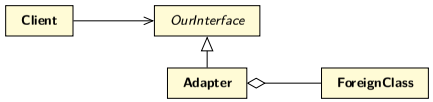
- Proxy: Given a *servant* class (doing the actual work), create a new *proxy* class with the same interface in order to inject code. The client can then use the `Proxy` instead of using the `Service` class directly.



Usage examples:
- Protection proxy: Enforce access rights (always check authorization before method call/attribute access; e.g. in web applications)
- Remote proxy: If the `Service` is located remotely, the proxy deals with transferring requests and results
- Extend the Service class with caching or logging
- ...

Adapter

- Facade: A class providing a simple interface for complicated operations that involve multiple servant classes
- Adapter: We have a 3rd party class `ForeignClass` whose interface is incompatible to interface `OurInterface` $\longrightarrow$ Create an *adapter* class as a wrapper



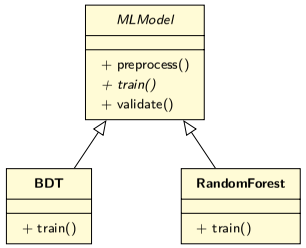
Usage case example: We want to switch between different machine learning models (strategy pattern $\longrightarrow$ later). Our models have a `train()` method, models from a foreign library have a `training()` method $\Longrightarrow$ create adapter(s) for library

Adapter

```python
class OurMLModel(ABC):
    """ Our interface """
    @abstractmethod
    def train(...):
        pass


class TheirMLModel(ABC):
    """ Their interface """
    @abstractmethod
    def training(...)  # <-- this method should be called train
        pass


class ModelAdapter(OurMLModel):  # <-- implements our interface
    def __init__(self, model: TheirMLModel):
        self._model = model   # <-- our adapter holds the foreign model

    def train(...):           # <-- and defines a different interface for it
        self._model.training(...)


# Their model with our interface:
model = ModelAdapter(TheirMLModel(...))
```


Template Method

- Use case: Several different classes that only contain minor differences in few places
- Suggestion: Put shared code in superclass, have subclasses implement or override specific methods



- Advantages: Simple and clean with little overhead
- Warnings:
  - If there are many differences between original classes, we need (to override) many methods ⟶ increasingly hard to read and mantain
  - If there are multiple "options" for every method and we want to realize them, the number of subclasses grows exponentially ⟶ *Strategy pattern*
  - If overriding default methods, the Liskov Substitution Principle can be easily violated

Template Method

Questionable:

```python
class MLModel():
    def load_data(...)
    def prepare_features(...)

    def train(...):
        if self.model == "BDT":
            # train BDT
        elif self.model == "RandomForest":
            # train random forest
        elif ...

    def validate(...)
    ...
```

- What if multiple methods depend on the model? ⟶ Need to keep track of more `if`s everywhere
- What if we want to add or remove a model? ⟶ Need to make changes in many places ⟶ ~~Open/Closed Principle~~, "divergent change"
- Depend on all implementations ⟶ ~~Dependency Inversion Principle~~

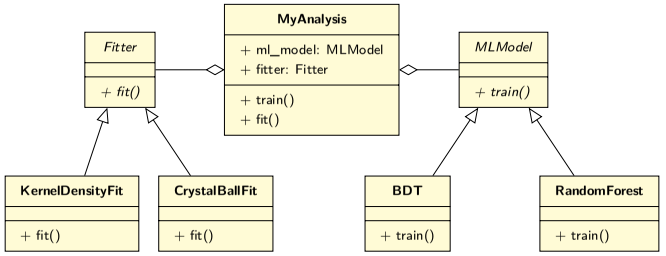
Template Method

Better:

```python
class MLModel(ABC):  # <-- abstract class
    def load_data(...)
    def prepare_features(...)

    @abstractmethod
    def train(...):
        pass

    def validate(...)
    ...


class BDT(MLModel):  # <-- concrete class
    def train(...):
        # Implementation


class RandomForest(MLModel):
    def train(...):
        # Implementation
```


Strategy

- Usage: You have a problem that can be solved with different algorithms (strategies)
- Suggestion: Create abstract class for algorithm and concrete subclasses with specific implementation; original class holds instance of algorithm class



Strategy

```python
class MyAnalysis():
    def __init__(ml_model: MLModel, fitter: Fitter)
        self.ml_model = ml_model
        self.fitter = fitter

    def fit(...):
        self.fitter.fit(...)

    def train(...):
        self.ml_model.train(...)

class MLModel(ABC):
    @abstractmethod
    def train(...)

class RandomForest(MLModel):
    def train(...):
        # Implementation


my_analysis = MyAnalysis(RandomForest(...), KernelDensityEstimator(...))
my_analysis.train(...)
my_analysis.fit(...)
```


Strategy

- Note: The main class holds instances of algorithm classes; the algorithm classes use the the template method pattern
- Advantages:
  - Open/Closed principle: Easily add new strategies
  - Dependencies inverted (`MyAnalysis` does not depend on the individual implementations)
  - Small number of subclasses
  - Separated implementation of algorithms from higher level code
  - For compiled languages: Change algorithms at runtime
- Warnings:
  - Might be overkill for very simple problems
  - For maximum performance, avoid virtual calls
- Alternatives:
  - If your language supports it: Use functions instead of objects (e.g. provide several `fit()` functions and pass them to the class)
  - Use template pattern if there is only one strategy that can be replaced

Command

The command pattern turns a method call into a standalone object.

Rather than directly calling a method, the interface creates a `Command` object (describing what we want to execute) and passing it on to a `Receiver` that executes it

Use cases:

- Decouple user interfaces from the backend (by using `Command` objects as means of communication)
- Build up a command history with undo functionality
- Remote execution of commands
- Queue or schedule operations

HEP specific use case example (Belle II software framework):

- Build up analysis by adding modules (`Command` objects) to a path (list of modules), each implementing a `event()` method to process one event
- After all modules are added, process the path: Loop over all events, calling the `event()` method of all modules in order

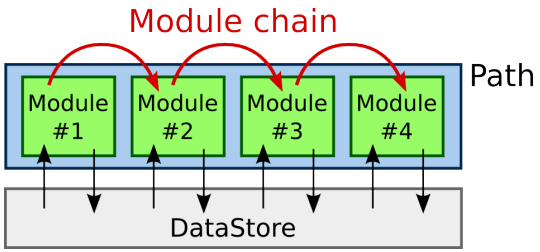
Command

Slightly simplified Belle II steering file:

```
# Create path to add modules (=Command objects) to
path = create_path()

# Load data (convenience function that adds a "DataLoader" module to the path)
inputMdstList("default", "/path/to/input/file", path=path)

# Get final state particles

# Fill 'pi+:loose' particle list with all particles that have pion ID > 0.01:
fillParticleList("pi+:loose", "piid > 0.01", path=path)
# Fill 'mu+:loose' particle list with all particles that have muon ID > 0.01:
fillParticleList("mu+:loose", "piid > 0.01", path=path)

# Reconstruct decay
# Fill 'K_S0:pipi' particle list with combinations of our pions and muons
reconstructDecay(
    "K_S0:pipi -> pi+:loose pi-:loose", "0.4 < M < 0.6", path=path
)

# Process path = call execute() on all Command objects
process(my_path)
```


Command

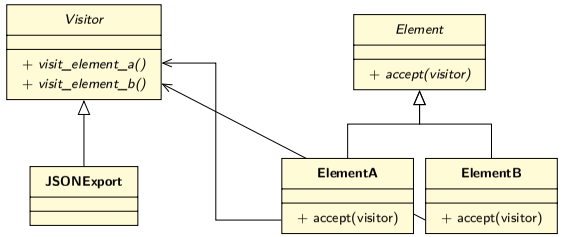
Upon processing path:



- Strictly declarative approach (no for loops or implementation details)
- Modules can be implemented in `python` or `C++`
- "Building block" approach makes steering files extremely easy to write and understand (even browser based graphical interface for highschoolers: try it at `masterclass.ijs.si`)

Visitor

- Concrete example: Serialize a collection of instances of different classes (e.g. provide JSON export for a list of different data objects)
- Possible implementation: Provide a `to_json()` method to all classes
- Potential issue: Might soon want to add export for export possibilities: (`XML`, `CSV`, `YAML`, etc.)
  - ⟹ more and more unrelated methods need to be added to the data class
  - ⟹ "polluted" interface (methods are irrelevant for core functionality); people might be wary of frequent changes to a well working class
- Solution: Separate algorithms from the objects they operate on



Visitor

Use case (more formally):

- Given a heterogeneous family of `Element` classes
- Do not expect significant changes to `Element` classes
- Various unrelated operations need to be performed on a collection of `Element` objects
- We expect frequent additions and changes for the operations
- Do not want to frequently change `Element` classes because of that

Advantages of visitor pattern:

- Single responsibility principle: All the operation functionality is in one place
- Open/Closed principle: Easy to add new operations

Disadvantages of the visitor pattern:

- No access to private information of `Element` classes
- Changes to `Element` classes can require changes to all visitors

Concurrency Patterns

Concurrency patterns need their own lecture, so this will only quickly mention basic concepts.

Use cases:

- Manage/synchronize access to shared resources (e.g. to avoid race conditions when several threads perform read and write operations)
- Scheduling tasks in parallel
- (A)synchronous event handling

Concurrency Patterns II

Advanced example: Active object pattern

- Want to decouple method calling from method execution
- Can request a calculation early and check later whether the result is available

The pattern consists of multiple components:

- The client calls a method of a proxy, which (immediately) returns a future object (can be used to check if results are available and get them)
- At the same time the proxy turns the method call into a request object and adds it to a request queue
- A scheduler takes requests from the request queue and executes it (on some thread)
- Once the request is executed, the result is added to the future object
- Only when the client accesses the future (wants to get the result value), the client thread waits (if the result is already available by that time, no waiting/blocking occurrs)

Anti-patterns

- God object, The blob: One massive class containing all functionality
- Object orgy: Not using data encapsulation (not distinguishing between public and private members); some objects modify the internals of others more than their own (⟶ it is not clear "who is doing what to whom")
- Not using polymorphism: Having many parallel sections of identical `if` statements rather than using classes and subclasses
- Misusing (multiple) inheritance: Some inherited methods do not make sense for subclass; violations of the Liskov substitution principle
- Overgeneralization/inner platform effect: A system so general and customizable that it reproduces your development platform

Common criticism

"Repetitive use of the same patterns and lots of boilerplate indicates lack or abstraction or lacking features of your programming language."

- Example: If functions are first-level objects (can be passed around like normal datatypes), I do not need to define a strategy class hierarchy.
  However, this could still be considered the same "pattern" (only with a simpler implementation)
- The "Patterns" give you vocabulary to describe your problem in an abstract way, even if the implementation details very a lot between languages
- Lots of pattern boilerplate should make you think about your design and language choices
- Be aware that the implementation of (or even the need for) certain patterns can be very dependent on your language features

Common criticism

"Design patterns are used excessively and introduce unneeded complexity."

- Remember the zen of python: simple is better than complex; but complex is better than complicated
- Do not introduce complexity (use the design pattern) if you do not fully understand why you need it.
- Some people highlight the KISS (keep it simple, stupid) and YAGNI (you aren't gonna need it) principle

Common criticism

"The common design patterns are often the direct results of thinking about good software design; focusing on patterns replaces actual thought with cut-and-paste programming."

- Take discussion of patterns as a mental practice of thinking about good design; **avoid** simple cut-and-paste

Outlook

Exercises tomorrow 09:00 – 11:00!

Get material at `github.com/klieret/icsc-paradigms-and-patterns`

C++ ABI: the only thing that is more important than performance

iCSC 2020

Nis Meinert

German Aerospace Center

DLR Deutsches Zentrum für Luft- und Raumfahrt

CERN School of Computing

Reading x86-64 Assembly

…for fun and profit

Function Prologue & Epilogue

- Few lines of code at the beginning (*prologue*) and end (*epilogue*) of a function, which **prepares** (and eventually restores)
  - the **stack** and
  - **registers**
- Not part of assembly: **convention** (defined & interpreted differently by different OS and compilers)

Prologue

```
1 push rbp     ; rbp: frame pointer
2 mov rbp, rsp ; rsp: stack pointer
3 sub rsp, N
```

alternatively

```
1 enter N, 0
```

(reserve N bytes on stack for local use)

Epilogue

```
1 mov rsp, rbp
2 pop rbp
3 ret
```

alternatively

```
1 leave
2 ret
```


Stack frame for function call

- CALL = PUSH *address of next instruction* + JMP *target*
- RET pops return address and transfers control there
- pass arguments 1 …6 in registers (`rsi`, `rdx`, …)

```
┌──────────────┐
│ ...          │
│ 8th Argument │ (rbp + 24)
│ 7th Argument │ (rbp + 16)
├──────────────┤
│ rip          │ (return address)
│ rbp          │ (rbp)
├──────────────┤
│ rbx          │
│ r12          │
│ r13          │ (rsp)
└──────────────┘
```

(stack frame for function call with 8 arguments and local registers `rbx`, `r12` and `r13`)

Reading assembly for fun and profit

```
1  int f(int x, int y, int z) {
2      int sum = x + y + z;
3      return sum;
4  }
```

godbolt.org/z/MaWcP9

```
# g92 -O0
   | _Z1fiii:
 1|   push rbp
 1|   mov rbp, rsp
 1|   mov DWORD PTR [rbp-20], edi
 1|   mov DWORD PTR [rbp-24], esi
 1|   mov DWORD PTR [rbp-28], edx
 2|   mov edx, DWORD PTR [rbp-20]
 2|   mov eax, DWORD PTR [rbp-24]
 2|   add edx, eax
 2|   mov eax, DWORD PTR [rbp-28]
 2|   add eax, edx
 2|   mov DWORD PTR [rbp-4], eax
 3|   mov eax, DWORD PTR [rbp-4]
 4|   pop rbp
 4|   ret
```

godbolt.org/z/MaWcP9

Reading assembly for fun and profit

```
1  int f(int x) {
2      return x + 1;
3  }
4
5  int g(int x) {
6      return f(x + 2);
7  }
```

godbolt.org/z/87GK4q

```
# g92 -O0
   | _Z1fi:
 1|   push rbp
 1|   mov rbp, rsp
 1|   mov DWORD PTR [rbp-4], edi
 2|   mov eax, DWORD PTR [rbp-4]
 2|   add eax, 1
 3|   pop rbp
 3|   ret
   | _Z1gi:
 5|   push rbp
 5|   mov rbp, rsp
 5|   sub rsp, 8
 5|   mov DWORD PTR [rbp-4], edi
 6|   mov eax, DWORD PTR [rbp-4]
 6|   add eax, 2
 6|   mov edi, eax
 6|   call _Z1fi
 7|   leave
 7|   ret
```

godbolt.org/z/87GK4q

Reading assembly for fun and profit

```
1  void side_effect();
2
3  int f(int x) {
4      auto a = x;
5      side_effect();
6      return a - x;
7  }
```

godbolt.org/z/5xq5n5

```
# g92 -O0
   | _Z1fi:
 3|   push rbp
 3|   mov rbp, rsp
 3|   sub rsp, 32
 3|   mov DWORD PTR [rbp-20], edi
 4|   mov eax, DWORD PTR [rbp-20]
 4|   mov DWORD PTR [rbp-4], eax
 5|   call _Z11side_effectv
 6|   mov eax, DWORD PTR [rbp-4]
 6|   sub eax, DWORD PTR [rbp-20]
 7|   leave
 7|   ret
```

godbolt.org/z/5xq5n5

Name mangling: C++ vs C

```
1  int f(int x) {
2      return x * x;
3  }
4
5  extern "C" int g(int x) {
6      return x * x;
7  }
```

godbolt.org/z/cj7bqx

```
# g92 -O0
   | _Z1fi:
 1|   push rbp
 1|   mov rbp, rsp
 1|   mov DWORD PTR [rbp-4], edi
 2|   mov eax, DWORD PTR [rbp-4]
 2|   imul eax, eax
 3|   pop rbp
 3|   ret
   | g:
 5|   push rbp
 5|   mov rbp, rsp
 5|   mov DWORD PTR [rbp-4], edi
 6|   mov eax, DWORD PTR [rbp-4]
 6|   imul eax, eax
 7|   pop rbp
 7|   ret
```

godbolt.org/z/cj7bqx

Name mangling: C++ vs C

```
1  int f(int x) {
2      return x * x;
3  }
4
5  extern "C" int g(int x) {
6      return x * x;
7  }
```

godbolt.org/z/cj7bqx

```
# g92 -O1
   | _Z1fi:
 2|   imul edi, edi
 3|   mov eax, edi
 3|   ret
   | g:
 6|   imul edi, edi
 7|   mov eax, edi
 7|   ret
```

godbolt.org/z/87z8vz

Why?

- → overloading
- → namespaces
- → templating

(Name of function doesn't suffice to resolve JMP location)

Name mangling in C++

```
1  void f(int) {}
2
3  void f(double) {}
4
5  namespace my_fancy_namespace {
6  void f(int) {}
7  } // my_fancy_namespace
```

godbolt.org/z/jWY14x

```
# g92 -O2
   | _Z1fi:
 1|   ret
   | _Z1fd:
 3|   ret
   | _ZN18my_fancy_namespace1fEi:
   |   ret
```

godbolt.org/z/jWY14x

- → C++ does not standardize name mangling
- → *Annotated C++ Reference Manual* even actively discourages usage of common mangling schemes. (Prevent linking when other aspects of ABI are incompatible.)

cf. name mangling for Itanium ABI:
`itanium-cxx-abi.github.io/cxx-abi/abi.html#mangling-structure`

What is ABI?

What is ABI (*Application Binary Interface*)?

Specifies interaction of functions and types across TUs† (translation units)

- → Platform-specific (*e.g.*, Linux on x86-64 CPU)
- → Vendor-specified (*e.g.*, gcc)
- → not controlled by WG21

(Titus Winters: *Similar to a binary network protocol*)

Photo by Spencerian at `en.wikipedia.org` (2005)

† *TU*: ultimate input to the compiler from which an object file is generated (*i.e.*, typically the `.cpp` file)

What is ABI (*Application Binary Interface*)?

Specifies interaction of functions and types across TUs[†] (translation units) covering:

- Name mangling of functions
- Name mangling of types
- `sizeof` and alignment of objects
- Bytes semantics of the binary representation of objects
- Calling convention

(Titus Winters: *Similar to a binary network protocol*)

Photo by Spencerian at `en.wikipedia.org` (2005)

[†] *TU*: ultimate input to the compiler from which an object file is generated (*i.e.*, typically the `.cpp` file)

Why should I care?

…do you depend on any pre-compiled shared library?

Why should I care?

Why should I care?

- **Linking** different TUs requires usage of same ABI
- Typically a problem at API boundaries when combining TUs (*e.g.*, shared libraries) that were compiled at different **time**s
- Similar to binary network protocols: ABI tells you how to interpret bytes

Why should I care? ⇔ Why do network protocols have versions?

(Problem: not all ABIs encode version number)

ABI: the problem

ABI does not encode version number

- **Q**: How to check if a given TU uses a compatible ABI?
- **A**: You can't!
- What happens if ABI is incompatible?
  (a) Linking fails during compile time (good)
  (b) Program spectacularly dies during run time (bad)
- Why isn't this a common problem?
  - Itanium ABI is mostly stable since C++11

ABI breakage of `std::string`

- Before C++11: libstdc++ relied on copy-on-write (COW)
- C++11 disallows COW
  - fewer indirections
  - short string optimization (SSO)
- Problem: passing COW string to impl that expects SSO **may link** (same mangled name)!
  - one (quad-)word passed
  - three (quad-)words read
- *Solution*[†]: gcc changed mangled name

```cpp
// until C++11
struct string {
    struct control_block {
        /* ... */
    };
    control_block *data;
};

// since C++11
struct string {
    char *data;
    std::size_t size;
    std::size_t capacity;
}
```

godbolt.org/z/KM5Tvq

↪ Take-away for compiler vendors: ABI break was a huge disaster

[†] RHEL 7 still uses old `std::string` ABI to provide compatibility for older `.so`

Quiz Time

Quiz: Will it break ABI?

Proposal: make `std::vector<T>::push_back` return a reference to the element in its new location

```cpp
void push_back(const T&);
```

↓

```cpp
T& push_back(const T&);
```


Quiz: Will it break ABI?

```cpp
template <typename T>
struct vector {
    void push_back_1(const T&);
    T& push_back_2(const T&);
};

void f(vector<int> v) {
    v.push_back_1(42);
    v.push_back_2(42);
}
```

godbolt.org/z/9def7a

```
# g92 -O0
  | _Z1f6vectorIiE:
 7|   push rbp
 7|   mov rbp, rsp
 7|   sub rsp, 16
 8|   mov DWORD PTR [rbp-8], 42
 8|   lea rax, [rbp-8]
 8|   mov rsi, rax
 8|   lea rdi, [rbp+16]
 8|   call _ZN6vectorIiE11push_back_1ERKi
 9|   mov DWORD PTR [rbp-4], 42
 9|   lea rax, [rbp-4]
 9|   mov rsi, rax
 9|   lea rdi, [rbp+16]
 9|   call _ZN6vectorIiE11push_back_2ERKi
10|   nop
10|   leave
10|   ret
```

godbolt.org/z/9def7a

Quiz: Will it break ABI?

Both, `void push_back` and `T& push_back` have the same mangled name (Itanium ABI)

- → **Two** definitions in the old and the new TU
- → ODR violation
- → Linker will pick only **one** definition (by **overwriting** the other)
- → ABI break: reading return value from `eax` when there is none

Quiz: Will it break ABI?

Proposal: make `std::vector<T>::emplace_back` return a reference to the element in its new location

```
template<class... Args> void emplace_back(Args&&...);
                        ↓
template<class... Args> T& emplace_back(Args&&...);
```


Quiz: Will it break ABI?

```
1  template <typename T>
2  struct vector {
3      template<class... Args>
4      void emplace_back_1(Args&&...);
5
6      template<class... Args>
7      T& emplace_back_2(Args&&...);
8  };
9
10 void f(vector<int> v) {
11     v.emplace_back_1(42);
12     v.emplace_back_2(42);
13 }
```

godbolt.org/z/dYMsza

Mangled names: (Itanium ABI)

1. `_ZN6vectorIiE14emplace_back_1IJiEEEvDpOT_`
2. `_ZN6vectorIiE14emplace_back_2IJiEEERiDpOT_`

Quiz: Will it break ABI?

`void emplace_back` and `T& emplace_back` have different mangled names (*cf.* Itanium ABI: *5.1.5.3 Function types*)

- → Two definitions in the old and the new TU
- → but no ODR violation
- → No ABI break: old code calls the old one, new code calls the new one
  - → proposal was accepted for `C++20`

Quiz: Will it break ABI?

Proposal: extend `std::lock_guard<T>` to allow for a variadic set of heterogeneous mutexes

```
template<class Mutex> class lock_guard;
                      ↓
template<class... Mutexes> class lock_guard;
```


Quiz: Will it break ABI?

```
1  template <typename>
2  struct lock_guard_1 {
3      lock_guard_1() {}
4  };
5
6  template <typename...>
7  struct lock_guard_2 {
8      lock_guard_2() {}
9  };
10
11 void f() {
12     lock_guard_1<int> l1{};
13     lock_guard_2<int> l2{};
14 }
```

godbolt.org/z/MKPq35

```
# g92 -O0
  | _Z1fv:
11|   push rbp
11|   mov rbp, rsp
11|   sub rsp, 16
12|   lea rax, [rbp-1]
12|   mov rdi, rax
12|   call _ZN12lock_guard_1IiEC1Ev
13|   lea rax, [rbp-2]
13|   mov rdi, rax
13|   call _ZN12lock_guard_2IJiEEC1Ev
14|   nop
14|   leave
14|   ret
  | _ZN12lock_guard_1IiEC2Ev:
 3|   push rbp
 3|   mov rbp, rsp
 3|   mov QWORD PTR [rbp-8], rdi
 3|   nop
[...]
```

godbolt.org/z/MKPq35

Quiz: Will it break ABI?

`<class T> class` and `<class... T> class` have different mangled names (Itanium ABI)

- → ABI break: for example in `auto f(std::lock_guard<M>& lk);`
  - → *cf.* `godbolt.org/z/Pex6Gx`
- → User compiles `f` using old `lock_guard`
- → User then tries to call it from a TU using new `lock_guard`
- → Mangled names don't match: linker error!

Quiz: Will it break ABI?

Proposal: change hashing by `std::hash` to improve performance of `std::unordered_map` by 3-4x (*cf.* `absl::node_hash_map`)

ABI break:

- → Hash value for an object is computed in old TU and stored in map
- → (Different) hash value is computed in new TU and used to lookup value in map
- → **Semantic meaning of binary representation has changed!**

Other Examples

More examples:

- `std::regex` currently is 10-100x slower than equivalents in Rust or Go (*cf.* any talk of Hana Dusíková)
- Make `std::unique_ptr` zero-overhead (*cf.* Chandler Carruth, *There Are No Zero-cost Abstractions*)
- Add `std::int128_t` which already is supported on more and more platforms
- Make `std::bitset` trivially destructible
- Exceptions (would be an entire talk on its own)
- …

(read P2028 and P1863 by Titus Winters for more information)

Future Prospects

Future Prospects

Break ABI with future releases: move run-time failures to compile-time (when possible) by changing mangled name

- new namespaces: *e.g.*, `std2::` or `std::abi42::`
  - developers now have to choose between `std::optional` and `std2::optional` or duplicate code with overloading
  - when will `std3::` arrive?
- change entire mangling scheme: *e.g.*, `_Z` ↦ `_Y` on Itanium
  - Compiler vendors could ship both forms
- Introduce version number (ABI break)
  - MSVC does already include version numbers in DLLs (MSVCs users are used to recompile with each new version of Visual Studio …)

State-of-the-Art

- No (major?) ABI breaks since C++11 (**12 years in 2023!**)
- Hyrum's Law (or `xkcd.com/1172`): passing ABI-unstable types across ABI boundaries happened to work for more than a decade. People rely on ABI stability, even though it was never explicitly promised.
- Expose C-APIs whenever possible

State-of-the-Art

Standard Meeting, Prague 2020

- WG21 is not in favor of an ABI break in C++23 or C++26
- WG21 is in favor of an ABI break in a future version of C++
- WG21 will take time to consider proposals requiring an ABI break
- WG21 will not promise stability forever
- WG21 wants to keep prioritizing performance over stability

Bryce | BlackLivesMatter @blelba... · 16 Feb
Performance
ABI Stability
Ability to Change

You can pick two, choose wisely.

#cppprg

25 12 63

Twitter: `@blebach` (2020)

(Corentin Jabot: *There was no applause. But I'm not sure we fully understood what we did and the consequences it could have.*)

State-of-the-Art

Jonathan Müller @foonathan · 03 Feb
What's the point of standard library containers if they can't give the best performance?

2 11

Titus Winters
@TitusWinters

Replying to @foonathan

They're common and readily available? (Which does have some value.)

Committing to ABI is like admitting that the standard library is aiming to be McDonald's - It's everywhere, it's consistent, and it technically solves the problem.

18:02 · 03 Feb 20 · Twitter Web App

2 Retweets **1** Quote Tweet **17** Likes

Twitter: `@TitusWinters` (2020)

STL alternatives

- `abseil` by Google: `abseil.io/about/`
- `folly` by Facebook: `github.com/facebook/folly`
- …

Literature

- Titus Winters, `P2028`: *What is ABI, and What Should WG21 Do About It?*
- Titus Winters, `P1863`: *ABI - Now or Never*
- Roger Orr, `P1654`: *ABI breakage - summary of initial comments*
- Titus Winters on CppCast #224: *The C++ ABI*
- John Lakos on CppCast #233: *Large Scale C++*
- Corentin Jabot on `cor3ntin.github.io/posts/abi/`: *The Day The Standard Library Died*
- JeanHeyd Meneide on `thephd.github.io/freestanding-noexcept-allocators-vector-memory-hole`
- Danila Kutenin on `youtu.be/GRuX31P4Ric`: *C++ STL best and worst performance features and how to learn from them*

Demystifying Value Categories in C++

iCSC 2020

Nis Meinert

Rostock University

Disclaimer

Disclaimer

- This talk is mainly about hounding (unnecessary) copy ctors
- In case you don't care:

"If you're not at all interested in performance, shouldn't you be in the Python room down the hall?" (Scott Meyer)

Table of Contents

PART I

Understanding References

Q: What is the output of the programs?

```python
#!/usr/bin/env python3

class S:
    def __init__(self, x):
        self.x = x

def swap(a, b):
    b, a = a, b

if __name__ == '__main__':
    a, b = S(1), S(2)
    swap(a, b)
    print(f'{a.x}{b.x}')
```

```cpp
#include <iostream>

struct S {
    int x;
};

void swap(S& a, S& b) {
    S tmp = a;
    a = b;
    b = tmp;
}

int main() {
    S a{1}; S b{2};
    swap(a, b);
    std::cout << a.x << b.x;
}
```

godbolt.org/z/rE6Ecd

Q: What is the output of the program?

```cpp
#include <iostream>

struct S {
    int x;
};

void swap(S& a, S& b) {
    S tmp = a;
    a = b;
    b = tmp;
}

int main() {
    S a{1}; S b{2};
    swap(a, b);
    std::cout << a.x << b.x;
}
```

godbolt.org/z/r6oq55

Q: What is the output of the program?

```cpp
#include <iostream>

struct S {
    int x;
    S(int x): x(x) { std::cout << 'a'; }
    S(const S& other): x(other.x) { std::cout << 'b'; }
    S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
};

void swap(S& a, S& b) {
    S& tmp = a;
    a = b;
    b = tmp;
}

int main() {
    S a{1}; S b{2};
    swap(a, b);
    std::cout << a.x << b.x;
}
```

godbolt.org/z/jfM6h1

Q: What is the output of the program?

```cpp
#include <iostream>

struct S {
    int x;
    S(int x): x(x) { std::cout << 'a'; }
    S(const S& other): x(other.x) { std::cout << 'b'; }
    S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
};

void swap(S& a, S& b) {
    S tmp = a;
    a = b;
    b = tmp;
}

int main() {
    S a{1}; S b{2};
    swap(a, b);
    std::cout << a.x << b.x;
}
```

godbolt.org/z/ohe3Wb

Q: What is the output of the program?

```cpp
#include <iostream>

struct S {
    int x;
    S(int x): x(x) { std::cout << 'a'; }
    S(const S& other): x(other.x) { std::cout << 'b'; }
    S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
};

void swap(S* a, S* b) {
    S* tmp = a;
    a = b;
    b = tmp;
}

int main() {
    S a{1}; S b{2};
    swap(&a, &b);
    std::cout << a.x << b.x;
}
```

godbolt.org/z/8fovsa

Q: What is the output of the program?

```cpp
#include <iostream>

struct S {
    int x;
    S(int x): x(x) { std::cout << 'a'; }
    S(const S& other): x(other.x) { std::cout << 'b'; }
    S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
};

void swap(S* a, S* b) {
    S* tmp = a;
    a = b;
    b = tmp;
}

int main() {
    S a{1}; S b{2}; S* a_ptr = &a; S* b_ptr = &b;
    swap(a_ptr, b_ptr);
    std::cout << a_ptr->x << b_ptr->x;
}
```

godbolt.org/z/6357rq

Q: What is the output of the program?

```cpp
#include <iostream>

struct S {
    int x;
    S(int x): x(x) { std::cout << 'a'; }
    S(const S& other): x(other.x) { std::cout << 'b'; }
    S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
};

void swap(S*& a, S*& b) {
    S* tmp = a;
    a = b;
    b = tmp;
}

int main() {
    S a{1}; S b{2}; S* a_ptr = &a; S* b_ptr = &b;
    swap(a_ptr, b_ptr);
    std::cout << a_ptr->x << b_ptr->x;
}
```

godbolt.org/z/dEsxEY

Q: What is the output of the program?

```cpp
#include <iostream>

struct S {
    int x;
    S(int x): x(x) { std::cout << 'a'; }
    S(const S& other): x(other.x) { std::cout << 'b'; }
    S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
};

void swap(S*& a, S*& b) {
    S* tmp = a;
    a = b;
    b = tmp;
}

int main() {
    S a{1}; S b{2};
    swap(&a, &b);
    std::cout << a.x << b.x;
}
```

godbolt.org/z/Eh656x

Q: What is the output of the program?

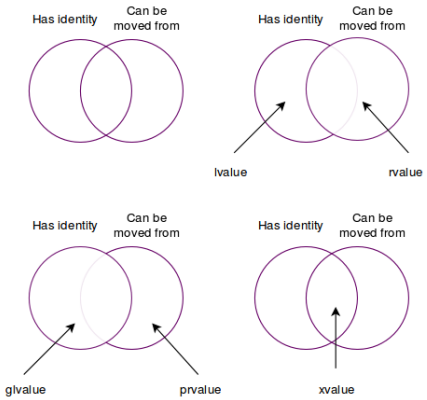
error: cannot bind non-const lvalue reference of type “S*&” to an rvalue of type “S*”

```cpp
#include <iostream>

struct S {
    int x;
    S(int x): x(x) { std::cout << 'a'; }
    S(const S& other): x(other.x) { std::cout << 'b'; }
    S& operator=(const S& other) { x = other.x; std::cout << 'c'; return *this; }
};

void swap(S*& a, S*& b) {
    S* tmp = a;
    a = b;
    b = tmp;
}

int main() {
    S a{1}; S b{2};
    swap(&a, &b);
    std::cout << a.x << b.x;
}
```

godbolt.org/z/Eh656x

Value Categories

Value categories with Venn diagrams

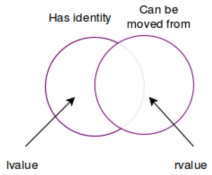


(diagrams shamelessly stolen from bajamircea.github.io/coding/cpp/2016/04/07/move-forward.html)

Value categories with Venn diagrams

(diagrams shamelessly stolen from bajamircea.github.io/coding/cpp/2016/04/07/move-forward.html)

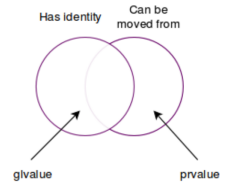
```
1  struct S{ int x; };
2
3  S make_S(int x) {
4      S s{.x = x};
5      return s; // has no name after returning
6  }
7
8  int main() {
9      S a = make_S(42); // `a` is an lvalue
10                       // initialized with a prvalue
11
12     S b = std::move(a); // prepare to die, `a`!
13                         // now `a` became an xvalue
14
15     auto x = a.x; // ERROR: `a` is in an undefined state
16     a = make_S(13);
17     x = a.x; // fine!
18 }
```



Value categories with Venn diagrams

(diagrams shamelessly stolen from bajamircea.github.io/coding/cpp/2016/04/07/move-forward.html)

```
1  struct S{ int x; };
2
3  S make_S(int x) {
4      S s{.x = x};
5      return s; // has no name after returning
6  }
7
8  int main() {
9      S a = make_S(42); // `a` is an lvalue
10                       // initialized with a prvalue
11
12     S b = std::move(a); // prepare to die, `a`!
13                         // now `a` became an xvalue
14
15     auto x = a.x; // ERROR: `a` is in an undefined state
16     a = make_S(13);
17     x = a.x; // fine!
18 }
```



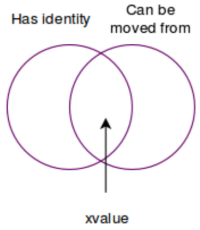
Nis Meinert – Rostock University Demystifying Value Categories in C++ – Value Categories 13 / 101

88

Value categories with Venn diagrams

(diagrams shamelessly stolen from bajamircea.github.io/coding/cpp/2016/04/07/move-forward.html)

```
1  struct S{ int x; };
2
3  S make_S(int x) {
4      S s{.x = x};
5      return s; // has no name after returning
6  }
7
8  int main() {
9      S a = make_S(42); // `a` is an lvalue
10                       // initialized with a prvalue
11
12     S b = std::move(a); // prepare to die, `a`!
13                         // now `a` became an xvalue
14
15     auto x = a.x; // ERROR: `a` is in an undefined state
16     a = make_S(13);
17     x = a.x; // fine!
18 }
```



Nis Meinert – Rostock University Demystifying Value Categories in C++ – Value Categories 13 / 101

Binding references to temporaries

error: cannot bind non-const lvalue reference of type “S*&” to an rvalue of type “S*”

```
1  template <typename T>
2  void swap(T& a, T& b) { ... }
3
4  int main() {
5      S a{1};
6      S b{2};
7      swap(&a, &b);
8  }
```

- → Memory addresses are always rvalues!
- → One cannot refer to something that doesn’t has a name…
- → …except it is a const reference (lifetime extension)

std::move

std::move

```
1  #include <iostream>
2  #include <utility>
3
4  struct S{};
5
6  void f(const S&) { std::cout << 'a'; }
7  void f(S&&) { std::cout << 'b'; }
8
9  int main() {
10     S s;
11     f(s);            // prints 'a'
12     f(std::move(s)); // prints 'b'
13 }
```

godbolt.org/z/aKbGEc

- `std::move` creates xvalues
- Syntax:
  - lvalue ref.: S&
  - rvalue ref.: S&&

Q: What is the output of the program?

```
1  #include <iostream>
2  #include <utility>
3
4  struct S{
5      S() { std::cout << 'a'; }
6      S(const S&) { std::cout << 'b'; }
7      S(S&&) { std::cout << 'c'; }
8  };
9
10 int main() {
11     S s1;
12     S s2(s1);
13     S s3(S{});
14     S s4(std::move(s1));
15 }
```

godbolt.org/z/16hYbz

A: abac

```
1  #include <iostream>
2  #include <utility>
3
4  struct S{
5      S() { std::cout << 'a'; }
6      S(const S&) { std::cout << 'b'; }
7      S(S&&) { std::cout << 'c'; }
8  };
9
10 int main() {
11     S s1;
12     S s2(s1);
13     S s3(S{});
14     S s4(std::move(s1));
15 }
```

godbolt.org/z/16hYbz

- `S s1`: no surprise
- `S s2(s1)`: no surprise
- `S s3(S{})`: *mandatory* copy elision (initializer is prvalue of the same class type)
- `S s4(std::move(s1))`: forced move construction

Q: What is the output of the program?

```cpp
#include <iostream>
#include <utility>

struct S {
    S() { std::cout << 'a'; }
    S(const S&) { std::cout << 'b'; }
    S(S&&) { std::cout << 'c'; }
};

void f(const S&) { std::cout << '1'; }
void f(S&) { std::cout << '2'; }
void f(S&&) { std::cout << '3'; }

int main() {
    S s1;
    f(s1);
    f(S{});
    f(std::move(s1));
}
```

godbolt.org/z/4MKojT

Q: What is the output of the program?

```cpp
#include <iostream>
#include <utility>

struct S {
    S() { std::cout << 'a'; }
    S(const S&) { std::cout << 'b'; }
    S(S&&) { std::cout << 'c'; }
};

void f(const S&) { std::cout << '1'; }
void f(S) { std::cout << '2'; }
void f(S&&) { std::cout << '3'; }

int main() {
    S s1;
    f(s1);
    f(S{});
    f(std::move(s1));
}
```

godbolt.org/z/jaYTYP

Q: What is the output of the program?

```cpp
#include <iostream>
#include <utility>

struct S {
    S() { std::cout << 'a'; }
    S(const S&) { std::cout << 'b'; }
    S(S&&) { std::cout << 'c'; }
};

void f(const S&) { std::cout << '1'; }
void f(S) { std::cout << '2'; }
void f(S&&) { std::cout << '3'; }

int main() {
    S s1;
    f(s1);
    f(S{});
    f(std::move(s1));
}
```

godbolt.org/z/jaYTYP

Compile-time error (in all three cases)

- → `f(s1)`: ambiguity between 2 and 1
- → `f(S{})`: ambiguity between 2 and 3
- → `f(std::move(s1))`: same as `f(S)`

↪ compiler cannot differentiate between copy and reference overloads! (neither lvalue, nor rvalue)

Q: What is the output of the program?

```cpp
#include <iostream>
#include <utility>

struct S {
    ~S() { std::cout << 'a'; }
};

void f(const S&) { std::cout << '1'; }
void f(S&) { std::cout << '2'; }
void f(S&&) { std::cout << '3'; }

int main() {
    S&& r1 = S{};
    f(r1);

    S&& r2 = S{};
    f(std::move(r2));
}
```

godbolt.org/z/5s1zc5

A: 23aa

```cpp
#include <iostream>
#include <utility>

struct S {
    ~S() { std::cout << 'a'; }
};

void f(const S&) { std::cout << '1'; }
void f(S&) { std::cout << '2'; }
void f(S&&) { std::cout << '3'; }

int main() {
    S&& r1 = S{};
    f(r1);

    S&& r2 = S{};
    f(std::move(r2));
}
```

godbolt.org/z/5s1zc5

- S&&: object that nobody cares about anymore and which will die soon (cf. lifetime extension!)
- `std::move` does not actually kill, but makes the object look like a dying object

NB: an rvalue ref behaves like an lvalue ref except that it can bind to a temporary (an rvalue), whereas one cannot bind a (non const) lvalue ref to an rvalue.

std::move

```cpp
#include <type_traits>

template <typename T>
decltype(auto) move(T&& t) {
    using R = std::remove_reference_t<T>&&;
    return static_cast<R>(t);
}
```

godbolt.org/z/W8zb8G

So what does `std::move`?

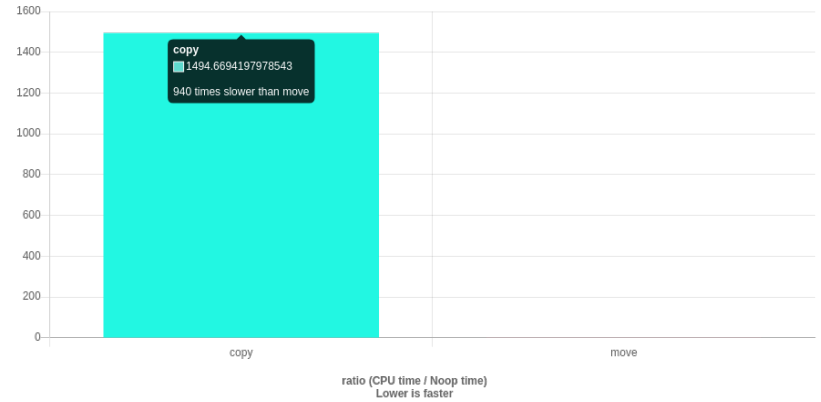
- does not *move*
- does not destroy
- does nothing at all during runtime
- **unconditionally casts** its argument to an rvalue

Quick Bench: tinyurl.com/y67sg7to

```cpp
std::vector<int> x(1000, 42);
std::vector<int> y(1000, 42);
for (auto _ : state) {
    auto tmp = x;
    x = y;
    y = tmp;
    benchmark::DoNotOptimize(x[345] + y[678]);
}
```

```cpp
std::vector<int> x(1000, 42);
std::vector<int> y(1000, 42);
for (auto _ : state) {
    auto tmp = std::move(x);
    x = std::move(y);
    y = std::move(tmp);
    benchmark::DoNotOptimize(x[345] + y[678]);
}
```


Quick Bench: tinyurl.com/y67sg7to



Universal References

Rvalue ref. or no rvalue ref.?

Rvalue refs are declared using “&&”: reasonable to assume that the presence of “&&” in a type declaration indicates an rvalue reference?

```
struct S{};

S&& s = S{};                    // (1)

auto&& s2 = s;                  // (2)

void f(S&& s);                  // (3)

template <typename T>
void f(T&& t);                  // (4)

template <typename T>
void f(const T&& t);            // (5)

template <typename T>
void f(std::vector<T>&& v); // (6)
```

Does “&&” mean rvalue reference?

- (1): ???
- (2): ???
- (3): ???
- (4): ???
- (5): ???
- (6): ???

Rvalue ref. or no rvalue ref.?

Rvalue refs are declared using “&&”: reasonable to assume that the presence of “&&” in a type declaration indicates an rvalue reference?

```
struct S{};

S&& s = S{};                    // (1)

auto&& s2 = s;                  // (2)

void f(S&& s);                  // (3)

template <typename T>
void f(T&& t);                  // (4)

template <typename T>
void f(const T&& t);            // (5)

template <typename T>
void f(std::vector<T>&& v); // (6)
```

Does “&&” mean rvalue reference?

- (1):
- (2):
- (3):
- (4):
- (5):
- (6):

* albeit questionable: `move` changes object in most cases ↛ `const`

* albeit questionable: `move` changes object in most cases ↛ `const`

```
1  #include <iostream>
2
3  struct S {
4      S() {}
5      S(const S&) { std::cout << 'A'; }
6      S(S&&) { std::cout << 'B'; }
7  };
8
9  int main() {
10     const S s;
11     auto s2 = std::move(s);
12 }
```

godbolt.org/z/r9hv8K

…prints A

(cf. https://stackoverflow.com/a/28595415)

Universal references

Universal references†

- Syntax (x is a universal reference):
  - `auto&& x`
  - `template <typename T> f(T&& x…`
- Rule of thumb: substitute fully qualified type into `auto` or T and reduce:
  - && ↦ &&
  - &&& ↦ &
  - &&&& ↦ &&

```
std::vector<S> v;
auto&& s = v[0]; // S&&& -> S&

auto&& s2 = S{}; // S&&&& -> S&&
auto&& s3 = s2;  // S&&&  -> S&

// S&&&& -> S&&
auto&& s3 = std::move(s2);
```

† *Universal reference*: term introduced by Scott Meyers

Q: What is the output of the program?

```
#include <iostream>
#include <type_traits>

struct S {
    S() { std::cout << 'a'; }
    S(const S&) { std::cout << 'b'; }
    S(S&&) { std::cout << 'c'; }
};

template <typename T>
S f(T&& t) { return t; }

int main() {
    S s{};
    f(s);
    f(std::move(s));
}
```

godbolt.org/z/6xn1n3

Q: What is the output of the program?

```
#include <iostream>
#include <type_traits>

struct S{};

void f(S&) { std::cout << 'a'; }
void f(S&&) { std::cout << 'b'; }

int main() {
    auto&& r1 = S{};
    static_assert(std::is_same_v<decltype(r1), S&&>);
    f(r1);
    f(static_cast<S&&>(r1));

    S s;
    auto&& r2 = s;
    static_assert(std::is_same_v<decltype(r2), S&>);
    f(r2);
}
```

godbolt.org/z/zTExze

Q: What is the output of the program?

```
#include <iostream>

struct S{
    void f() & { std::cout << 'a'; }
    void f() && { std::cout << 'b'; }
};

int main() {
    auto&& r1 = S{};
    r1.f();
    static_cast<decltype(r1)>(r1).f();

    auto&& r2 = r1;
    r2.f();
    static_cast<decltype(r2)>(r2).f();
}
```

godbolt.org/z/WcYYsd

Perfect forwarding

How do we fuse these implementations?

```cpp
// if `t` is an lvalue of type `S`
S& forward(S& t) {
    return t;
}

// if `t` is an rvalue of type `S`
S&& forward(S& t) { // not `S&&`!
    return std::move(t); // static_cast<S&&>(t)
}
```

```cpp
#include <type_traits>

template <typename T>
T&& forward(std::remove_reference_t<T>& t) {
    return static_cast<T&&>(t);
}
```

godbolt.org/z/EjPnPr

Q: What is the output of the program?

```cpp
#include <iostream>
#include <type_traits>

struct S {
    S() { std::cout << 'a'; }
    S(const S&) { std::cout << 'b'; }
    S(S&&) { std::cout << 'c'; }
};

template <typename T>
S f(T&& t) { return std::forward<T>(t); }

int main() {
    S s{};
    f(s);
    f(std::move(s));
}
```

godbolt.org/z/7Worb3

A: abc

```cpp
#include <iostream>
#include <type_traits>

struct S {
    S() { std::cout << 'a'; }
    S(const S&) { std::cout << 'b'; }
    S(S&&) { std::cout << 'c'; }
};

template <typename T>
S f(T&& t) { return std::forward<T>(t); }

int main() {
    S s{};
    f(s);
    f(std::move(s));
}
```

godbolt.org/z/7Worb3

Rule of thumb: Use `std::move` for rvalues and `std::forward` for universal references

Q: Why can't we use perfect forwarding here?

```cpp
#include <functional>

template <typename Iter, typename Callable, typename... Args>
void foreach (Iter current, Iter end, Callable op, const Args&... args) {
    while (current != end) {
        std::invoke(op, args..., *current);
        ++current;
    }
}
```

godbolt.org/z/TvnEfT

Reading x86-64 Assembly

…for fun and profit

Function Prologue & Epilogue

- Few lines of code at the beginning (*prologue*) and end (*epilogue*) of a function, which **prepares** (and eventually restores)
  - the **stack** and
  - **registers**
- Not part of assembly: **convention** (defined & interpreted differently by different OS and compilers)

Prologue

```
1 push rbp     ; rbp: frame pointer
2 mov rbp, rsp ; rsp: stack pointer
3 sub rsp, N
```

alternatively

```
1 enter N, 0
```

(reserve N bytes on stack for local use)

Epilogue

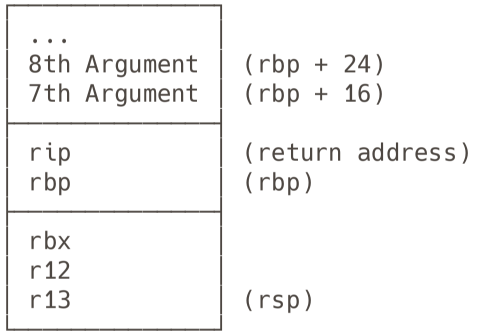
```
1 mov rsp, rbp
2 pop rbp
3 ret
```

alternatively

```
1 leave
2 ret
```


Stack frame for function call

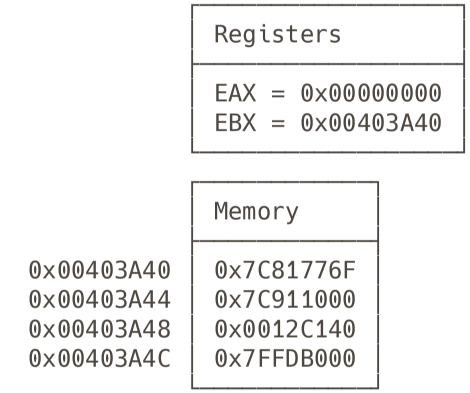
- CALL = PUSH *address of next instruction* + JMP *target*
- RET pops return address and transfers control there
- pass arguments 1 …6 in registers (`rsi`, `rdx`, …)



(stack frame for function call with 8 arguments and local registers `rbx`, `r12` and `r13`)

`lea` vs. `mov`

- `lea`: load effective address
- puts memory address from `src` into the destination `dest`
- Example: `lea eax, [ebx+8]`
  - put `[ebx+8]` into `eax`
  - value of `eax` after instruction: `0x00403A48`
- …whereas: `mov eax, [ebx+8]`
  - value of `eax` after instruction: `0x0012C140`



Reading assembly for fun and profit

```
1 int f(int x, int y, int z) {
2     int sum = x + y + z;
3     return sum;
4 }
```

godbolt.org/z/MaWcP9

```
# g92 -O0
  | f(int, int, int):
 1|   push rbp
 1|   mov rbp, rsp
 1|   mov DWORD PTR [rbp-20], edi
 1|   mov DWORD PTR [rbp-24], esi
 1|   mov DWORD PTR [rbp-28], edx
 2|   mov edx, DWORD PTR [rbp-20]
 2|   mov eax, DWORD PTR [rbp-24]
 2|   add edx, eax
 2|   mov eax, DWORD PTR [rbp-28]
 2|   add eax, edx
 2|   mov DWORD PTR [rbp-4], eax
 3|   mov eax, DWORD PTR [rbp-4]
 4|   pop rbp
 4|   ret
```

godbolt.org/z/MaWcP9

Reading assembly for fun and profit

```
1 int f(int x, int y, int z) {
2     int sum = x + y + z;
3     return sum;
4 }
```

godbolt.org/z/MaWcP9

```
# g92 -O1
  | f(int, int, int):
 2|   add edi, esi
 2|   lea eax, [rdi+rdx]
 4|   ret
```

godbolt.org/z/67WsqT

```
# g92 -O0
  | f(int, int, int):
 1|   push rbp
 1|   mov rbp, rsp
 1|   mov DWORD PTR [rbp-20], edi
 1|   mov DWORD PTR [rbp-24], esi
 1|   mov DWORD PTR [rbp-28], edx
 2|   mov edx, DWORD PTR [rbp-20]
 2|   mov eax, DWORD PTR [rbp-24]
 2|   add edx, eax
 2|   mov eax, DWORD PTR [rbp-28]
 2|   add eax, edx
 2|   mov DWORD PTR [rbp-4], eax
 3|   mov eax, DWORD PTR [rbp-4]
 4|   pop rbp
 4|   ret
```

godbolt.org/z/MaWcP9

Reading assembly for fun and profit

```
1 int f(int x) {
2     return x + 1;
3 }
4
5 int g(int x) {
6     return f(x + 2);
7 }
```

godbolt.org/z/87GK4q

```
# g92 -O0
  | f(int):
 1|   push rbp
 1|   mov rbp, rsp
 1|   mov DWORD PTR [rbp-4], edi
 2|   mov eax, DWORD PTR [rbp-4]
 2|   add eax, 1
 3|   pop rbp
 3|   ret
  | g(int):
 5|   push rbp
 5|   mov rbp, rsp
 5|   sub rsp, 8
 5|   mov DWORD PTR [rbp-4], edi
 6|   mov eax, DWORD PTR [rbp-4]
 6|   add eax, 2
 6|   mov edi, eax
 6|   call f(int)
 7|   leave
 7|   ret
```

godbolt.org/z/87GK4q

Reading assembly for fun and profit

```
1 int f(int x) {
2     return x + 1;
3 }
4
5 int g(int x) {
6     return f(x + 2);
7 }
```

godbolt.org/z/87GK4q

```
# g92 -O1
  | f(int):
 2|   lea eax, [rdi+1]
 3|   ret
  | g(int):
 2|   lea eax, [rdi+3]
 7|   ret
```

godbolt.org/z/Yxbb6q

```
# g92 -O0
  | f(int):
 1|   push rbp
 1|   mov rbp, rsp
 1|   mov DWORD PTR [rbp-4], edi
 2|   mov eax, DWORD PTR [rbp-4]
 2|   add eax, 1
 3|   pop rbp
 3|   ret
  | g(int):
 5|   push rbp
 5|   mov rbp, rsp
 5|   sub rsp, 8
 5|   mov DWORD PTR [rbp-4], edi
 6|   mov eax, DWORD PTR [rbp-4]
 6|   add eax, 2
 6|   mov edi, eax
 6|   call f(int)
 7|   leave
 7|   ret
```

godbolt.org/z/87GK4q

Reading assembly for fun and profit

```
1  void side_effect();
2
3  int f(int x) {
4      auto a = x;
5      side_effect();
6      return a - x;
7  }
```

godbolt.org/z/5xq5n5

```
# g92 -O0
  | f(int):
 3|   push rbp
 3|   mov rbp, rsp
 3|   sub rsp, 32
 3|   mov DWORD PTR [rbp-20], edi
 4|   mov eax, DWORD PTR [rbp-20]
 4|   mov DWORD PTR [rbp-4], eax
 5|   call side_effect()
 6|   mov eax, DWORD PTR [rbp-4]
 6|   sub eax, DWORD PTR [rbp-20]
 7|   leave
 7|   ret
```

godbolt.org/z/5xq5n5

Implicit Costs of using `const&`

```
1  void side_effect();
2
3  int f(int x) {
4      auto a = x;
5      side_effect();
6      return a - x;
7  }
```

godbolt.org/z/5xq5n5

```
1  void side_effect();
2
3  int f(const int& x) {
4      auto a = x;
5      side_effect();
6      return a - x;
7  }
```

godbolt.org/z/333ME7

Implicit Costs of using `const&`

```
# g92 -O0
  | f(int):
 3|   push rbp
 3|   mov rbp, rsp
 3|   sub rsp, 32
 3|   mov DWORD PTR [rbp-20], edi
 4|   mov eax, DWORD PTR [rbp-20]
 4|   mov DWORD PTR [rbp-4], eax
 5|   call side_effect()
 6|   mov eax, DWORD PTR [rbp-4]
 6|   sub eax, DWORD PTR [rbp-20]
 7|   leave
 7|   ret
```

godbolt.org/z/5xq5n5

```
# g92 -O0
  | f(int const&):
 3|   push rbp
 3|   mov rbp, rsp
 3|   sub rsp, 32
 3|   mov QWORD PTR [rbp-24], rdi
 4|   mov rax, QWORD PTR [rbp-24]
 4|   mov eax, DWORD PTR [rax]
 4|   mov DWORD PTR [rbp-4], eax
 5|   call side_effect()
 6|   mov rax, QWORD PTR [rbp-24]
 6|   mov eax, DWORD PTR [rax]
 6|   mov edx, DWORD PTR [rbp-4]
 6|   sub edx, eax
 6|   mov eax, edx
 7|   leave
 7|   ret
```

godbolt.org/z/333ME7

Implicit Costs of using `const&`

```
# g92 -O3
  | f(int):
 3|   sub rsp, 8
 5|   call side_effect()
 7|   xor eax, eax
 7|   add rsp, 8
 7|   ret
```

godbolt.org/z/od8v6e

NB #1: adjusting `rsp` in function prologue necessary when function is not a leaf function since callee have to know where to start saving variables on stack. (Adjusting `rsp` can be ommitted in leaf functions.)

```
# g92 -O3
  | f(int const&):
 3|   push rbp
 3|   push rbx
 3|   mov rbx, rdi
 3|   sub rsp, 8
 4|   mov ebp, DWORD PTR [rdi]
 5|   call side_effect()
 6|   mov eax, ebp
 6|   sub eax, DWORD PTR [rbx]
 7|   add rsp, 8
 7|   pop rbx
 7|   pop rbp
 7|   ret
```

godbolt.org/z/cr8f9b

Implicit Costs of using `const&`

```
# g92 -O3
  | f(int):
 3|   sub rsp, 8
 5|   call side_effect()
 7|   xor eax, eax
 7|   add rsp, 8
 7|   ret
```

godbolt.org/z/od8v6e

NB #2: Offset `x` in `sub rsp, x` is objective of optimizations such as alignment: ABI requires stack to be aligned to 16 bytes.

```
# g92 -O3
  | f(int const&):
 3|   push rbp
 3|   push rbx
 3|   mov rbx, rdi
 3|   sub rsp, 8
 4|   mov ebp, DWORD PTR [rdi]
 5|   call side_effect()
 6|   mov eax, ebp
 6|   sub eax, DWORD PTR [rbx]
 7|   add rsp, 8
 7|   pop rbx
 7|   pop rbp
 7|   ret
```

godbolt.org/z/cr8f9b

Implicit Costs of using `const&`

```
# g92 -O3
  | f(int):
 3|   sub rsp, 8
 5|   call side_effect()
 7|   xor eax, eax
 7|   add rsp, 8
 7|   ret
```

godbolt.org/z/od8v6e

NB #1: adjusting `rsp` in function prologue necessary when function is not a leaf function since callee have to know where to start saving variables on stack. (Adjusting `rsp` can be ommitted in leaf functions.)

```
# g92 -O3
  | f(int const&):
 3|   push rbp
 3|   push rbx
 3|   mov rbx, rdi
 3|   sub rsp, 8
 4|   mov ebp, DWORD PTR [rdi]
 5|   call side_effect()
 6|   mov eax, ebp
 6|   sub eax, DWORD PTR [rbx]
 7|   add rsp, 8
 7|   pop rbx
 7|   pop rbp
 7|   ret
```

godbolt.org/z/cr8f9b

Implicit Costs of using `const&`

```
# g92 -O3
  | f(int):
 3|   sub rsp, 8
 5|   call side_effect()
 7|   xor eax, eax
 7|   add rsp, 8
 7|   ret
```

godbolt.org/z/od8v6e

NB #2: Offset `x` in `sub rsp, x` is objective of optimizations such as alignment: ABI requires stack to be aligned to 16 bytes.

```
# g92 -O3
  | f(int const&):
 3|   push rbp
 3|   push rbx
 3|   mov rbx, rdi
 3|   sub rsp, 8
 4|   mov ebp, DWORD PTR [rdi]
 5|   call side_effect()
 6|   mov eax, ebp
 6|   sub eax, DWORD PTR [rbx]
 7|   add rsp, 8
 7|   pop rbx
 7|   pop rbp
 7|   ret
```

godbolt.org/z/cr8f9b

Implicit Costs of using `const&`

```
1  #include <string>
2  #include <string_view>
3
4  auto get_size(const std::string& s) {
5          return s.size();
6  }
7
8  auto get_size(std::string_view sv) {
9      return sv.size();
10 }
```

godbolt.org/z/br9Mfz

```
# clang900 -O3 -std=c++2a -stdlib=libc++
  | get_size(std::string const&):
xx|   movzx eax, byte ptr [rdi]
xx|   test al, 1
xx|   je .LBB0_1
 0|   mov rax, qword ptr [rdi + 8]
 5|   ret
  | .LBB0_1:
 0|   shr rax
 5|   ret
  | get_size(std::string_view):
 8|   mov rax, rsi
 9|   ret
  |
```

godbolt.org/z/br9Mfz

Even though we *only* pass a reference, we pay the cost of the complex object `std::string` (i.e., first bit is tested for short string optimization)

↪ prefer views such as `std::string_view` or `std::span`

Implicit Costs of using `const&`

```cpp
#include <string>
#include <string_view>

auto get_size(const std::string& s) {
        return s.size();
}

auto get_size(std::string_view sv) {
    return sv.size();
}
```

godbolt.org/z/br9Mfz

```
# clang900 -O3 -std=c++2a -stdlib=libc++
   | get_size(std::string const&):
xx|   movzx eax, byte ptr [rdi]
xx|   test al, 1
xx|   je .LBB0_1
 0|   mov rax, qword ptr [rdi + 8]
 5|   ret
   | .LBB0_1:
 0|   shr rax
 5|   ret
   | get_size(std::string_view):
 8|   mov rax, rsi
 9|   ret
   |
```

godbolt.org/z/br9Mfz

*Confession: switching to `libstdc++` resolves this issue here

Will it compile?

```cpp
#include <array>
#include <span>

int main() {
    constexpr std::array x{
        4, 8, 15, 16, 23, 42
    };
    constexpr std::span x_view{x};
}
```

godbolt.org/z/66exs6

```cpp
template <std::size_t N>
constexpr span(const std::array<value_type, N>& arr) noexcept;
```

cppreference.com/w/cpp/container/span/span

Will it compile?

```cpp
#include <array>
#include <span>

int main() {
    constexpr std::array x{
        4, 8, 15, 16, 23, 42
    };
    constexpr std::span x_view{x};
}
```

godbolt.org/z/66exs6

No!

- → Constructor takes by reference
- → References to automatic storage objects are not constant expressions!
- → Solutions?

```cpp
template <std::size_t N>
constexpr span(const std::array<value_type, N>& arr) noexcept;
```

cppreference.com/w/cpp/container/span/span

Will it compile?

```cpp
#include <array>
#include <span>

int main() {
    constexpr static std::array x{
        4, 8, 15, 16, 23, 42
    };
    constexpr std::span x_view{x};
}
```

godbolt.org/z/Ga5Ysv

Nota bene …

this will work though, since reference / pointer does not *escape* constant expression …

```
#include <array>
#include <span>

constexpr auto f() {
    std::array x{4, 8, 15, 16, 23, 42};
    std::span x_view{x};
    return 0;
}

int main() {
    static_assert(f() == 0);
}
```

godbolt.org/z/rso3na

PART II

Dangling References

Will it compile?

```
struct S {
    int x;
};

auto f() {
    S s{.x = 42};
    return s;
}

int main() {
    S& s = f();
    return s.x;
}
```

godbolt.org/z/x4rWKj

Will it invoke undefined behavior?

```
1  struct S {
2      int x;
3  };
4
5  auto f() {
6      S s{.x = 42};
7      return s;
8  }
9
10 int main() {
11     const S& s = f();
12     return s.x;
13 }
```

godbolt.org/z/avGMPa

…binding a reference to a temporary???

Temporary object lifetime extension

```
1  struct S {
2      int x;
3  };
4
5  auto f() {
6      S s{.x = 42};
7      return s;
8  }
9
10 int main() {
11     const S& s = f();
12     return s.x;
13 }
```

godbolt.org/z/avGMPa

cppreference.com: “The lifetime of a temporary object may be extended by binding to a const lvalue reference or to an rvalue reference (since C++11).”

Q: What is the output of the program?

```
1  #include <iostream>
2
3  template <char id> struct Log {
4      Log() { std::cout << id << 1; }
5      virtual ~Log() { std::cout << id << 2; }
6  };
7  struct A: Log<'a'> {
8      int x;
9      A(int x): x(x) {};
10 };
11 struct B: Log<'b'> {
12     const A& a;
13     B(const A& a): a(a) {}
14 };
15
16 int main() {
17     const B& b = B{A{42}};
18     std::cout << 'x';
19     return b.a.x;
20 }
```

godbolt.org/z/hcods4

A: a1b1a2xb2

```
1  #include <iostream>
2
3  template <char id> struct Log {
4      Log() { std::cout << id << 1; }
5      virtual ~Log() { std::cout << id << 2; }
6  };
7  struct A: Log<'a'> {
8      int x;
9      A(int x): x(x) {};
10 };
11 struct B: Log<'b'> {
12     const A& a;
13     B(const A& a): a(a) {}
14 };
15
16 int main() {
17     const B& b = B{A{42}};
18     std::cout << 'x';
19     return b.a.x;
20 }
```

godbolt.org/z/hcods4

Dangling reference!!!

- → lifetime extension only for result of the temporary expression, **not any sub-expression**
- → use address sanitizer!

contrived?

Reference lifetime extension

(derived from abseil.io: Tip of the Week #107: “Reference Lifetime Extension”)

```
1  std::vector<std::string_view> explode(const std::string& s);
2
3  for (std::string_view s: explode(str_cat("oo", "ps"))) { // WRONG!
4      [...]
```


Q: What is the output of the program?

```
1  #include <vector>
2
3  int main() {
4      std::vector<int> v;
5      v.push_back(1);
6      auto& x = v[0];
7      v.push_back(2);
8      return x;
9  }
```

godbolt.org/z/M6bx1Y

Q: What is the output of the program?

```
1  #include <vector>
2
3  int main() {
4      std::vector<int> v;
5      v.push_back(1);
6      auto& x = v[0];
7      v.push_back(2);
8      return x;
9  }
```

godbolt.org/z/M6bx1Y

Dangling reference!!!

- → `std::vector` needs to reallocate all the space the second time an element is pushed
- → use address sanitizer!

`std::move` in the wild

Moving `std::string`

(derived from CppCon 2019: *Ben Deane* "Everyday Efficiency: In-Place Construction (Back to Basics?)")

```
1  static void cp_small_str(benchmark::State& state) {
2      for (auto _ : state) {
3          std::string original("small");
4          benchmark::DoNotOptimize(original);
5          std::string copied = original;
6          benchmark::DoNotOptimize(copied);
7      }
8  }
9  BENCHMARK(cp_small_str);
```

```
1  static void mv_small_str(benchmark::State& state) {
2      for (auto _ : state) {
3          std::string original("small");
4          benchmark::DoNotOptimize(original);
5          std::string moved = std::move(original);
6          benchmark::DoNotOptimize(moved);
7      }
8  }
9  BENCHMARK(mv_small_str);
```


Moving `std::string`

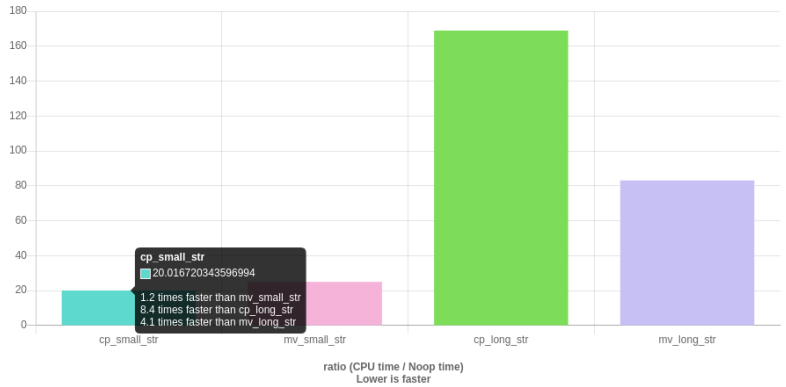
(derived from CppCon 2019: *Ben Deane* "Everyday Efficiency: In-Place Construction (Back to Basics?)")

```
1  static void cp_long_str(benchmark::State& state) {
2      for (auto _ : state) {
3          std::string original("this is too long for short string optimization");
4          benchmark::DoNotOptimize(original);
5          std::string copied = original;
6          benchmark::DoNotOptimize(copied);
7      }
8  }
9  BENCHMARK(cp_long_str);
```

```
1  static void mv_long_str(benchmark::State& state) {
2      for (auto _ : state) {
3          std::string original("this is too long for short string optimization");
4          benchmark::DoNotOptimize(original);
5          std::string moved = std::move(original);
6          benchmark::DoNotOptimize(moved);
7      }
8  }
9  BENCHMARK(mv_long_str);
```


Moving `std::string`

Quick Bench result



Quick Bench: `tinyurl.com/yybmdngv`

Moving `std::string`

Copy small `std::string`

1. copy stack allocated data

Move small `std::string`

1. copy stack allocated data
2. set string length of moved string to zero

↪ moving is not necessarily better than copying!

Moving `std::map`

Did they forget to mark the move ctor `noexcept`?

```
// since C++11
std::map(const std::map&&)

// until C++17
std::map& operator=(std::map&&)

// since C++17
std::map& operator=(std::map&&) noexcept
```


Moving `std::map`

Did they forget to mark the move ctor `noexcept`? No!

```
// since C++11
std::map(const std::map&&)

// until C++17
std::map& operator=(std::map&&)

// since C++17
std::map& operator=(std::map&&) noexcept
```

- → Move ctor needs to allocate new sentinel node, because moved from container must still be a valid container (albeit in an unspecified state)
- → Move assignment can swap, thus no need to allocate

↪ move ctor of `std::map` allocates heap space!

(Billy O'Neal: `twitter.com/MalwareMinigun/status/1165310509022736384`)

Moving `std::map`

```
1  static void rvo(benchmark::State& state) {
2      for (auto _ : state) {
3          auto m = []() -> std::map<int, int> {
4              std::map<int, int> m{{0, 42}};
5              return m;
6          }();
7          benchmark::DoNotOptimize(m);
8      }
9  }
10 BENCHMARK(rvo);
```

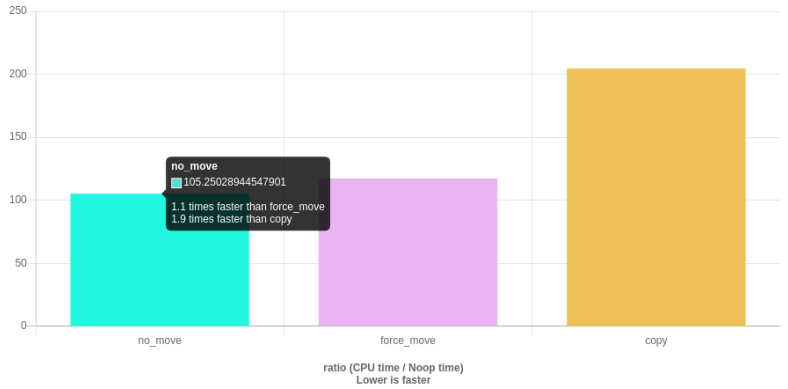
```
1  static void fmove(benchmark::State& state) {
2      for (auto _ : state) {
3          auto m = []() -> std::map<int, int> {
4              std::map<int, int> m{{0, 42}};
5              return std::move(m);
6          }();
7          benchmark::DoNotOptimize(m);
8      }
9  }
10 BENCHMARK(fmove);
```

Moving `std::map`

```
1  static void copy(benchmark::State& state) {
2      for (auto _ : state) {
3          std::map<int, int> m{{0, 42}};
4          benchmark::DoNotOptimize(m);
5          auto m2 = m;
6          benchmark::DoNotOptimize(m2);
7      }
8  }
9  BENCHMARK(copy);
```


Moving `std::map`

Quick Bench result



Quick Bench: `tinyurl.com/y57egvjp`

Why?

Does this code bother anyone?

```
1  #include <cstddef>
2  #include <type_traits>
3  #include <utility>
4
5  template <typename T>
6  struct Data final {
7      T *data;
8      explicit Data(const std::size_t size): data(new T[size]) {}
9      ~Data() { delete [] data; }
10 };
11
12 auto init() {
13     Data<int> d(3);
14     d.data[0] = 1; d.data[1] = 2; d.data[2] = 3;
15     return d;
16 }
17
18 int main() { return init().data[2]; }
```

godbolt.org/z/j19Pbq

Interlude

What happens on `return`?

Will it compile?

```
1  struct A {
2      int x;
3      A(int x, int y = 0) : x(x + y) {}
4  };
5
6  struct B {
7      int x;
8      explicit B(int x, int y = 0) : x(x + y) {}
9  };
10
11 template <typename T>
12 T init() { return 42; }
13
14 int main() {
15   auto x = init<A>().x;
16   auto y = init<B>().x;
17 }
```

godbolt.org/z/vYab6f

Implicit conversion to the function return type

Implicit conversion

- …if ctor is **not** marked `explicit`
- Examples:
  - `std::optional(T&&)`
  - `std::string(const char*)`

```
1  #include <optional>
2  #include <string>
3
4  std::optional<int> f() {
5      return 42;
6  }
7
8  std::string g() {
9      return "foo";
10 }
```

godbolt.org/z/bh4svz

Q: What is the output of the program?

Compiler flags: `-std=c++14 -fno-elide-constructors`

```
1  #include <iostream>
2
3  struct S {
4      S() { std::cout << 'a'; }
5      S(const S&) { std::cout << 'b'; }
6      S(const S&&) { std::cout << 'c'; }
7      S& operator=(const S&) { std::cout << 'd'; return *this; }
8      S& operator=(const S&&) { std::cout << 'e'; return *this; }
9  };
10
11 S f() { return S{}; }
12
13 int main() {
14     S s(S{}); std::cout << ", ";
15     auto s1 = f(); std::cout << ", ";
16     auto s2{f()};
17 }
```

godbolt.org/z/xxb9xe

Q: What is the output of the program?

Compiler flags: `-std=c++17`

```
1  #include <iostream>
2
3  struct S {
4      S() { std::cout << 'a'; }
5      S(const S&) { std::cout << 'b'; }
6      S(const S&&) { std::cout << 'c'; }
7      S& operator=(const S&) { std::cout << 'd'; return *this; }
8      S& operator=(const S&&) { std::cout << 'e'; return *this; }
9  };
10
11 S f() { return S{}; }
12
13 int main() {
14     S s(S{}); std::cout << ", ";
15     auto s1 = f(); std::cout << ", ";
16     auto s2{f()};
17 }
```

godbolt.org/z/7oPa4Y

Why?

Ben Deane: “Perhaps the most important optimization the compiler does”

Copy Elision

```
1  struct S {
2      S() = default;
3      S(const S&) = delete;
4      S& operator=(const S&) = delete;
5  };
6
7  S f() { return S{}; }
8
9  int main() {
10     S s(S{});
11     auto s1 = f();
12     auto s2{f()};
13 }
```

godbolt.org/z/ThqjzP

Mandatory elision of copy/move operations since C++17):

- → Return statement: when operand is a `prvalue` of same class type as return type
- → Initialization of a variable: when initializer expression is a `prvalue` of same class type as the variable type

…even if the copy/move constructor and the destructor has observable side-effects!

Rule of thumb: avoid naming return values

RVO in Depth

C++ Objects in Assembly

```
1  struct S final {
2      int a, b, c;
3
4      S(int a, int b, int c) noexcept:
5          a(a), b(b), c(c) {}
6
7      ~S() noexcept {}
8
9      auto sum() noexcept {
10         return a + b + c;
11     }
12 };
13
14 int main() {
15     S s(1, 2, 3);
16     return s.sum();
17 }
```

godbolt.org/z/4oWTr6

```
   | main:
14|   push rbp
14|   mov rbp, rsp
14|   sub rsp, 16
14|   mov dword ptr [rbp - 4], 0
15|   lea rdi, [rbp - 16]
15|   mov esi, 1
15|   mov edx, 2
15|   mov ecx, 3
15|   call S::S(int, int, int)
16|   lea rdi, [rbp - 16]
16|   call S::sum()
16|   mov dword ptr [rbp - 4], eax
17|   lea rdi, [rbp - 16]
17|   call S::~S()
17|   mov eax, dword ptr [rbp - 4]
17|   add rsp, 16
17|   pop rbp
17|   ret
```


C++ Objects in Assembly

```
1  struct S final {
2      int a, b, c;
3
4      S(int a, int b, int c) noexcept:
5          a(a), b(b), c(c) {}
6
7      ~S() noexcept {}
8
9      auto sum() noexcept {
10         return a + b + c;
11     }
12 };
13
14 int main() {
15     S s(1, 2, 3);
16     return s.sum();
17 }
```

godbolt.org/z/4oWTr6

```
   | S::S(int, int, int):
 5|   push rbp
 5|   mov rbp, rsp
 5|   mov qword ptr [rbp - 8], rdi
 5|   mov dword ptr [rbp - 12], esi
 5|   mov dword ptr [rbp - 16], edx
 5|   mov dword ptr [rbp - 20], ecx
 5|   mov rax, qword ptr [rbp - 8]
 5|   mov ecx, dword ptr [rbp - 12]
 5|   mov dword ptr [rax], ecx
 5|   mov ecx, dword ptr [rbp - 16]
 5|   mov dword ptr [rax + 4], ecx
 5|   mov ecx, dword ptr [rbp - 20]
 5|   mov dword ptr [rax + 8], ecx
 5|   pop rbp
 5|   ret
```


C++ Objects in Assembly

```
1  struct S final {
2      int a, b, c;
3
4      S(int a, int b, int c) noexcept:
5          a(a), b(b), c(c) {}
6
7      ~S() noexcept {}
8
9      auto sum() noexcept {
10         return a + b + c;
11     }
12 };
13
14 int main() {
15     S s(1, 2, 3);
16     return s.sum();
17 }
```

godbolt.org/z/4oWTr6

```
   | S::sum():
 9|   push rbp
 9|   mov rbp, rsp
 9|   mov qword ptr [rbp - 8], rdi
 9|   mov rax, qword ptr [rbp - 8]
10|   mov ecx, dword ptr [rax]
10|   add ecx, dword ptr [rax + 4]
10|   add ecx, dword ptr [rax + 8]
10|   mov eax, ecx
10|   pop rbp
10|   ret
```

RVO in Assembly

```
1  struct S final {
2      int x;
3      explicit S(int x) noexcept;
4      S(const S&) noexcept;
5      S(S&&) noexcept;
6      ~S() noexcept;
7  };
8
9  S f() {
10     return S{42};
11 }
12
13 auto g() {
14     auto s = f();
15     return s.x;
16 }
```

godbolt.org/z/z86r3d

```
# g92 -fno-elide-constructors
  | g():
  |   [...]
14|   lea rax, [rbp-20]
14|   mov rdi, rax
14|   call f()
14|   lea rdx, [rbp-20]
14|   lea rax, [rbp-24]
14|   mov rsi, rdx
14|   mov rdi, rax
14|   call S::S(S&&)
14|   lea rax, [rbp-20]
14|   mov rdi, rax
14|   call S::~S()
15|   mov ebx, DWORD PTR [rbp-24]
14|   lea rax, [rbp-24]
14|   mov rdi, rax
14|   call S::~S()
15|   mov eax, ebx
  |   [...]
```


RVO in Assembly

```
1  struct S final {
2      int x;
3      explicit S(int x) noexcept;
4      S(const S&) noexcept;
5      S(S&&) noexcept;
6      ~S() noexcept;
7  };
8
9  S f() {
10     return S{42};
11 }
12
13 auto g() {
14     auto s = f();
15     return s.x;
16 }
```

godbolt.org/z/z86r3d

```
# g92 -fno-elide-constructors
  | f():
  |   [...]
 9|   mov QWORD PTR [rbp-24], rdi
10|   lea rax, [rbp-4]
10|   mov esi, 42
10|   mov rdi, rax
10|   call S::S(int)
10|   lea rdx, [rbp-4]
10|   mov rax, QWORD PTR [rbp-24]
10|   mov rsi, rdx
10|   mov rdi, rax
10|   call S::S(S&&)
10|   lea rax, [rbp-4]
10|   mov rdi, rax
10|   call S::~S()
  |   [...]
```


RVO in Assembly

```
# g92 -fno-elide-constructors
f():
  [...]
  mov QWORD PTR [rbp-24], rdi
  lea rax, [rbp-4]
  mov esi, 42
  mov rdi, rax
  call S::S(int)
  lea rdx, [rbp-4]
  mov rax, QWORD PTR [rbp-24]
  mov rsi, rdx
  mov rdi, rax
  call S::S(S&&)
  lea rax, [rbp-4]
  mov rdi, rax
  call S::~S()
  nop
  mov rax, QWORD PTR [rbp-24]
  leave
  ret
```

```
# g92
f():
  [...]
  mov QWORD PTR [rbp-8], rdi
  mov rax, QWORD PTR [rbp-8]
  mov esi, 42
  mov rdi, rax
  call S::S(int)
  mov rax, QWORD PTR [rbp-8]
  leave
  ret
```


RVO in Assembly

```
# g92 -fno-elide-constructors
g():
  [...]
  lea rax, [rbp-20]
  mov rdi, rax
  call f()
  lea rdx, [rbp-20]
  lea rax, [rbp-24]
  mov rsi, rdx
  mov rdi, rax
  call S::S(S&&)
  lea rax, [rbp-20]
  mov rdi, rax
  call S::~S()
  mov ebx, DWORD PTR [rbp-24]
  lea rax, [rbp-24]
  mov rdi, rax
  call S::~S()
  mov eax, ebx
  [...]
```

```
# g92
g():
  [...]
  lea rax, [rbp-20]
  mov rdi, rax
  call f()
  mov ebx, DWORD PTR [rbp-20]
  lea rax, [rbp-20]
  mov rdi, rax
  call S::~S()
  mov eax, ebx
```

(N)RVO or no (N)RVO?

```
#include <utility>

struct S final {
    S() noexcept;
    S(const S&) noexcept;
    S(S&&) noexcept;
    ~S() noexcept;
};

S f1() { S s; return s; }
S f2() { S s; return std::move(s); }
S f3() { const S s; return s; }
S f4() { const S s; return std::move(s); }
```

godbolt.org/z/6Es7Ys

- f1: ???
- f2: ???
- f3: ???
- f4: ???

(N)RVO or no (N)RVO?

```
struct S {
    S() noexcept;
    S(const S&) noexcept;
    S(const S&&) noexcept;
    ~S() noexcept;
};

S f1(S s) { return s; }
S f2(S& s) { return s; }
S f3(const S& s) { return s; }
```

godbolt.org/z/7sf6jY

- f1: ???
- f2: ???
- f3: ???

(N)RVO or no (N)RVO?

```
struct S final {
    S() noexcept;
    S(const S&) noexcept;
    S(S&&) noexcept;
    ~S() noexcept;
};

S f() {
    S s;
    auto& t = s;
    return t;
}
```

godbolt.org/z/a6hrE1

(N)RVO or no (N)RVO?

```
struct S {
    S(int) noexcept;
    S(const S&) noexcept;
    S(const S&&) noexcept;
    ~S() noexcept;
};

S f1(bool x) { return x ? S{1} : S{2}; }
S f2(bool x) { S s{1}; return x ? s : S{2}; }
```

godbolt.org/z/TEcq1o

- f1: ???
- f2: ???

(N)RVO or no (N)RVO?

```
1  #include <utility>
2
3  struct S final {
4      S() noexcept;
5      S(const S&) noexcept;
6      S(S&&) noexcept;
7      ~S() noexcept;
8  };
9  auto f() { return std::pair<S, S>{}; }
10 S g1() { auto [s1, s2] = f(); return s1; }
11 S g2() { auto&& [s1, s2] = f(); return s1; }
12 S g3() { auto [s1, s2] = f(); return std::move(s1); }
13 S g4() { auto&& [s1, s2] = f(); return std::move(s1); }
```

godbolt.org/z/v5ro3q

- g1: ???
- g2: ???
- g3: ???
- g4: ???

(N)RVO or no (N)RVO?

(derived from CppCon 2019: *Jason Turner* "Great C++ is_trivial")

No (N)RVO in any of these examples!

```
1 S g1() { auto [s1, s2] = f(); return s1; } // copy
2 S g2() { auto&& [s1, s2] = f(); return s1; } // copy: no implicit move yet (?)
3 S g3() { auto [s1, s2] = f(); return std::move(s1); } // move
4 S g4() { auto&& [s1, s2] = f(); return std::move(s1); } // move
```

…`return std::move` is not always bad

Why? Structured bindings:

- Creation of temporary object `e`
- Like a reference: structured binding is an alias into `e`

Implicit move

`return std::move` is not yet necessarily a code smell

(use `-Wpessimizing-move`)

Automatic move from local variables and parameters if:

- return expression names a variable whose type is either
  - an object type or (since C++11)
  - an rvalue reference to object type (since C++20*)
- …and that variable is declared
  - in the body or
  - as a parameter of
- …the innermost enclosing function or lambda expression

* P1825R0 (not yet implemented in GCC or Clang: cppreference.com/w/cpp/compiler_support)

Perfect Backwarding

Forwarding Values and Preserving Value Category

(derived from CppCon 2018: *Hayun Ezra Chung* "Forwarding Values... and Backwarding Them Too?")

```cpp
#include <iostream>
#include <memory>

struct Resource {};

struct Target {
    Target(const Resource&) { std::cout << 'a'; }
    Target(Resource&&) { std::cout << 'b'; }
};

auto make_target(??? resource) {
    return std::make_unique<Target>(???);
}

int main() {
    Resource resource;
    make_target(resource);            // should print 'a'
    make_target(Resource(resource));  // should print 'b'
    make_target(std::move(resource)); // should print 'b'
}
```

godbolt.org/z/WMPGGq

Forwarding Values and Preserving Value Category

(derived from CppCon 2018: *Hayun Ezra Chung* "Forwarding Values... and Backwarding Them Too?")

```cpp
#include <iostream>
#include <memory>

struct Resource {};

struct Target {
    Target(const Resource&) { std::cout << 'a'; }
    Target(Resource&&) { std::cout << 'b'; }
};

template <typename T> auto make_target(T&& resource) {
    return std::make_unique<Target>(std::forward<T>(resource));
}

int main() {
    Resource resource;
    make_target(resource);            //  lvalue: T = Resource&
    make_target(Resource(resource));  // prvalue: T = Resource
    make_target(std::move(resource)); //  xvalue: T = Resource
}
```

godbolt.org/z/nbd6P4

Forwarding Values and Preserving Value Category

(derived from CppCon 2018: *Hayun Ezra Chung* "Forwarding Values... and Backwarding Them Too?")

```cpp
#include <iostream>
#include <memory>

struct Resource {};

struct Target {
    Target(const Resource&) { std::cout << 'a'; }
    Target(Resource&&) { std::cout << 'b'; }
};

auto make_target(auto&& resource) {
    return std::make_unique<Target>(std::forward<decltype(resource)>(resource));
}

int main() {
    Resource resource;
    make_target(resource);
    make_target(Resource(resource));
    make_target(std::move(resource));
}
```

godbolt.org/z/znq1K4

Backwarding Values and Preserving Value Category

```cpp
#include <iostream>

struct Resource {};
struct ResourceManager {
    Resource resource;

    decltype(auto) visit(auto visitor) { return visitor(resource); }
};
struct Target {
    Target(const Resource&) { std::cout << 'a'; }
    Target(Resource&&) { std::cout << 'b'; }
};

int main() {
    ResourceManager rm;
    Target(rm.visit([](Resource& r) -> Resource& { return r; }));
    Target(rm.visit([](Resource& r) -> Resource { return r; }));
    Target(rm.visit([](Resource& r) -> Resource&& { return std::move(r); }));
}
```

godbolt.org/z/f56vaP

Backwarding Values and Preserving Value Category

```cpp
#include <iostream>

struct Resource {};
struct ResourceManager {
    Resource resource;

    // what if we want to do sth. with the result before returning?
    decltype(auto) visit(auto visitor) { return visitor(resource); }
};
struct Target {
    Target(const Resource&) { std::cout << 'a'; }
    Target(Resource&&) { std::cout << 'b'; }
};

int main() {
    ResourceManager rm;
    Target(rm.visit([](Resource& r) -> Resource& { return r; }));
    Target(rm.visit([](Resource& r) -> Resource { return r; }));
    Target(rm.visit([](Resource& r) -> Resource&& { return std::move(r); }));
}
```

godbolt.org/z/35zbYW

Backwarding Values and Preserving Value Category

Q: Why is this a bad idea?

```cpp
auto&& visit(auto visitor) {
    auto&& result = visitor(resource);
    return result;
}
```


Backwarding Values and Preserving Value Category

Q: Why is this a bad idea?

```cpp
auto&& visit(auto visitor) {
    auto&& result = visitor(resource);
    return result;
}
```

A: Dangling reference for

```cpp
visit([](Resource& r) -> Resource { return r; }));
```

`auto&&` is always a reference!

Backwarding Values and Preserving Value Category

```cpp
#include <iostream>

struct Resource {};
struct ResourceManager {
    Resource resource;

    ??? visit(auto visitor) {
        ??? result = visitor(resource);
        return ???;
    }
};
struct Target {
    Target(const Resource&) { std::cout << 'a'; }
    Target(Resource&&) { std::cout << 'b'; }
};

int main() {
    ResourceManager rm;
    Target(rm.visit([](Resource& r) -> Resource { return r; }));
}
```

godbolt.org/z/d1WeqT

Backwarding Values and Preserving Value Category

```cpp
#include <iostream>

struct Resource {};
struct ResourceManager {
    Resource resource;

    Resource visit(auto visitor) {
        Resource result = visitor(resource);
        return result;
    }
};
struct Target {
    Target(const Resource&) { std::cout << 'a'; }
    Target(Resource&&) { std::cout << 'b'; }
};

int main() {
    ResourceManager rm;
    Target(rm.visit([](Resource& r) -> Resource { return r; }));
}
```

godbolt.org/z/9joq3h

Backwarding Values and Preserving Value Category

```cpp
#include <iostream>

struct Resource {};
struct ResourceManager {
    Resource resource;

    ??? visit(auto visitor) {
        ??? result = visitor(resource);
        return ???;
    }
};
struct Target {
    Target(const Resource&) { std::cout << 'a'; }
    Target(Resource&&) { std::cout << 'b'; }
};

int main() {
    ResourceManager rm;
    Target(rm.visit([](Resource& r) -> Resource& { return r; }));
}
```

godbolt.org/z/MzGGqc

Backwarding Values and Preserving Value Category

```cpp
#include <iostream>

struct Resource {};
struct ResourceManager {
    Resource resource;

    Resource& visit(auto visitor) {
        Resource& result = visitor(resource);
        return result;
    }
};
struct Target {
    Target(const Resource&) { std::cout << 'a'; }
    Target(Resource&&) { std::cout << 'b'; }
};

int main() {
    ResourceManager rm;
    Target(rm.visit([](Resource& r) -> Resource& { return r; }));
}
```

godbolt.org/z/Y6437o

Backwarding Values and Preserving Value Category

```cpp
#include <iostream>

struct Resource {};
struct ResourceManager {
    Resource resource;

    ??? visit(auto visitor) {
        ??? result = visitor(resource);
        return ???;
    }
};
struct Target {
    Target(const Resource&) { std::cout << 'a'; };
    Target(Resource&&) { std::cout << 'b'; }
};

int main() {
    ResourceManager rm;
    Target(rm.visit([](Resource& r) -> Resource&& { return std::move(r); }));
}
```

godbolt.org/z/rj4xqz

Backwarding Values and Preserving Value Category

```cpp
#include <iostream>

struct Resource {};
struct ResourceManager {
    Resource resource;

    Resource&& visit(auto visitor) {
        Resource&& result = visitor(resource);
        return result;
    }
};
struct Target {
    Target(const Resource&) { std::cout << 'a'; };
    Target(Resource&&) { std::cout << 'b'; }
};

int main() {
    ResourceManager rm;
    Target(rm.visit([](Resource& r) -> Resource&& { return std::move(r); }));
}
```

godbolt.org/z/a7rq34

Backwarding Values and Preserving Value Category

error: cannot bind rvalue reference of type “Resource&&” to lvalue of type “Resource”

```cpp
#include <iostream>

struct Resource {};
struct ResourceManager {
    Resource resource;

    Resource&& visit(auto visitor) {
        Resource&& result = visitor(resource);
        return result;
    }
};
struct Target {
    Target(const Resource&) { std::cout << 'a'; };
    Target(Resource&&) { std::cout << 'b'; }
};

int main() {
    ResourceManager rm;
    Target(rm.visit([](Resource& r) -> Resource&& { return std::move(r); }));
}
```

godbolt.org/z/a7rq34

Backwarding Values and Preserving Value Category

```cpp
#include <iostream>

struct Resource {};
struct ResourceManager {
    Resource resource;

    Resource&& visit(auto visitor) {
        Resource&& result = visitor(resource);
        return std::move(result); // static_cast<Resource&&>(result)
    }
};
struct Target {
    Target(const Resource&) { std::cout << 'a'; };
    Target(Resource&&) { std::cout << 'b'; }
};

int main() {
    ResourceManager rm;
    Target(rm.visit([](Resource& r) -> Resource&& { return std::move(r); }));
}
```

godbolt.org/z/T91Tbq

Backwarding Values and Preserving Value Categroy

How do we fuse these implementations?

```cpp
Resource visit(auto visitor) {
    Resource result = visitor(resource);
    return result;
}

Resource& visit(auto visitor) {
    Resource& result = visitor(resource);
    return result;
}

Resource&& visit(auto visitor) {
    Resource&& result = visitor(resource);
    return static_cast<Resource&&>(result);
}
```

```cpp
Target(rm.visit([](Resource& r) -> Resource { return r; }));
Target(rm.visit([](Resource& r) -> Resource& { return r; }));
Target(rm.visit([](Resource& r) -> Resource&& { return std::move(r); }));
```

Backwarding Values and Preserving Value Categroy

How do we fuse these implementations?

```
1  Resource visit(auto visitor) {
2      Resource result = visitor(resource);
3      return result;
4  }
5
6  Resource& visit(auto visitor) {
7      Resource& result = visitor(resource);
8      return result;
9  }
10
11 Resource&& visit(auto visitor) {
12     Resource&& result = visitor(resource);
13     return static_cast<Resource&&>(result);
14 }
```

```
1  decltype(auto) visit(auto visitor) {
2      decltype(auto) result = visitor(resource);
3      return static_cast<decltype(result)>(result);
4  }
```


(derived from CppCon 2018: *Hayun Ezra Chung* "Forwarding Values... and Backwarding Them Too?")

```
1  #include <iostream>
2
3  struct Resource {};
4  struct Target {
5      Target(const Resource&) { std::cout << 'a'; }
6      Target(Resource&&) { std::cout << 'b'; }
7  };
8  struct ResourceManager {
9      Resource resource;
10
11     decltype(auto) visit(auto visitor) {
12         decltype(auto) result = visitor(resource);
13         return static_cast<decltype(result)>(result);
14     }
15 };
16
17 int main() {
18     ResourceManager rm;
19     Target(rm.visit([](Resource& r) -> Resource& { return r; }));
20     Target(rm.visit([](Resource& r) -> Resource { return r; }));
21     Target(rm.visit([](Resource& r) -> Resource&& { return std::move(r); }));
22 }
```

godbolt.org/z/96rjsq

Q: What is the output of the program?

```
1  #include <iostream>
2
3  struct Resource {
4      Resource() {}
5      Resource(const Resource&) { std::cout << 'a'; }
6  };
7  struct ResourceManager {
8      Resource resource;
9
10     Resource visit(auto visitor) {
11         Resource result = visitor(resource);
12         return result;
13     }
14 };
15 struct Target { Target(const Resource&) {}; };
16
17 int main() {
18     ResourceManager rm;
19     Target(rm.visit([](Resource& r) -> Resource { return r; }));
20 }
```

godbolt.org/z/aK8f4x

Q: What is the output of the program?

```
#include <iostream>

struct Resource {
    Resource() {}
    Resource(const Resource&) { std::cout << 'a'; }
};
struct ResourceManager {
    Resource resource;

    Resource visit(auto visitor) {
        Resource result = visitor(resource);
        return static_cast<Resource>(result);
    }
};
struct Target { Target(const Resource&) {}; };

int main() {
    ResourceManager rm;
    Target(rm.visit([](Resource& r) -> Resource { return r; }));
}
```

godbolt.org/z/rhhM8a

Missing (N)RVO

```
struct Resource {
    [...]
    Resource(const Resource&) { std::cout << 'a'; }
};

Resource visit(auto visitor) {
    Resource result = visitor(resource);
    return static_cast<Resource>(result);
}
```

Neither RVO nor NRVO!

- → `static_cast` is not the name of a variable (c-style cast does not work either)
- → compiler cannot elide observable side effects of copy construction
- → “Solution”
  - → remove explicit cast, or
  - → remove side effect (`std::cout`)

Missing (N)RVO

```
template <typename T>
decltype(auto) visit(T visitor) {
    decltype(auto) result = visitor(resource);
    if constexpr (std::is_same_v<decltype(result), Resource&&>) {
        return std::move(result);
    } else {
        return result;
    }
}
```

…works for GCC (without `auto` concept), not for Clang though

Missing (N)RVO

```
template <typename T>
static constexpr bool returns_rref = std::is_same_v<std::invoke_result_t<T,
    ↪ Resource&>, Resource&&>;

template <typename T, std::enable_if_t<returns_rref<T>, int> = 0>
decltype(auto) visit(T visitor) {
    decltype(auto) result = visitor(resource);
    return std::move(result);
}

template <typename T, std::enable_if_t<not returns_rref<T>, int> = 0>
decltype(auto) visit(T visitor) {
    decltype(auto) result = visitor(resource);
    return result;
}
```

…still, no NRVO with Clang but this time due to the deduced return type!

Missing (N)RVO

```cpp
template <typename T>
static constexpr bool returns_rref = std::is_same_v<std::invoke_result_t<T,
    ↪ Resource&>, Resource&&>;

template <typename T, std::enable_if_t<returns_rref<T>, int> = 0>
decltype(auto) visit(T visitor) {
    decltype(auto) result = visitor(resource);
    return std::move(result);
}

template <typename T, std::enable_if_t<not returns_rref<T>, int> = 0>
auto visit(T visitor) -> decltype(visitor(resource)) {
    decltype(auto) result = visitor(resource);
    return result;
}
```

…now works for GCC and Clang!


```cpp
#include <iostream>

struct Resource {
    Resource() {}
    Resource(const Resource&) { std::cout << 'a'; }
};
struct ResourceManager {
    Resource resource;

    template <typename T>
    static constexpr bool returns_rref = std::is_same_v<std::invoke_result_t<T,
        ↪ Resource&>, Resource&&>;

    template <typename T, std::enable_if_t<returns_rref<T>, int> = 0>
    decltype(auto) visit(T visitor) {
        decltype(auto) result = visitor(resource);
        return std::move(result);
    }

    template <typename T, std::enable_if_t<not returns_rref<T>, int> = 0>
[...]
```

godbolt.org/z/97jdrs

More things that don’t work

Missing (N)RVO

```cpp
#include <iostream>

struct Resource {
    Resource() {}
    Resource(const Resource&) { std::cout << 'a'; }
};
struct ResourceManager {
    Resource resource;

    template <typename T>
    auto visit(T visitor) -> decltype(visitor(resource)) {
        using R = std::invoke_result_t<T, Resource&>;

        decltype(auto) result = visitor(resource);
        if constexpr (std::is_same_v<R, Resource&&>) {
            return std::move(result);
        } else {
            return result;
        }
[...]
```

godbolt.org/z/P9n1o8

…works with GCC, fails with Clang

Missing (N)RVO

```
#include <iostream>

struct Resource {
    Resource() {}
    Resource(const Resource&) { std::cout << 'a'; }
};
struct ResourceManager {
    Resource resource;

    template <typename T>
    auto visit(T visitor) -> decltype(visitor(resource)) {
        using R = std::invoke_result_t<T, Resource&>;
        if constexpr (std::is_same_v<R, Resource&&>) {
            decltype(auto) result = visitor(resource);
            return std::move(result);
        } else {
            decltype(auto) result = visitor(resource);
            return result;
        }
[...]
```

godbolt.org/z/8heP76

…works with Clang, fails with GCC

Missing (N)RVO

```
#include <iostream>

struct Resource {
    Resource() {}
    Resource(const Resource&) { std::cout << 'a'; }
};
struct ResourceManager {
    Resource resource;

    template <typename T>
    auto visit(T visitor) -> decltype(visitor(resource)) {
        using R = std::invoke_result_t<T, Resource&>;
        if constexpr (decltype(auto) result = visitor(resource);
                      std::is_same_v<R, Resource&&>) {
            return std::move(result);
        } else {
            return result;
        }
    }
[...]
```

godbolt.org/z/e48EeT

…fails with GCC and Clang

Conclusion

(shamelessly copied from CppCon 2018: *Hayun Ezra Chung* “Forwarding Values… and Backwarding Them Too?”)

Forwarding

- Parameter Type: T&&
- Function Argument: `std::forward<E>(e)`
- Alternatively: `static_cast<decltype(e)&&>(e)`

Backwarding

- Parameter Type: `decltype(auto)`
- Function Argument: `decltype(e)(e)`
- Alternatively: `static_cast<decltype(e)>(e)`*

Everything you (n)ever wanted to know about C++'s Lambdas

iCSC 2020

Nis Meinert

Rostock University

Introduction

What is a Lambda Expression in C++?

cube is a lambda …

```
1  int main() {
2      auto cube = [](int x) { return x * x * x; };
3      return cube(3);
4  }
```

godbolt.org/z/zBE2_n

is_even is a lambda …

```
1  #include <algorithm>
2  #include <vector>
3
4  int main() {
5      std::vector<int> xs{1, 2, 3, 4, 5, 6, 7};
6      auto is_even = [](int x) { return x % 2 == 0; };
7      return std::count_if(xs.begin(), xs.end(), is_even);
8  }
```

godbolt.org/z/kWx7qu

Syntax

C++'s Lambda Expression

The simplest (and most boring) lambda

```
1 auto x = []{};
```

…no capturing, takes no parameters and returns nothing

A slightly more “useful” lambda

```
1 int main() {
2     auto x = [] { return 5; };
3     return x();
4 }
```

godbolt.org/z/DrnSSE

…is equivalent to

```
1 int main() {
2     struct {
3         auto operator()() const {
4             return 5;
5         }
6     } x;
7     return x();
8 }
```

godbolt.org/z/R8qx3Q

Q: What is the output of the program?

```
1 #!/usr/bin/env python3
2
3 if __name__ == '__main__':
4     f = {k: lambda x: x + k for k in range(3)}
5
6     for k in range(3):
7         print(f[k](2), end='')
```


Fix

```
1  #!/usr/bin/env python3
2
3  from functools import partial
4
5  if __name__ == '__main__':
6      f = {k: partial(lambda x, k: x + k, k=k) for k in range(3)}
7      # f = {k: lambda x, k=k: x + k for k in range(3)}
8      # ... would change API
9
10     for k in range(3):
11         print(f[k](2), end='')
```

Now prints: 234

Q: What is the output of the program?

```
1  #include <functional>
2  #include <iostream>
3  #include <map>
4
5  int main() {
6      std::unordered_map<int, std::function<int(int)>> f;
7
8      for (int k = 0; k < 3; k++) {
9          f.emplace(k, [](int x) { return x + k; });
10     }
11
12     for (int i = 0; i < 3; i++) {
13         std::cout << f[i](2);
14     }
15 }
```

godbolt.org/z/QssJXN

Q: What is the output of the program?

```cpp
#include <functional>
#include <iostream>
#include <map>

int main() {
    std::unordered_map<int, std::function<int(int)>> f;

    for (int k = 0; k < 3; k++) {
        f.emplace(k, [k](int x) { return x + k; });
    }

    for (int i = 0; i < 3; i++) {
        std::cout << f[i](2);
    }
}
```

godbolt.org/z/qHFY32

Q: What is the output of the program?

```cpp
#include <functional>
#include <iostream>
#include <map>

int main() {
    std::unordered_map<int, std::function<int(int)>> f;

    int k = 0;
    for (; k < 3; k++) {
        f.emplace(k, [&k](int x) { return x + k; });
    }

    for (int i = 0; i < 3; i++) {
        std::cout << f[i](2);
    }
}
```

godbolt.org/z/-FTWqI

C++'s Lambda Expression

Capturing rules

- `[x]`: captures `x` by value
- `[&x]`: captures `x` by reference
- `[=]`: captures all variables (used in the lambda) by value
- `[&]`: captures all variables (used in the lambda) by reference
- `[=, &x]`: captures variables like with `[=]`, but `x` by reference
- `[&, x]`: captures variables like with `[&]`, but `x` by value

Capturing by value

```cpp
int main() {
    int i = 1;
    auto z = [i](int y) {
        return i + y;
    }(3);
    return z;
}
```

godbolt.org/z/bHveG8

…or equivalently

```cpp
class X {
private:
    int i;

public:
    X(int i): i(i) {}

    int operator()(int y) const {
        return i + y;
    }
};

// potentially lots of lines of code

int main() {
    int i = 1;
    auto z = X{i}(3);
    return z;
}
```

godbolt.org/z/8tiwby

Capturing by reference

```
1  int main() {
2      int i = 1;
3      auto z = [&i](int y) {
4          return i + y;
5      }(3);
6      return z;
7  }
```

godbolt.org/z/xazquF

…or equivalently

```
1  class X {
2  private:
3      int& i;
4
5  public:
6      X(int& i): i(i) {}
7
8      int operator()(int y) /*const*/ {
9          return i + y;
10     }
11 };
12
13 // potentially lots of lines of code
14
15 int main() {
16     int i = 1;
17     auto z = X{i}(3);
18     return z;
19 }
```

godbolt.org/z/3ycaAW

Q: What is the output of the program?

```
1  #include <iostream>
2
3  int main() {
4      int i = 1;
5      auto x = [i]() { return ++i; };
6      std::cout << i << x() << i;
7  }
```

godbolt.org/z/nv83nh

Q: What is the output of the program?

```
1  #include <iostream>
2
3  int main() {
4      int i = 1;
5      auto x = [i]() mutable { return ++i; };
6      std::cout << i << x() << i;
7  }
```

godbolt.org/z/Gs995r

Q: What is the output of the program?

```
1  #include <iostream>
2
3  int main() {
4      int i = 1;
5      auto x = [&i]() mutable { return ++i; };
6      std::cout << i << x() << i;
7  }
```

godbolt.org/z/gEhwLt

Q: What is the output of the program?

```
#include <iostream>

int main() {
    auto x = [i=0]() mutable { return ++i; };
    std::cout << x() << x();
}
```

godbolt.org/z/iLqPrn

Q: What is the output of the program?

```
#include <iostream>
#include <utility>

int main() {
    auto x = [i=0, j=1]() mutable {
        i = std::exchange(j, j + i);
        return i;
    };

    for (int i = 0; i < 5; ++i) {
        std::cout << x();
    }
}
```

godbolt.org/z/eTdadM

(cppreference.com/w/cpp/utility/exchange)

C++'s Lambda Expression

Remember, lambda expressions are pure syntactic sugar and are equivalent to `structs` with an appropriate `operator()()` overload …

Q: What is the output of the program?

```
#include <iostream>

int main() {
    auto x = [] { return 1; };
    auto y = x;
    std::cout << x() << y();
}
```

godbolt.org/z/i_AnMx

Q: What is the output of the program?

```
#include <iostream>

int main() {
    int i = 1;
    int j = 2;
    auto x = [&i, j] { return i + j; };
    i = 4;
    j = 6;
    auto y = x;
    std::cout << x() << y();
}
```

godbolt.org/z/35Q3uR

Q: What is the output of the program?

```
#include <iostream>
#include <memory>

int main() {
    auto x = [i=std::make_unique<int>(1)] { return *i; };
    auto y = x;
    std::cout << x () << y();
}
```

godbolt.org/z/u-6mxM

Stateful Lambdas

Q: What is the output of the program?

```
#include <iostream>

int main() {
    auto x = [i=0]() mutable { return ++i; };
    auto y = x;
    x();
    x();
    y();
    y();
    std::cout << x();
}
```

godbolt.org/z/U-CLpA

Q: What is the output of the program?

```cpp
#include <iostream>

int main() {
    auto x = [] { static int i = 0; return ++i; };
    auto y = x;
    x();
    x();
    y();
    y();
    std::cout << x();
}
```

godbolt.org/z/_8QjoA

Stateful Lambdas

Let us now try to interact with the state of the Lambda …

```cpp
#include <utility>

int main() {
    auto fib = [i=0, j=1]() mutable {
        struct Result {
            int &i, &j;

            auto next() {
                i = std::exchange(j, j + i);
                return *this;
            }
        };
        return Result{.i=i, .j=j}.next();
    };

    auto r = fib();
    r.i = 2; // mutate state
    r.j = 3; // mutate state
    return fib().j; // 5
}
```

godbolt.org/z/xpLDpb

Stateful Lambdas

Fibonacci (again):

```cpp
#include <utility>

int main() {
    auto fib = [i=0, j=1]() mutable {
        struct Result {
            int &i, &j;

            auto next() {
                i = std::exchange(j, j + i);
                return *this;
            }
        };
        return Result{.i=i, .j=j}.next();
    };

    fib().next().next().next(); // mutate state
    return fib().i;
}
```

godbolt.org/z/m9s7ei

Stateful Lambdas

…or slightly more conveniently:

```cpp
#include <utility>

int main() {
    auto fib = [i=0, j=1]() mutable {
        struct Result {
            int &i, &j;

            auto next(int n = 1) {
                while (n-- > 0) {
                    i = std::exchange(j, j + i);
                }
                return *this;
            }
        };
        return Result{.i=i, .j=j}.next();
    };

    return fib().next(3).j; // 5
}
```

godbolt.org/z/aN3sNi

Stateful Lambdas

```
1  #include <utility>
2
3  int main() {
4      auto fib = [i=0, j=1]() mutable {
5          struct Result {
6              int &i, &j;
7
8              auto next(int n = 1) {
9                  while (n-- > 0) {
10                     i = std::exchange(j, j + i);
11                 }
12                 return *this;
13             }
14         };
15         return Result{.i=i, .j=j}.next();
16     };
17
18     return fib().next(10).j; // 144
19 }
```

godbolt.org/z/ok7Za-

```
# g92 -O3
   | main:
19|   mov eax, 144
19|   ret
```

godbolt.org/z/ok7Za-

Best Practices

(partially taken from “Effective Modern C++” by Scott Meyers)

Use Lambdas in STL algorithm

```
1  #include <algorithm>
2  #include <vector>
3
4  std::vector<int> get_ints();
5
6  int main() {
7      auto ints = get_ints();
8      auto in_range = [](int x) { return x > 0 && x < 10; };
9      return *std::find_if(ints.begin(), ints.end(), in_range);
10 }
```

godbolt.org/z/y-343Z

Use Lambdas in STL algorithm

```
1  #include <algorithm>
2  #include <vector>
3
4  std::vector<int> get_ints();
5
6  int main() {
7      auto ints = get_ints();
8      return *std::find_if(ints.begin(), ints.end(),
9                           [](int x) { return x > 0 && x < 10; });
10 }
```

godbolt.org/z/J7cccJ

Stop pollution of namespace with helper variables

```cpp
#include <cmath>
#include <iostream>

int main() {
    auto y = []<typename T>(T x) {
        T mean = 1.;
        T width = 3.;
        auto norm = 1. / std::sqrt(2. * M_PI);
        auto arg = (x - mean) / width;
        return norm * std::exp(-.5 * arg * arg);
    }(.5);

    std::cout << y;
}
```

godbolt.org/z/NC6DKj

Allow variables to be `const`

```cpp
#include <vector>

std::vector<int> get_ints();

int main() {
    auto ints = get_ints();
    const auto sum = [&ints] {
        int acc = 0;
        for (auto& x: ints) acc += x;
        return acc;
    }();

    return sum;
}
```

godbolt.org/z/p_I8hF

Avoid default capture modes

Below, there is a dangling pointer lurking in the wings …

```cpp
void add_filter() {
    auto divisor = get_magic_number();
    filters.emplace_back([&](int x) { return x % divisor == 0; });
}
```

This error becomes more obvious, when explicit capturing is used:

```cpp
void add_filter() {
    auto divisor = get_magic_number();
    filters.emplace_back([&divisor](int x) { return x % divisor == 0; });
}
```


Avoid default capture modes

Mitigation of copy & paste bugs:

```cpp
auto divisor = get_magic_number();
std::find_if(container.begin(),
             container.end(),
             [&divisor](int x) { return x % divisor == 0; });
```

`[&divisor]` indicates that there is an *external* dependency and it is not enough to "just copy" the lambda function if needed elsewhere.

(off-topic: check out this interesting article about copy & paste bugs in real world applications: "The Last Line Effect" by the PVS-Studio team, `www.viva64.com/en/b/0260/`)

Avoid default capture modes

Does the following implementation looks fine?

```
1  struct Widget {
2      int divisor = 2;
3
4      void add_filter() const {
5          filters.emplace_back([=](int x) { return x % divisor == 0; });
6      }
7  };
```

…given a sufficient implementation of `filters`

No! Horrible code! Capturing only applies to non-`static` local variables. Why does this work?

Avoid default capture modes

Capturing only applies to non-`static` local variables. Why does this work?

```
1  Widget::add_filter() const {
2      filters.emplace_back([=](int x) { return x % divisor == 0; });
3  }
```

…but this fails

```
1  Widget::add_filter() const {
2      filters.emplace_back([](int x) { return x % divisor == 0; });
3  }
```

…and this also

```
1  Widget::add_filter() const {
2      filters.emplace_back([divisor](int x) { return x % divisor == 0; });
3  }
```


Avoid default capture modes

There is no local variable `divisor`! But what happes is the following

```
1  Widget::add_filter() const {
2      filters.emplace_back([=](int x) {
3          return x % divisor == 0;
4      });
5  }
```

copies (implicitly) `this` pointer (until C++17), i.e.

```
1  Widget::add_filter() const {
2      auto copy_of_this = this;
3      filters.emplace_back([copy_of_this](int x) {
4          return x % copy_of_this->divisor == 0;
5      });
6  }
```

…welcome to the world of undefined behavior, when `Widget` goes out of scope!

Avoid default capture modes

Default capturing by value can be misleading and gives the impression that a lambda is self-contained:

```
1  static auto divisor = 1;
2  filters.emplace_back([=](int x) { return x % divisor == 0; });
3  ++divisor;
```

Above, `divisor` is not copied! (as one may have guessed seeing `[=]`)

Stop using `std::bind`

Stop using `std::bind`

…and prefer lambda expression, since

- this increases readability,
- lambdas are much more flexible,
- `std::bind` can potentially introduce additional overhead at run-time, whereas lambdas are default `constexpr`

Stop using `std::function`

Stop using `std::function`

- `std::function` add multiple copies of passed object (consider using drop-in replacements such as *delegates**)
- may cause heap allocation
- is just a wrapper …

…deduce type of lambda via `auto` or template deduction, **if possible** (cf. exercise)

*codereview.stackexchange.com/questions/14730/impossibly-fast-delegate-in-c11

Inheriting from Lambdas

Inheriting from Lambdas

Consider two lambdas

```
1 auto f1 = [] { return 1; };
2 auto f2 = [](int x) { return x; };
```

Is it possible to combine both lambdas (by inheritance) in one common type X?

```
1 X combined{f1, f2};
2 auto a = combined();   // should return 1
3 auto b = combined(42); // should return 42
```

Inheriting from Lambdas

```
1  struct X: F1, F2 {
2      X(F1 f1, F2 f2): F1(std::move(f1)), F2(std::move(f2)) {}
3
4      using F1::operator();
5      using F2::operator();
6  };
```

…but what is the type of a lambda / what are F1 and F2?

According to the C++17 standard, will this compile?

```
1  #include <iostream>
2
3  template <typename F1, typename F2> struct X: F1, F2 {
4      X(F1 f1, F2 f2): F1(std::move(f1)), F2(std::move(f2)) {}
5      using F1::operator();
6      using F2::operator();
7  };
8
9  int main() {
10     auto f1 = [] { return 1; };
11     auto f2 = [](int x) { return x; };
12     X combined{f1, f2};
13     std::cout << combined() << combined(2); // should print "12"
14 }
```

godbolt.org/z/nMNbMZ

According to the C++14 standard, will this compile?

```
1  #include <iostream>
2
3  template <typename F1, typename F2> struct X: F1, F2 {
4      X(F1 f1, F2 f2): F1(std::move(f1)), F2(std::move(f2)) {}
5      using F1::operator();
6      using F2::operator();
7  };
8
9  int main() {
10     auto f1 = [] { return 1; };
11     auto f2 = [](int x) { return x; };
12     X combined{f1, f2};
13     std::cout << combined() << combined(2); // should print "12"
14 }
```

godbolt.org/z/nMNbMZ

Inheriting from Lambdas

What are the deduced types of auto / what are the types of f1 and f2?

```
1  auto f1 = [] { return 1; };
2  auto f2 = [](int x) { return x; };
```

Use decltype to find out!

```
1  X<decltype(f1), decltype(f2)> combined{f1, f2};
```

Inheriting from Lambdas

…or extract this to a factory function `make_combined`

```cpp
#include <iostream>

template <typename F1, typename F2> struct X: F1, F2 {
    X(F1 f1, F2 f2): F1(std::move(f1)), F2(std::move(f2)) {}
    using F1::operator();
    using F2::operator();
};

template <typename F1, typename F2> auto make_combined(F1&& f1, F2&& f2) {
    return X<std::decay_t<F1>, std::decay_t<F2>>{std::forward<F1>(f1),
                                                 std::forward<F2>(f2)};
}

int main() {
    auto f1 = [] { return 1; };
    auto f2 = [](int x) { return x; };
    auto combined = make_combined(f1, f2);
    std::cout << combined() << combined(2); // should print "12"
}
```

godbolt.org/z/dmBP8E

According to the C++17 standard, will this compile?

```cpp
#include <iostream>

template <typename F1, typename F2> struct X: F1, F2 {
    using F1::operator();
    using F2::operator();
};

int main() {
    auto f1 = [] { return 1; };
    auto f2 = [](int x) { return x; };
    X combined{f1, f2};
    std::cout << combined() << combined(2); // should print "12"
}
```

godbolt.org/z/MhrL87

According to the C++17 standard, will this compile?

```cpp
#include <iostream>

template <typename F1, typename F2> struct X: F1, F2 {
    using F1::operator();
    using F2::operator();
};

template <typename F1, typename F2>
X(F1, F2) -> X<std::decay_t<F1>, std::decay_t<F2>>;

int main() {
    auto f1 = [] { return 1; };
    auto f2 = [](int x) { return x; };
    X combined{f1, f2};
    std::cout << combined() << combined(2); // should print "12"
}
```

godbolt.org/z/qDYu3G

Variadic Templates

```cpp
#include <iostream>

template <typename... Fs> struct X: Fs... {
    using Fs::operator()...;
};

template <typename... Fs>
X(Fs...) -> X<std::decay_t<Fs>...>;

int main() {
    auto f1 = [] { return 1; };
    auto f2 = [](int x) { return x; };
    auto f3 = [](double x) { return -x; };
    X combined{f1, f2, f3};
    std::cout << combined() << '\n'     // should print "1"
              << combined(2) << '\n'    // should print "2"
              << combined(3.4) << '\n'; // should print "-3.4"
}
```

godbolt.org/z/T8wYP2

Inheriting from Lambdas

Why?

{

std::variant

An enum class models a **choice between values**:

```
1 enum class Oven { on, off };
```

`std::variant` models a **choice between types**:

```
1 struct on { double temperature; };
2 struct off {};
3 using Oven = std::variant<on, off>;
```


std::variant

An aggregate type of some simple shapes …

```
1 struct Shape {
2     enum class Type { Circle, Box } type;
3
4     union {
5         struct { double radius; } circle;
6         struct { double width, height; } box;
7     } geometry;
8 };
```

std::variant

…and an outer function that calculates the respective area

```
auto area(const Shape& shape) {
    switch(shape.type) {
        case Shape::Type::Circle: {
            const auto& g = shape.geometry.circle;
            return M_PI * g.radius * g.radius;
        }
        case Shape::Type::Box: {
            const auto& g = shape.geometry.box;
            return g.width * g.height;
        }
    }

    assert(false);
    __builtin_unreachable();
}
```



```
#include <cassert>
#include <cmath>

struct Shape {
    enum class Type { Circle, Box } type;

    union {
        struct { double radius; } circle;
        struct { double width, height; } box;
    } geometry;
};

auto area(const Shape& shape) {
    switch(shape.type) {
        case Shape::Type::Circle: {
            const auto& g = shape.geometry.circle;
            return M_PI * g.radius * g.radius;
        }
        case Shape::Type::Box: {
[...]
```

godbolt.org/z/U6Uaip

Using std::variant instead

```
#include <cmath>
#include <variant>

struct Circle { double radius; };
struct Box { double width, height; };
using Shape = std::variant<Circle, Box>;

auto area(const Shape& shape) {
    struct {
        auto operator()(const Circle& c) const {
            return M_PI * c.radius * c.radius;
        }
        auto operator()(const Box& b) const {
            return b.width * b.height;
        }
    } visitor;

    return std::visit(visitor, shape);
}
```

godbolt.org/z/nkvKi2

}

Q: What is the output of the program?

```cpp
#include <algorithm>
#include <iostream>
#include <variant>
#include <vector>

template <typename... Fs> struct X: Fs... {
    using Fs::operator()...;
};
template <typename... Fs> X(Fs...) -> X<std::decay_t<Fs>...>;

int main() {
    int a = 0; double b = 0.;
    X visitor{[&a](int x) { a += x; },
              [&b](double x) { b += x; }};
    std::vector<std::variant<int, double>> v{1, 1.9, 2, 2.1};
    std::for_each(v.begin(), v.end(), [&visitor](const auto &x) {
        std::visit(visitor, x);
    });
    std::cout << a << ' ' << b;
}
```

godbolt.org/z/8j3M7v

Q: What is the output of the program?

```cpp
#include <algorithm>
#include <iostream>
#include <variant>
#include <vector>

template <typename... Fs> struct X: Fs... {
    using Fs::operator()...;
};
template <typename... Fs> X(Fs...) -> X<std::decay_t<Fs>...>;

int main() {
    int a = 0; double b = 0.;
    X visitor{[&a](int x) { a += x; },
              [&b](double x) { b += x; }};
    std::vector<std::variant<int, double, const char*>> v{1, 1.9, 2, 2.1, "foo"};
    std::for_each(v.begin(), v.end(), [&visitor](const auto& x) {
        std::visit(visitor, x);
    });
    std::cout << a << b;
}
```

godbolt.org/z/qYwPjF

Using plain old `structs`

```cpp
#include <algorithm>
#include <iostream>
#include <variant>
#include <vector>

struct X {
    int &a; double &b;
    auto operator()(int x) { a += x; };
    auto operator()(double x) { b += x; };
};

int main() {
    int a = 0; double b = 0.;
    X visitor{.a=a, .b=b};
    std::vector<std::variant<int, double>> v{1, 1.9, 2, 2.1};
    std::for_each(v.begin(), v.end(), [&visitor](const auto& x) {
        std::visit(visitor, x);
    });
    std::cout << a << b;
}
```

godbolt.org/z/E6SNXT

Nota bene

One could also use a generic lambda…

```cpp
#include <algorithm>
#include <iostream>
#include <variant>
#include <vector>

int main() {
    int a = 0; double b = 0.;
    std::vector<std::variant<int, double>> v{1, 1.9, 2, 2.1};
    std::for_each(v.begin(), v.end(), [&a, &b](const auto& x) {
        std::visit([&a, &b](auto x) {
            if constexpr (std::is_same_v<int, decltype(x)>) a += x;
            else b += x;
        }, x);
    });
    std::cout << a << b;
}
```

godbolt.org/z/Dcdmoi

…however, no check for exhaustiveness at compile-time here!

Q: What is the output of the program?

```cpp
#include <iostream>
#include <variant>

struct A { auto f() { return 1; }};
struct B { auto g() { return 2; }};

int main() {
    std::visit([](auto x) {
        using X = decltype(x);
        if constexpr (std::is_same_v<X, A>) {
            std::cout << x.f();
        } else if constexpr (std::is_same_v<X, B>) {
            std::cout << x.g();
        } else {
            std::cout << x.palim();
        }
    }, std::variant<A, B>{A{}});
}
```

godbolt.org/z/Dyy9mg

`std::variant` evaluation at compile-time

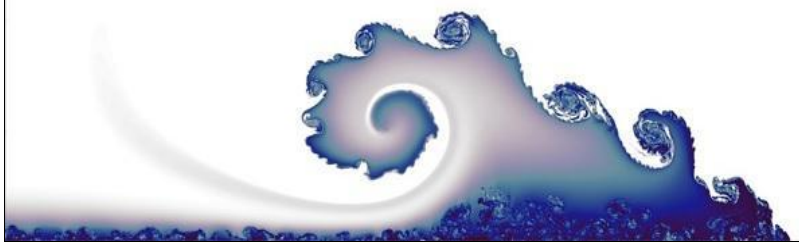
```cpp
#include <variant>

template<typename... Ts> struct overloaded : Ts... { using Ts::operator()...; };
template<typename... Ts> overloaded(Ts...) -> overloaded<Ts...>;

int main() {
    using T = std::variant<int, double>;
    overloaded visitor = overloaded{[](int x) -> T { return x + 1; },
                                    [](double x) -> T { return x + 2.; }};
    constexpr auto result = std::visit(visitor, T{41});
    static_assert(result.index() == 0 && std::get<0>(result) == 42);
}
```

godbolt.org/z/b988jT

Computational fluid dynamics for physicists and engineers

Ruchi Mishra
Nicolaus Copernicus Astronomical Center

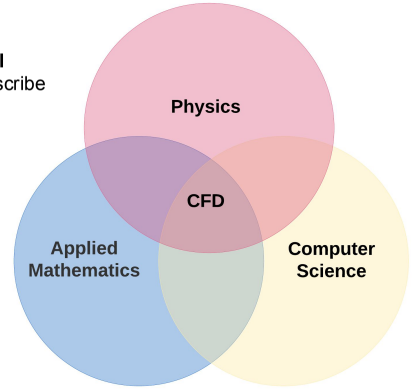
Inverted CERN school of Computing (iCSC), CERN, 28th Sep- 2nd Oct 2020



Content

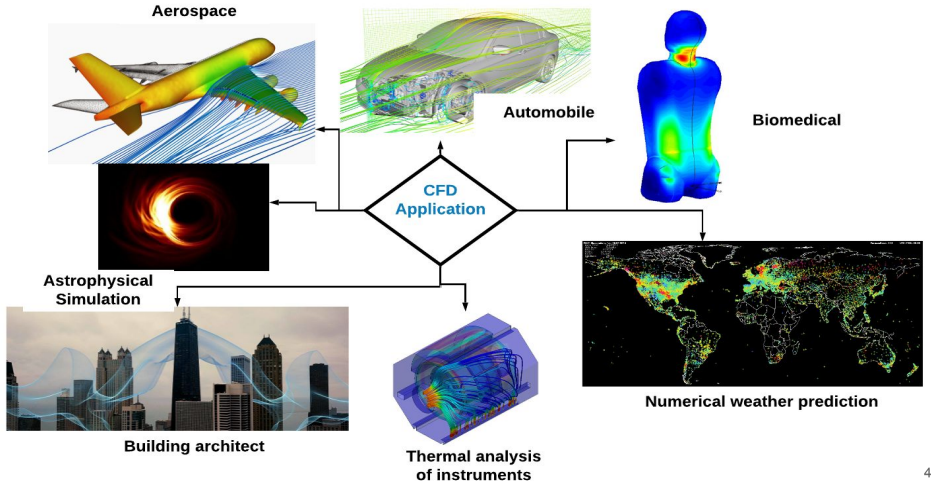
WHAT IS CFD?

Fluid flows are governed by system of **partial differential equations** (PDEs) which describe the conservation of mass, momentum and energy.

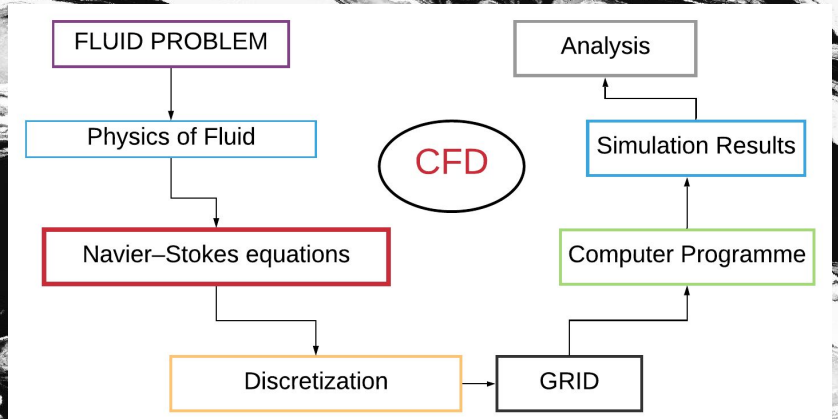


Computational fluid dynamics (CFD) solves these PDEs by replacing them with algebraic equations.

CFD APPLICATION IN DIFFERENT FIELDS



HOW DOES CFD WORK?



5

EQUATIONS GOVERNING FLUID FLOW

$$-\frac{\partial \rho}{\partial t} = \frac{\partial(\rho v)}{\partial x}$$

Mass conservation equation / continuity equation

$$\rho\left(\frac{Dv}{Dt}\right) = -\nabla P + \mu\nabla^2 v + F_x$$

mass X acceleration | Pressure gradiant | Internal Forces | External Forces

Momentum conservation equation

Navier-Stokes equation

$$\frac{D}{Dt} = \left[\frac{\partial}{\partial t} + (v.\nabla)\right]$$

$$\frac{\partial e}{\partial t} + v.\nabla e = -\frac{P}{\rho}\nabla.v$$

Energy conservation equation

$$p = (\gamma - 1)\rho e$$

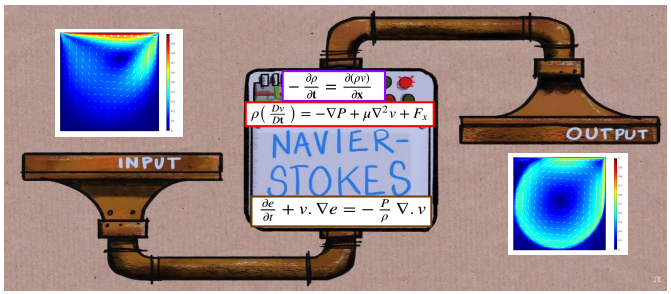
Equation of state

6

FROM INPUT TO OUTPUT

Initial conditions

Boundary conditions



Unknown Physical Quantities

ρ Density v Velocity p Pressure e Energy

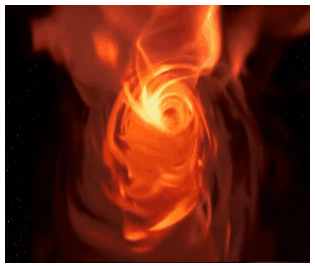
7

DIFFERENT APPROACHES TO MODEL FLUID IN CFD

Eulerian Approach

Grid based hydrodynamics

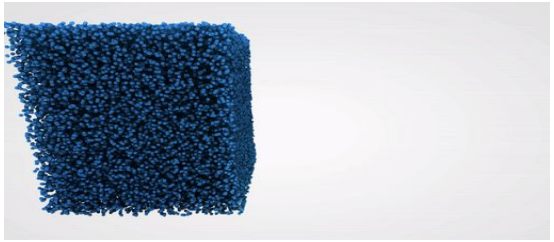
- Solves the fluid dynamics equations by calculating the flux of conserved quantities through adjacent cell boundaries



Lagrangian Approach

Smooth particle hydrodynamics(SPH)

- Calculates the properties on each particle by averaging over its nearest neighbour
- Satisfies mass conservation without extra computation as the particles themselves represent mass



*IN THIS LECTURE WE ARE GOING TO COVER ONLY THE **GRID BASED HYDRODYNAMICS***

8

SIMPLIFIED EQUATIONS IN CONSERVED FORM

$$\rho\left(\frac{Dv}{Dt}\right) = -\nabla P + \mu\nabla^2 v + F_x$$

mass X acceleration | Pressure gradiant | Internal Forces | External Forces

Momentum conservation equation

Navier-Stokes equations

We replace our equations by simpler ones.

$$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho v)}{\partial x} = 0$$

$$\frac{\partial(\rho v)}{\partial t} + \frac{\partial(\rho v^2 + P)}{\partial x} = 0$$

$$\frac{\partial e}{\partial t} + \frac{\partial((e+P)v)}{\partial x} = 0$$

Original Navier-Stokes equation reduces to Euler equations.

$$\frac{\partial U}{\partial t} + \frac{\partial F}{\partial x} = 0$$

$$\frac{\partial}{\partial t}\left[\text{Conserved quantity}\right] + \frac{\partial}{\partial x}\left[\text{Flux}\right] = 0$$

How to solve it numerically ?

9

HOW TO SOLVE THEM NUMERICALLY?

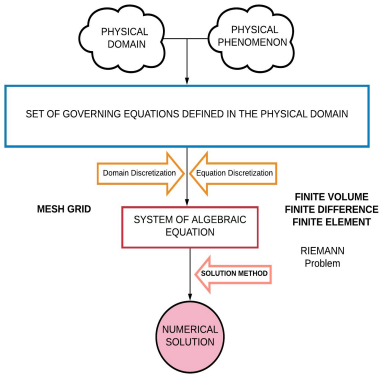
Unknown Physical Quantities

ρ Density v Velocity p Pressure e Energy

$$\frac{\partial}{\partial t}\left[\text{Conserved quantity}\right] + \frac{\partial}{\partial x}\left[\text{Flux}\right] = 0$$

Physical domain: space (x,y,z) and time t
Physical quantities: $\rho\, v\, p\, e$

Since we will solve equations numerically, we have to discretize
1) Physical domain
2) Physical quantities *(aka equation discretization)*

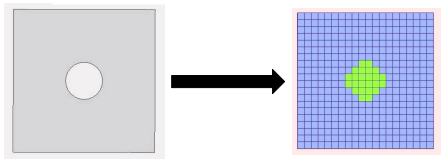


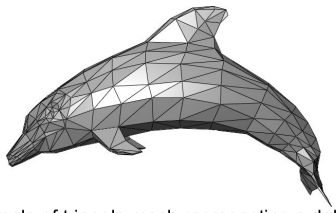
10

DOMAIN DISCRETIZATION

MESH GRID - Division of a continuous geometric space into discrete geometric cells

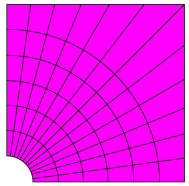
Model of flow around cylinder using **cartesian grid.**



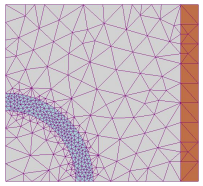


Example of triangle mesh representing a dolphin

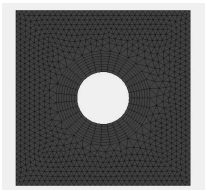
Structured curvilinear grid



Unstructured curvilinear grid



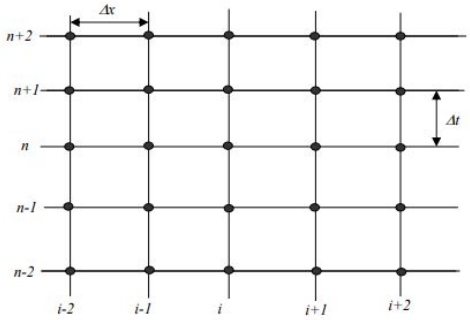
Hybrid grid



11

THE DISCRETIZED PHYSICAL DOMAIN

Grid generation

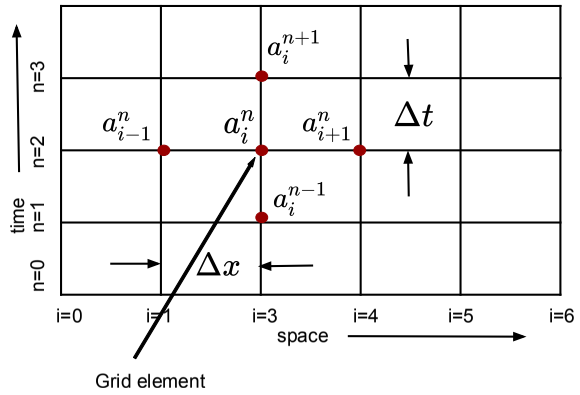


- A simple method of placing points in the domain
- Each point is labeled using i for spatial discretization and n for time discretization
- The spacing can be of variable size

12

DISCRETIZATION OF PHYSICAL QUANTITIES

Equation discretization



Backward difference

$$\frac{\partial a_i^n}{\partial x} \simeq \frac{a_i^n - a_{i-1}^n}{\Delta x}$$

Forward difference

$$\frac{\partial a_i^n}{\partial x} \simeq \frac{a_i^{n+1} - a_i^n}{\Delta x}$$

Central difference

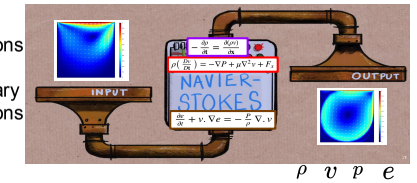
$$\frac{\partial a_i^n}{\partial x} \simeq \frac{a_{i+1}^n - a_{i-1}^n}{2\Delta x}$$

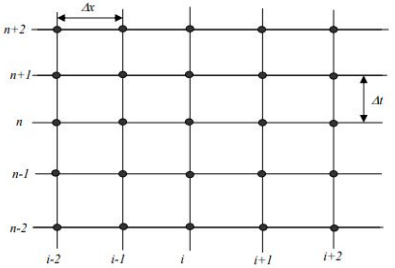
13

SPECIFYING INPUT THROUGH INITIAL AND BOUNDARY CONDITIONS

Initial conditions

Boundary conditions



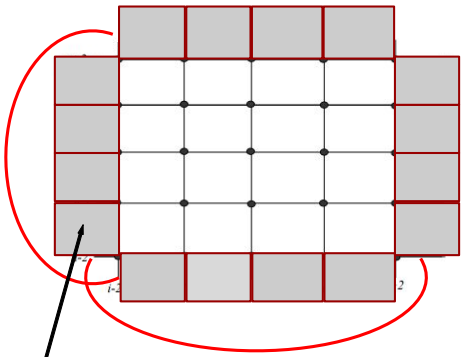


Some of the boundary conditions used in CFD

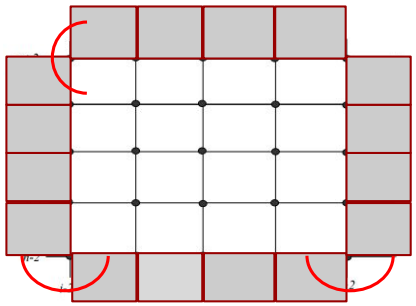
1- inlet condition
2- symmetric condition
3- periodic boundary condition
4- reflective boundary condition
5- outlet condition

BOUNDARY CONDITIONS

Periodic Boundary Condition



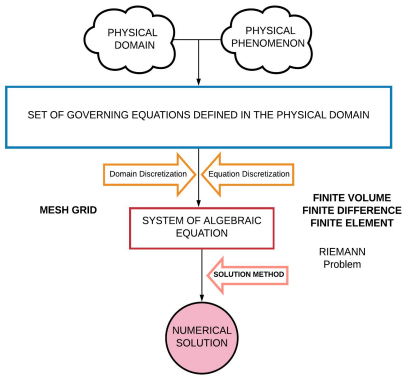
Symmetric Boundary Condition



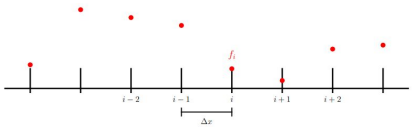
Ghost cells are used here to extend the grid beyond physical boundary to accommodate boundary condition

15

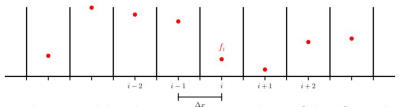
STORING DATA IN GRIDS



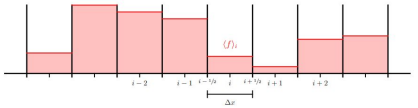
Finite-difference grid : Data is stored at grid edges



Cell-centered finite-difference grid : Data is stored at cell centers

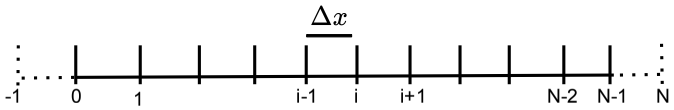


Finite-volume grid—the average value of the function is stored within each zone.



16

EQUATION DISCRETIZATION USING FINITE DIFFERENCE METHOD



Finite difference grid with ghost cell at each end

$$\frac{\partial U}{\partial t} = -\frac{\partial F}{\partial x}$$

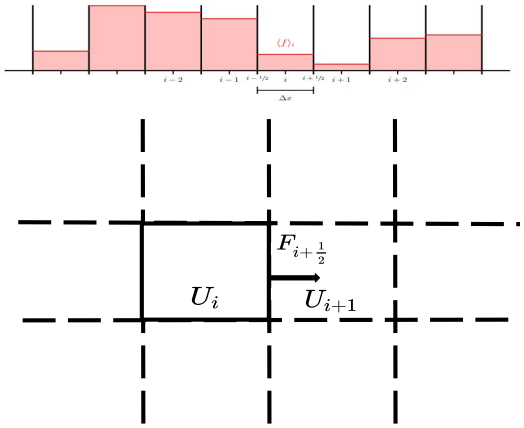
$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = \frac{-1}{\Delta x}\left[F_i^n - F_{i-1}^n\right]$$

Time evolution

$$U_i^{n+1} = \frac{-\Delta t}{\Delta x}\left[F_i^n - F_{i-1}^n\right] + U_i^n$$

17

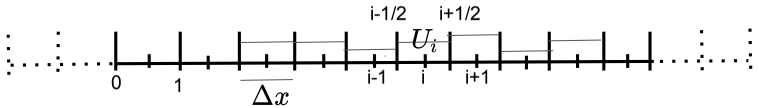
FINITE VOLUME METHOD



Fluxes are calculated at cell edges (i ± 1/2)

18

EQUATION DISCRETIZATION USING FINITE DIFFERENCE METHOD



Finite volume grid with two ghost cells at both ends

$$\frac{\partial U}{\partial t} = -\frac{\partial F}{\partial x}$$

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = \frac{-1}{\Delta x}\left[F_{i+\frac{1}{2}}^{n+\frac{1}{2}} - F_{i-\frac{1}{2}}^{n-\frac{1}{2}}\right]$$

Time evolution

$$U_i^{n+1} = \frac{-\Delta t}{\Delta x}\left[F_{i+\frac{1}{2}}^{n+\frac{1}{2}} - F_{i-\frac{1}{2}}^{n-\frac{1}{2}}\right] + U_i^n$$

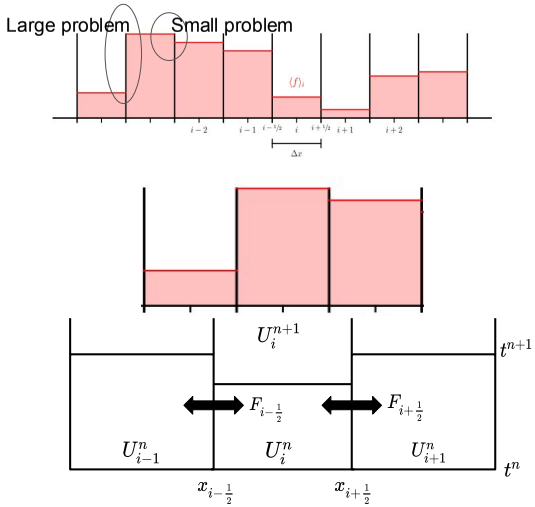
19

At the interface there will be a jump.
How do we calculate flux at the interface ?

For flux evaluation at half time, we need information of state U at half time

$$[F_{i+\frac{1}{2}}^{n+\frac{1}{2}}] = f(U_{i+\frac{1}{2}}^{n+\frac{1}{2}})$$

THE RIEMANN PROBLEM



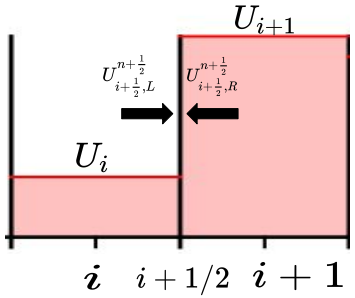
20

THE RIEMANN PROBLEM

Two states separated by a discontinuity.

This is called a Riemann problem.

Solution to Riemann problem results in single state at interface.



$$U^{n+\frac{1}{2}}_{i+\frac{1}{2}} = R\left(U^{n+\frac{1}{2}}_{i+\frac{1}{2},L}, U^{n+\frac{1}{2}}_{i+\frac{1}{2},R}\right)$$

21

APPROXIMATE RIEMANN SOLVERS

$$U^{n+\frac{1}{2}}_{i+\frac{1}{2}} = R\left(U^{n+\frac{1}{2}}_{i+\frac{1}{2},L}, U^{n+\frac{1}{2}}_{i+\frac{1}{2},R}\right)$$

Different Riemann Solvers (Approximate solvers)

- ROE Solver
- HLL Solver
- Rotated-hybrid Riemann solvers

The exact solution of Riemann problem at every interface is very expensive!

Use approximate Riemann solver instead!

22

HLL RIEMANN SOLVER

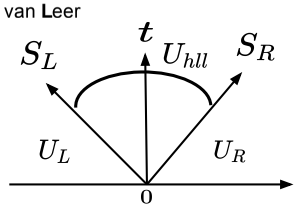
Different Riemann Solvers

- ROE Solver
- **HLL Solver**
- Rotated-hybrid Riemann solvers

Harten, **L**ax and van **L**eer

Solution is computed from two wave speeds S_L and S_R

If we have algorithm to track these wave speeds an approximate intercell flux can be calculated from it.



$$F^{hll}_{i+\frac{1}{2}} = \begin{cases} F_L & \text{if} \quad 0 \le S_L \\ \frac{S_R F_L - S_L F_R + S_L S_R (U_R - U_L)}{S_R - S_L} & \text{if} \quad S_L \le 0 \le S_R \\ F_R & \text{if} \quad 0 \ge S_L \end{cases}$$

23

We have a new problem while discretizing the time.

CFL CONDITION

When we discretize the time, the step must be less than the time it takes for the information to propagate across a single zone.

This is called **CFL**(Courant–Friedrichs–Lewy) condition

$$\Delta t \le \frac{\Delta x}{v}$$

$$C = \frac{v \Delta t}{\Delta x}$$

$$C \le 1$$

Necessary condition for stability

24

SHOCK TUBE OR SOD PROBLEM IN 1D

Gary A.Sod (1978)

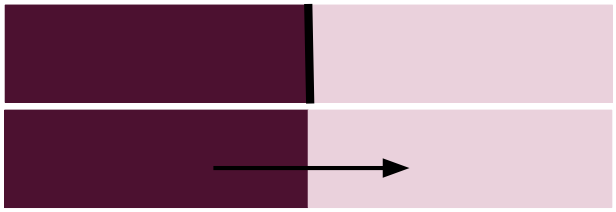
Commonly used problem to test accuracy of CFD codes using Riemann Solver.

Initial Condition

$$v_L = 0 \qquad v_R = 0$$
$$\rho_L = 1 \qquad \rho_R = 0.1$$
$$P_L = 1 \qquad P_R = 0.125$$

wall

1. The fluid (gas) is initially at rest separated by a wall
2. The sudden breakdown of the wall generates a high-speed flow resulting a shock wave, which propagates to the right



25

THE ALGORITHM

EQUATION IN CONSERVED FORM

$$\frac{\partial \rho}{\partial t} + \frac{\partial (\rho v)}{\partial x} = 0$$
$$\frac{\partial (\rho v)}{\partial t} + \frac{\partial (\rho v^2 + P)}{\partial x} = 0$$
$$\frac{\partial e}{\partial t} + \frac{\partial ((e + P)v)}{\partial x} = 0$$

U

$$\begin{Bmatrix} \rho \\ \rho v \\ \rho e + \frac{1}{2}\rho v^2 \end{Bmatrix}$$

Conserved quantities

F

$$\begin{Bmatrix} \rho v \\ \rho v^2 + P \\ (\rho e + \frac{1}{2}\rho v^2 + P)v \end{Bmatrix}$$

Fluxes

P

$$\begin{Bmatrix} \rho \\ v \\ e \end{Bmatrix}$$

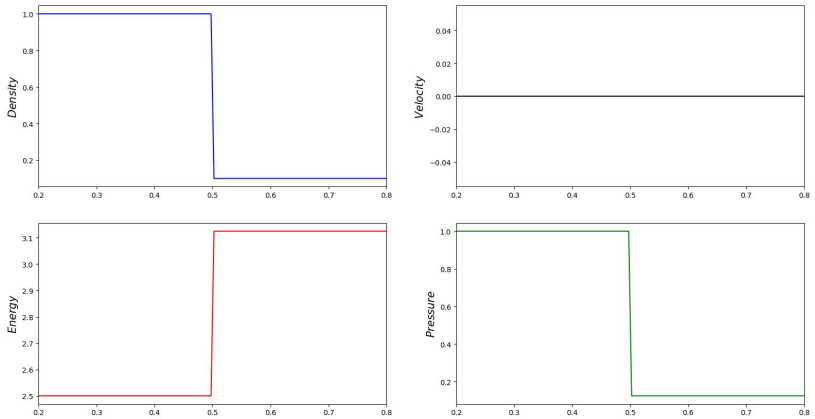
Physical quantities

- Discretize physical domain by setting **grid cells**
- **Set initial conditions for physical quantities**
- **Reconstruct** Calculation left step P_L and P_R right state of physical quantity
- **Calculate flux** (from P_R and P_L, F_L and F_R are calculated)
- Convert physical quantity to conserved quantity **(P to U)**
- Solve the **Reimann Problem**
- Check **CFL condition**
- **UPDATE IN TIME**
- Convert conserved quantity to physical quantity **(U to P)**

26

INITIAL CONDITION

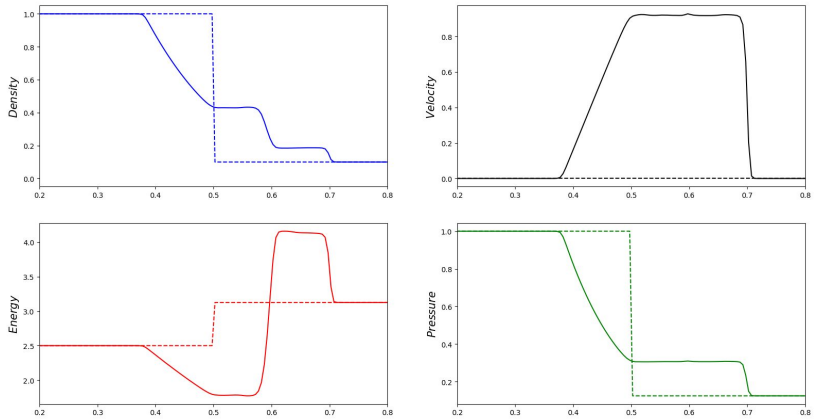
t = 0



27

TIME EVOLUTION

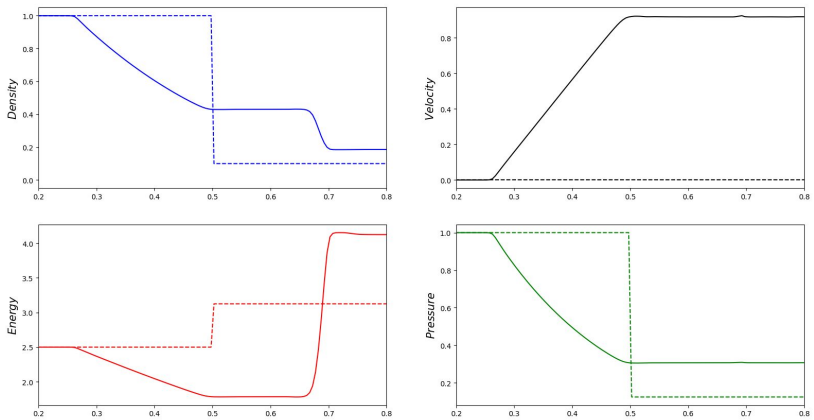
t = 0.1



28

TIME EVOLUTION

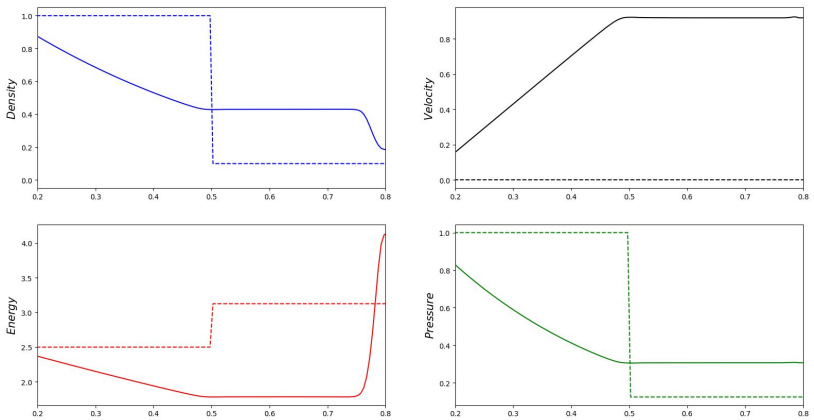
t = 0.2



29

TIME EVOLUTION

t = 0.3



30

Why do we do simulation in Astrophysics?

SIMULATION IN ASTROPHYSICS

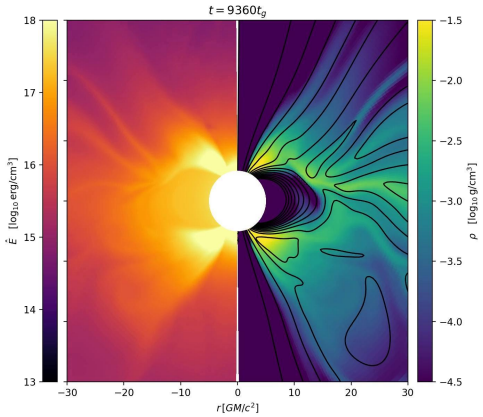
Simulation enables us to build a model of a system

It allows us to do virtual experiments to understand how this system reacts to a range of conditions and assumptions

31

General Relativistic Radiative MagnetoHydroDynamics

Using **KORAL**



32

TAKE HOME MESSAGE :

- CFD enables us to predict fluid flow
- The fundamentals of CFD lie in solving the set of partial differential equations that describe the fluid flow (e.g. ***Navier-Stokes equation***)
- In Eulerian grid based approach, the physical domain is discretized into large number of cells
- In each of these cells, Navier-Stokes equations can be rewritten as algebraic equations
- These equations are then solved numerically
- At the end we get the complete description of flow throughout the domain

33

THANK YOU

34

From the electric pulse to image quality

the analysis chain of an imaging detector

Rita Roque

ritaroque@fis.uc.pt

16/03/2020

LIBPhys

Coimbra, Portugal

CERN School of Computing

Genebra, Switzerland

Table of contents

Section 1

Imaging detectors

Incoming radiation ⟶ electric pulse

Imaging detectors: what are they?

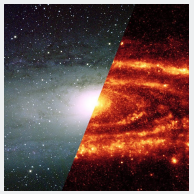
Imaging detector: any instrument that converts incoming radiation into an image.

Radiation type	Radio	Microwave	Infrared	Visible	Ultraviolet	X-ray	Gamma ray
Wavelenght (m)	10^{3}	10^{-2}	10^{-5}	0.5×10^{-6}	10^{-8}	10^{-10}	10^{-12}

Why are they so important? They extend the capabilities of our eyes!

Who doesn't want to be Superman?



Imaging detectors: how do they work?

Imaging detector: any instrument that converts incoming radiation into an image.

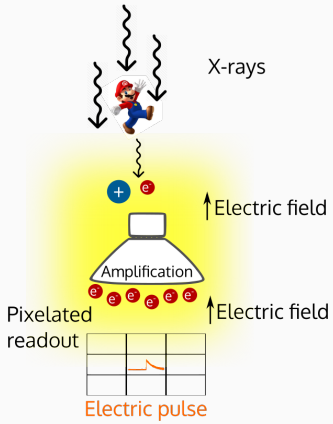
How?

Magic?

Physics and hard work

Let's take the example of **X-ray gaseous imaging detectors**.

Imaging detectors: X-ray imaging gaseous detectors



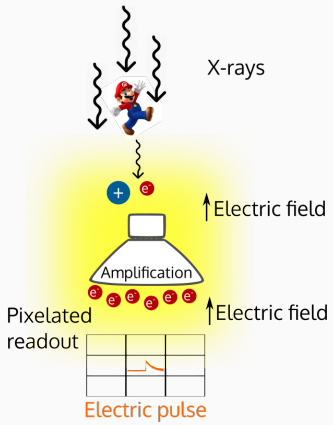
Working principle

1. X-rays are attenuated by the object we want to image;
2. The (noble) gas converts the photons into electrons and ions;
3. We apply an electric field that guides the electrons to an amplification stage;
4. Another electric field guides the final electrons to a pixelated readout.
5. Each electron that arrives generates an electronic pulse.

The number of pulses in each pixel is what codifies the final image.

Imaging detectors: the engineer's work

But physics doesn't work without the engineers.



So many variables to optimize:

- How strong are the electric fields?
- Which gas to use?
- What are the distances between the detector regions?
- Which amplification mechanism to use?

The ultimate goals:

- Maximum charge gain.
- Minimum noise.
- Uniform response.
- The sharpest image.

Imaging detectors: coffee break

Images get better with longer exposure times.

5 min. 10 min 30 min. 1 hour

Imaging detectors: key messages

- **Imaging detectors** convert incoming radiation into an image;
- There are **many types** of imaging detectors, with **different working principles**;
- **X-ray imaging gaseous detectors** convert X-rays into a flow of electrons that produces an electric pulse. **The object is codified by the number of electric pulses produced in each readout pixel**.
- Any imaging detector has **many variables** that we can optimize to get **the best performance possible**.

Section 2

The electronic chain

Electric pulse ⟶ data file

The electronic chain: the detector pulse

This is what happens in each readout pixel.

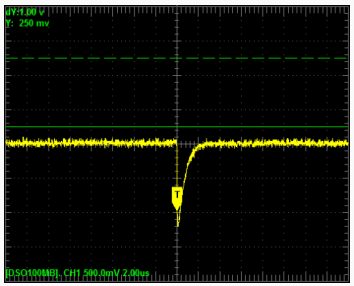


Figure: Detector pulse measured with an oscilloscope.

What can we know from the pulse?

- **Time:** the pulse begins when a cloud of electrons arrives;
- **Radiation energy:** proportional to the area of the pulse.
- **Position:** the pixel of the readout that was triggered.

Engineering challenges

The electronics should determine the time and the radiation energy from this profile as accurately as possible.

The electronic chain: the preamplifier

The preamplifier shapes the detector pulse into a **step function voltage pulse**.
It also **amplifies the signal** to decrease the noise effects.

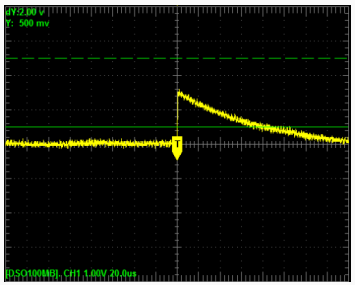


Figure: Preamplifier pulse measured with an oscilloscope.

The perfect preamplifier pulse

- Short rise time;
- Long decay time, but not too long.

Why? Pile-up 

Measurement:

- **Time:** start of pulse.
- **Radiation energy:** proportional to the amplitude of the pulse.

It's easier to calculate the amplitude of the pulse,
but it's still not an accurate and reproducible measure.
The pulse needs a better shape.

The electronic chain: the shaping amplifier

The shaping amplifier shapes the detector pulse into a **quasi-Gaussian**.
It also **amplifies** and **filters high and low frequency noise**.

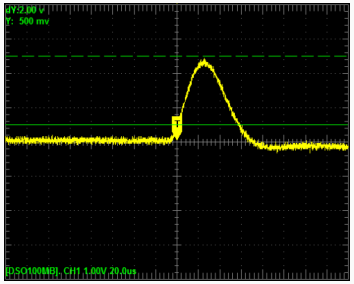


Figure: Shaping amplifier pulse measured with an oscilloscope.

How can we quantify the pulse?

The electronics can evaluate the Gaussian profile and get:

- **Time:** prior threshold definition;
- **Radiation energy:** from the amplitude (proportional to energy).
- **Position:** the pixel of the readout that was triggered.

Now we have accurate values of **time** and **energy**.
These values are digitized (**ADC**) and saved into a file.

The electronic chain: the computer file

Each channel creates a **different file** in the computer with information on the **energy** and **time** of each event.

Each electronic channel corresponds to a detector pixel.

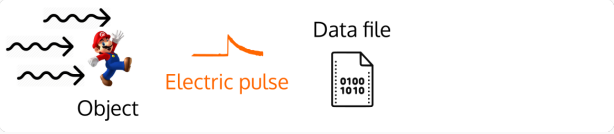
Channel 0	Channel 1	...	Channel n
(Energy, Time)	(Energy, Time)		(Energy, Time)

We need to conjugate all the information in a single file, with **energy** values for each **x**, **y** position.

x	*y*	Energy	(Time)
23245	27044	4622	0000
42360	27505	3511	0010
25944	41019	5242	0013
36118	32834	4325	0018
...	...	...	...

The electronic chain: key messages

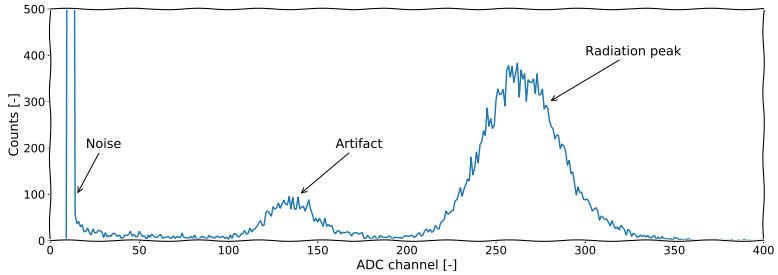
- The output of the detector is an electric pulse with its **area proportional to the radiation energy**;
- A **preamplifier** converts the output pulse into a **step function voltage pulse**, with **amplitude proportional to the radiation energy**;
- The **shaping amplifier** converts the preamplifier pulse into a **quasi-Gaussian**, allowing a reproducible measurement of energy and time;
- For each pixel, the time and energy values of the pulses are **digitized and saved into a file**;
- Every step of the electronic chain introduces an **amplification** and **filters the electronic noise**.



Let's take a break: detectors without position discrimination

What distinguishes non-imaging detectors? There is only one channel!

What information can we get from a single channel?
What is the response of a single pixel in an imaging detector?



The ADC channel of the radiation peak is proportional to the **radiation energy** and the **charge gain**. We can also calculate the **energy resolution** of the detector.

Section 3

Event selection and image reconstruction

Data file ⟶ image

Event selection and image reconstruction: the candidate events

Are all the events recorded in the files valid?

Sometimes we have surprising information that give unexpected errors.

Empty rows

Energy	Time
23245	4622
42360	
NaN	41019
...	...

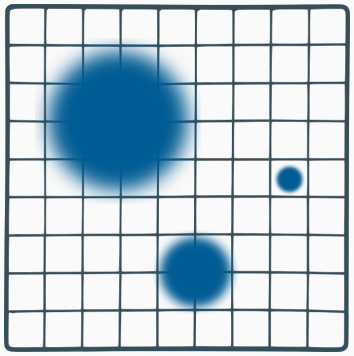
Out-of-range data

Energy	Time
23245	27044
42360	-3511
0	41012
...	...

Some of these problems can be solved easily.

Event selection and image reconstruction: neighbor cells

Sometimes, only one readout channel (pixel) is triggered, which does not make much (physical) sense.



- **Electron clouds are spread** over a certain area;
- This **triggers neighbor cells** simultaneously.
- In a certain time interval, we look for **clusters of triggered cells**.
- Individual triggered cells are probably **noise** or other **electronic artifacts**.

Normally, we **discard** signals that are only recorded in a **single channel**.

Event selection and image reconstruction: energy calibration

The data file looks like this:

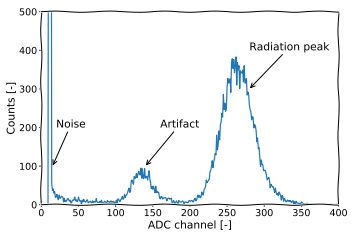
x	*y*	Energy
23245	27044	4622
42360	27505	3511
25944	41019	5242
36118	32834	4325

Can you guess the units?

cm? mm? µm?
J? keV?

We only know that these values are proportional to their real measurement.

The solution: **Calibrate!**



Physics tells us that:

- A ^{55}Fe source has 5.89 keV;
- This artifact has 2.93 keV.

With **extrapolation**:
Energy [ADC unit] ⟶ Energy [Physical unit].

Event selection and image reconstruction: image reconstruction

The reconstructed image is simply a **2D histogram** of the data points.

But what should be the weights?

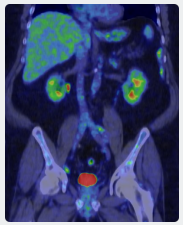
Count wise

All events contribute the same, regardless of their energy value.

Good for object recognition.

Energy wise

More energetic events are favored in the final image.

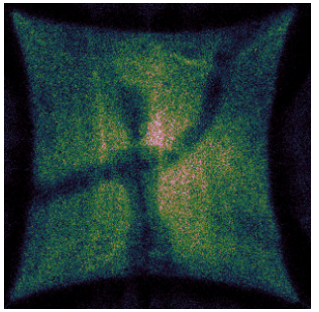


Good to find radioactive materials.

The choice varies from application to application.

Event selection and image reconstruction: applying corrections

In real life detector response is not uniform — it varies with position.



Some examples are:

- Charge gain
- Energy resolution
- Background
- Distortion

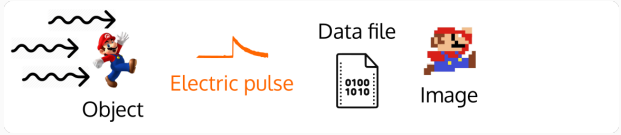
Why?

- Detector geometry
- Electronics

We need to **map the response** of these parameters along the detector's area and then correct every acquired image with a **correction matrix**.

Event selection and image reconstruction: key messages

- **Not all information in the data files are real events**: we have to filter them;
- We usually analyze **triggered neighbor cells** to evaluate a true event;
- Quantities in the data file have to be **calibrated** to have physical meaning;
- The reconstructed image is a **2D histogram** of the true events and can be weighted by **energy** value or by **counts**, depending on the final application.
- After reconstruction, the necessary **correction matrices** should be applied.



Section 4

Image quality assessment

Image ⟶ quality parameters

Image quality assessment: basic parameters

How can we measure image quality?

How much noise?

How many details?

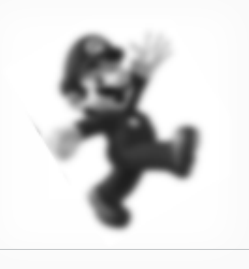
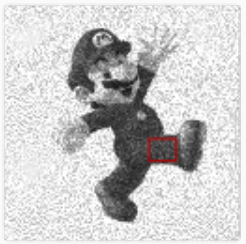


Image quality assessment: Signal-to-noise ratio (SNR)

Noise is a consequence of:

- Statistical fluctuations
- Electronic fluctuations



Measuring SNR

1. Select a **uniform region** of the image.
2. Determine:
   - μ: average pixel value;
   - σ: standard deviation.
3. Calculate

$$SNR = 20 \times \log_{10}\left(\frac{\mu}{\sigma}\right) \quad [\text{dB}]$$

Evaluating SNR

The Rose Criterion:

The amount of noise in an image is appropriate if $SNR \geq 13.98$ dB

Image quality assessment: position resolution

What is the most important thing for an imaging detector to do?

High level of position discrimination – **Resolution!**

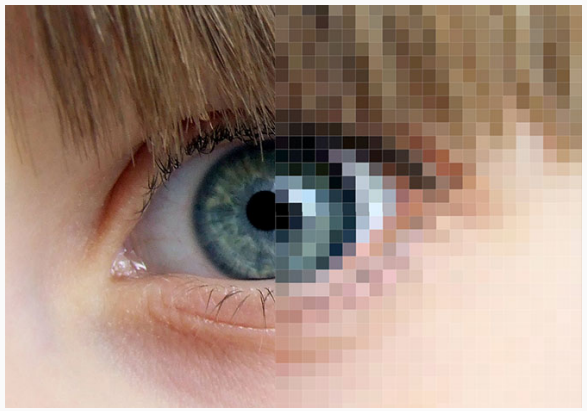
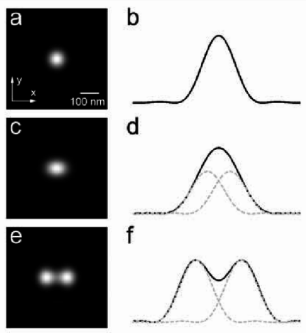


Image quality assessment: what is position resolution?

Position resolution: the minimum required distance between two objects for them to be distinguished by the imaging system.



Is limited by

Readout pixelization;

Physical processes:

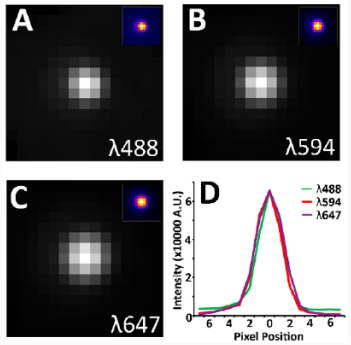
- Type of gas;
- Radiation energy;
- Amplification process:
- ...

Image quality assessment: our intrinsic position resolution

A bunch of well-defined points becomes a well-known picture.

Image quality assessment: how to measure position resolution

Point-spread function (PSF): is the response of the imaging system to an input point source.



Measurement

Usually the PSF can be modeled as a **2D Gaussian function**. The position resolution is equal to the **Full-Width at Half Maximum** in each direction.

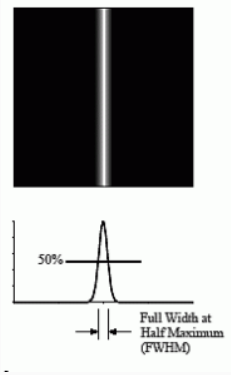
$$FWHM = 2\sqrt{2\ln 2}\sigma$$

Problem

Point-sources are not experimentally feasible.

Image quality assessment: how to measure position resolution

Line-spread function (LSF): is the response of the imaging system to a line stimuli.



Measurement

The LSF can be modeled as a **1D Gaussian** function. Again, the position resolution is equal to the FWHM.

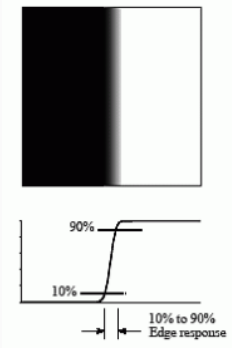
$$\mathrm{LSF}(x) = \int_{-\infty}^{+\infty} \mathrm{PSF}(x,y)\,dy$$

Problem

The line should have an infinitesimal width.

Image quality assessment: how to measure position resolution

Edge-spread function (ESF): is the response of the imaging system to a sharp edge.



Measurement

The ESF can be modeled as a Gauss error function:

$$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2}\,dt$$

$$\mathrm{ESF}(x) = \int_{-\infty}^{x'} \mathrm{LSF}(x)\,dx$$

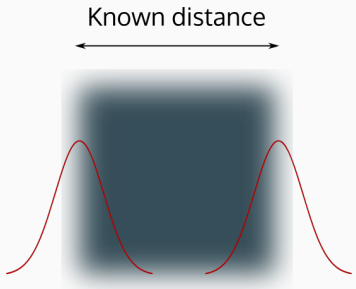
The algorithm — Edge-gradient method

1. Image a sharp edge;
2. Get the ESF
3. Derive to get the LSF and fit to a Gaussian function.

Image quality assessment: position calibration

The x and y values of the image are still in pixels/ADC channels.

We need to translate them into physical units (cm, mm, ...) — **Position Calibration**.

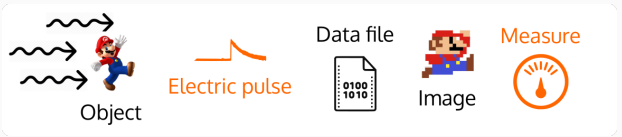


All we need is to image a known distance and get the distance between the two LSFs.

Pixels ⟶ Physical units.

Image quality assessment: key messages

- To quantify the quality of an imaging detector, we have many parameters at hand, like **SNR** and **position resolution**.
- **SNR** is limited by **statistical** and **electronic** fluctuations in the system and should be above 13.98 dB.
- **Position resolution** is limited by **pixelization** and the **physical processes**. It is calculated by the **edge-gradient method**.
- **Position calibration** can be achieved by imaging a known distance.

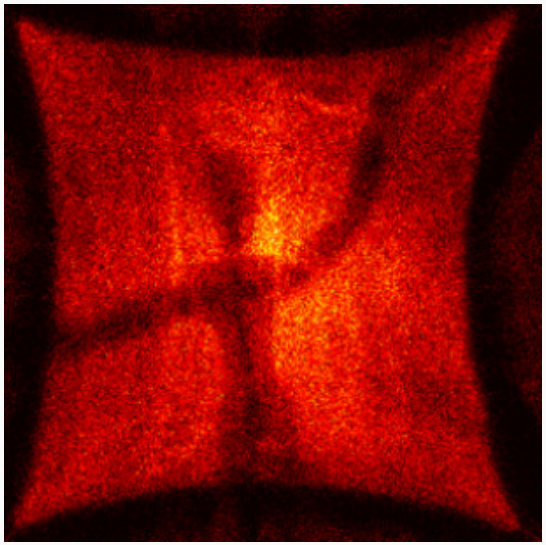


Section 5

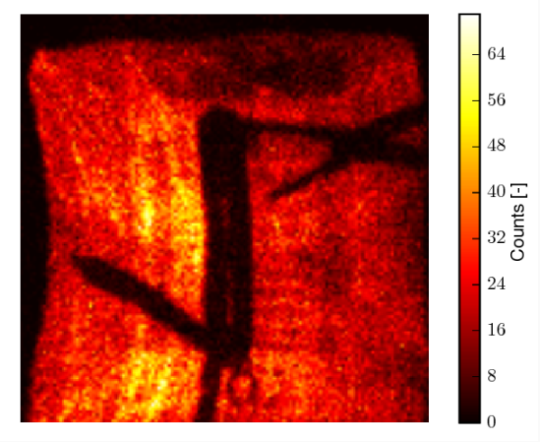
Live Quiz

Go to **live.voxvote.com**
Enter the pin **64256**.

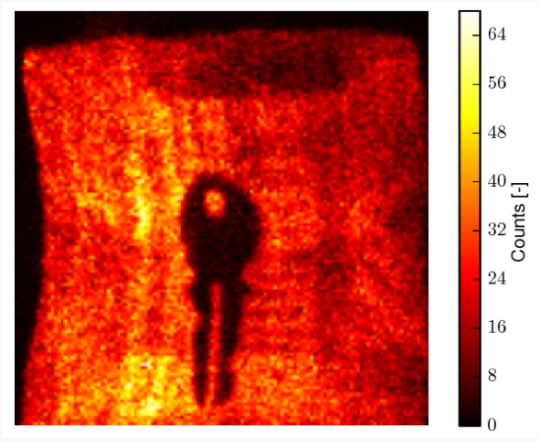
X-ray imaging Can you recognize these objects?



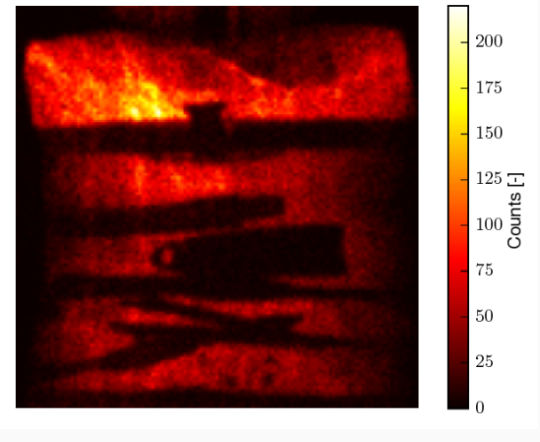
X-ray imaging Can you recognize these objects?



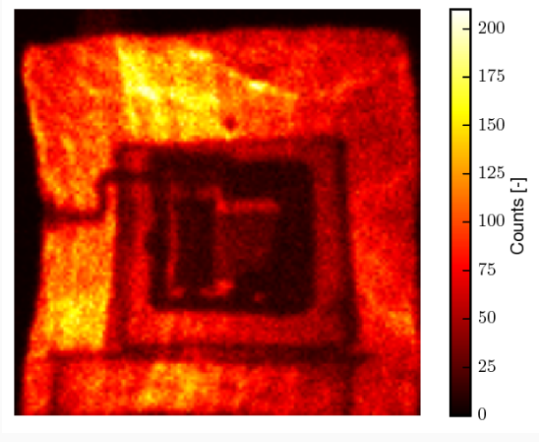
X-ray imaging Can you recognize these objects?



X-ray imaging Can you recognize these objects?



X-ray imaging Can you recognize these objects?



For more information on the
CERN School of Computing:

http://cern.ch/csc