Table of Contents

The LHCb DevelopKit lessons 

These are the lessons taught during the LHCb DevelopKit.

Contributing

starterkit-lessons is an open source project, and we welcome contributions of all kinds:

- New lessons;
- Fixes to existing material;
- Bug reports; and
- Reviews of proposed changes.

By contributing, you are agreeing that we may redistribute your work under these licenses. You also agree to abide by our contributor code of conduct.

Getting Started

1. We use the fork and pull model to manage changes. More information about forking a repository and making a Pull Request.
2. To build the lessons please install the dependencies.
3. For our lessons, you should branch from and submit pull requests against the `master` branch.
4. When editing lesson pages, you need only commit changes to the Markdown source files.
5. If you're looking for things to work on, please see the list of issues for this repository. Comments on issues and reviews of pull requests are equally welcome.

Dependencies

To build the lessons locally, install the following:

1. Gitbook

Install the Gitbook plugins:

```
$ gitbook install
```

Then (from the starterkit-lessons directory) build the pages and start a web server to host them:

```
$ gitbook serve
```

You can see your local version by using a web-browser to navigate to `http://localhost:4000` or wherever it says it's serving the book.

First Development Steps

This is the LHCb DevelopKit, a workshop/lesson series covering development in LHCb software, focused on the upgrade for Run 3.

If you have any problems or questions, you can send an email to `lhcb-starterkit@cern.ch`.

Prerequisites

You should have completed StarterKit, and optionally ImpactKit. Before starting, you should have lesson series 1 completed, as they deal with setup, and take a fair amount of time. Select one that best suits your development environment; for upgrade work, you will want a complete environment.

Setting up a development environment

There are several options for setting up a development environment. Three different options for preparing an environment are discussed in the following three lessons.

Setting up a CernVM

Using CernVM

- Download and prepare a CernVM

To get a virtual machine setup, follow the instructions at the CernVM website. Since there are quite a few choices listed, here is a recommended method for setting up a CernVM.

In order to use a virtual machine, you will need to create a context. Select LHCb as your main group, and make sure you pay attention to your user settings; that's how you will log in later. Under CernVM preferences, you'll want to look at the CernVM Edition variable, and set it to desktop if you want a UI.

Get VirtualBox in order to run the VM. Download CernVM and follow the manual installation instructions to get a local install. Spawning from online seems to have persistence issues, and the OVA image seems to have some hardware options baked in that might not match your computer.

Next you will need to pair your instance with your context. You'll get a context code that you'll enter as your "username" when booting up your instance. Make sure you enter a number sign preceding the pairing number, or it will try to log into a user with your number instead. After it pairs, you can use the username and password that you previously specified.

Using a CernVM

You will need to run the setup script `source /cvmfs/lhcb.cern.ch/group_login.sh` to prepare an LHCb environment.

Setting up a Docker development environment

Using Docker

- Download and prepare a Docker

The following setup is based on the Hackathon setup here.

Prerequisites

- Docker: you should be able to run Docker containers. On MacOS you should have the latest version of Docker, which finally has built in virtualization. Be sure to use the most recent version of `upgrade-hackathon-setup`, as well, as several Mac related issues were fixed, and Mac is now fully supported with no (known) caveats. Also ensure that you're logged in on Docker by running the command `docker login`, supplying your Docker ID (username, **not** email address!) and password.
- Fuse: You should have a very recent version of Fuse (or OSXFuse on MacOS)
- CVMFS: you should have access to /cvmfs/lhcb.cern.ch. On a Mac you will need OSXFuse and CernVM-FS.

Mounting CVMFS manually

If you want to mount CVMFS manually, such as for accessing it for other tasks, this is how you can do that:

Set your proxy, for example:

```
local:~ $ echo "CVMFS_HTTP_PROXY=DIRECT" | sudo tee -a /etc/cvmfs/default.local
```

Mount a CernVM-FS repository using the following steps:

```
local:~ $ sudo mkdir /cvmfs/lhcb.cern.ch
local:~ $ sudo mount -t cvmfs lhcb.cern.ch /cvmfs/lhcb.cern.ch
local:~ $ sudo mkdir /cvmfs/lhcbdev.cern.ch
local:~ $ sudo mount -t cvmfs lhcbdev.cern.ch /cvmfs/lhcbdev.cern.ch
```

If you are on Mac, you will also need to add the `/cvmfs` directory to the shared directory list in the Docker applet.

Quick start

To get started, get the tools with:

```
local:~ $ git clone ssh://git@gitlab.cern.ch:7999/lhcb/upgrade-hackathon-setup.git hackathon
```

Using your ssh key inside the container

If you have loaded your identity, you can share to the container. To load it, type:

```
local:~ $ test -z "$SSH_AGENT_PID" && eval $(ssh-agent)
local:~ $ ssh-add
```

This is also useful, since it keeps you from having to type a passphrase every time you try to use git. The `test -z` command is just ensuring that you don't run the agent multiple times. If you are using multiple identities on one account, you can tell `ssh-add` to add the correct one.

Then from the `hackathon` directory just created invoke:

```
local:hackathon $ ./lb-docker-run -c x86_64-centos7-gcc7-opt
```

which will pull the latest image of the CentOS7 Docker image we use to build our software and start an interactive shell in the special directory `/workspace`, mapped to the local `hackathon` directory.

Note about the options of the `lb-docker-run` command

This command has quite a few options. You can see them with:

```
local:hackathon $ ./lb-docker-run -h
```

The persistent home option used above may require that you own `/<username>` on your system as well.

The `--ssh-agent` option forwards your ssh identity to the container (if one was added). You can also try the `--wrap` option to set up defaults differently, and to run in the same directory instead of `/workspace`.

Building the whole stack the first time may take a lot of time, so you can optionally get a pre-built image with:

```
local:hackathon $ make pull-build
```

At this point you can build the software stack with (will build automatically in parallel):

```
local:hackathon $ make
```

If you didn't pull the pre-built image, this command will checkout the code and build from scratch.

Using CVMFS

Using CVMFS requires a few changes compared to working on LxPlus, where you have access to AFS. If you run an `lb-dev` or `lb-run` command on the nightlies, you should add `--nightly-cvmfs` to the command. For example:

```
local:hackathon $ lb-run --nightly-cvmfs --nightly lhcb-head Kepler HEAD $SHELL
```

Would drop you into a shell with the CVMFS nightly head of the Kepler package.

Building a project

This is how you would run MiniBrunel:

```
local:~ $ . /cvmfs/lhcb.cern.ch/lib/lhcb/LBSCRIPTS/dev/InstallArea/scripts/LbLogin.sh -c x86_64-slc6-gcc49-opt
local:~ $ lb-dev --nightly-cvmfs --nightly lhcb-future 282 Brunel/future
local:~ $ cd BrunelDev_future
local:BrunelDev_future $ git lb-use -q Brunel
local:BrunelDev_future $ git lb-use -q Rec
local:BrunelDev_future $ git lb-checkout Brunel/future Rec/Brunel
local:BrunelDev_future $ make
local:BrunelDev_future $ ./run gaudirun.py Rec/Brunel/options/MiniBrunel.py
```

Setting up a local LHCb environment - CentOS 7

Using a local copy of CentOS 7

- Get and configure the LHCb stack.

On CentOS 7, the following steps allow you to build the LHCb software:

First, mount CVMFS. It is obtainable through yum with:

```
local:~ $ sudo yum install https://ecsft.cern.ch/dist/cvmfs/cvmfs-release/cvmfs-release-latest.noarch.rpm
local:~ $ sudo yum install cvmfs cvmfs-config-default
local:~ $ sudo cvmfs_config setup
```

Open the file `/etc/cvmfs/default.local` and add the lines:

```
CVMFS_HTTP_PROXY=DIRECT
CVMFS_QUOTA_LIMIT=25000
```

Verify these are available with `cvmfs_config probe`. If not, restart with `sudo service autofs restart`.

You also may need uuid on your system:

```
local:~ $ sudo yum install uuid-devel libuuid-devel
```

You can run on CentOS 7 using:

```
local:~ $ source /cvmfs/lhcb.cern.ch/group_login.sh -c x86_64-centos7-gcc62-opt
```

It is still recommended that you use the Upgrade-hackathon-setup, and you can manually run:

```
local:upgrade-hackathon-setup $ . setup.sh
```

from that directory. You can now run `make -j1`, along with all the other make commands available to you. See Upgrade-hackathon-setup.

Git for LHCb

Most of LHCb's code has been moved to git using CERN's GitLab instance. There are two options for using git in LHCb. One is an "easy" method, which is provided with the intent to mimic the old SVN procedures. It provides the fastest builds when you are just working on one piece of LHCb software, but does not handle standard git workflows very well, making it difficult to use with larger projects.

The second method is using pure git; while it takes a little longer to setup and build, you'll be able to use git tools to manipulate branches, pull and push, and more; this is usually preferred by developers.

Setup Simple

Setting up an LHCb package

- Prepare a package and check out a portion of it.

Prerequisites for building LHCb software

You should have a valid LHCb environment, using one of the ways previously mentioned, or by working on `lxplus`. You may need to run `source /cvmfs/lhcb.cern.ch/group_login.sh` to set up your environment variables.

Git

LHCb uses the git versioning system.

Prerequisites for git

Before using git, you should go to your gitlab.cern.ch account and set up an ssh key. The procedure is the same as with github (and the public key is the same). In short, on Linux or Mac, make sure you have key pair in `~/.ssh/id_rsa.pub` and `~/.ssh/id_rsa`. If you don't have one, run

```
local:~ $ ssh-keygen
```

and use the default location. If you don't put in a passphrase, you can directly push and pull without a password; if you do put a passphrase, you can run

```
local:~ $ test -z "$SSH_AGENT_PID" && eval $(ssh-agent)
local:~ $ ssh-add
```

when you start a terminal to enter your passphrase once per session. Your computer keychain (Mac, Ubuntu, etc) can also store your password once per login.

If you encounter problems

If you get an error about your ssh agent not running, you can execute the output of the ssh-agent command with `eval $(ssh-agent)` and try again.

Once you verify you have an ssh key pair, you need to go to the gitlab website, go to your profile (possibly hidden behind the hamburger button), then edit, then SSH Keys (or just go here), and paste the contents of the `id_rsa.pub` file into the key edit box. You can give it a title that allows you to identify the computer or VM associated with the key.

Defaults

You can set the lb-commands to default to ssh by running the following line to save a global (all repository) config variable:

```
local:~ $ git config --global lb-use.protocol ssh
```

You can also set the git default push policy for git if you have a version between 1.7.11 and 2.0:

```
local:~ $ git config --global push.default simple
```

This is not mandatory, but is a more reasonable default (and is the default in git 2.0. Note that SLC6 systems, like lxplus, have a version less than 1.7.11. On these systems, you can still set the default to `current`, which is better than the original default of `matching`.

Aside: Explanation of defaults for the curious

This aside is will only make sense if you know a little bit about git. If you are working on a branch `local_branch` and you push without explicitly telling git what to do, it will do the following:

- Upstream: Push to the upstream branch (must set an upstream branch), even if the name does not match the current branch.
- Simple: If you have an upstream branch set, *and* it has the same name, it will push to that. Why is this a desired behavior? Because you may make a new branch but forget to change the upstream branch, so you could end up pushing your new branch to a differently named remote branch! (Default in git 2.0+)
- Current: Push to a branch with the same name on the remote.
- Matching: push all branches with the same names as remote branches to the remote. Great way to make a mess by pushing branches you did not intend. (Default in git<2.0).

Coming from SVN?

If you have a background with SVN, you might want to look at the two part tutorial from IBM for subversion users, Part 1 and Part 2.

LHCb simplifications

While vanilla git commands are covered in the alternate lesson, this lesson uses special LHCb commands added to git. These are special programs that sit in the LHCb bin, and have names that start with `git-`. Git automatically converts a command like `git x` to `git-x`, so these seamlessly blend with the built in commands. To differentiate them, they all start with `lb-`.

Setting up a satellite project

If you want to setup a satellite project (local project) for small changes, you'll need to prepare that project:

```
local:~ $ lb-dev Project vXrY
local:~ $ cd ProjectDev_vXrY
```

For example, the project might be `DaVinci`, and the version might be `v40r3p1`. Then, you'll been to tell git where it can checkout subprojects from:

```
local:ProjectDev_vXrY $ git lb-use Project
```

This sets up a remote that points to the LHCb repository Project on GitLab.

To get the code from Project/Branch to checkout into Some/Package:

```
local:ProjectDev_vXrY $ git lb-checkout Project/Branch Some/Package
```

The branch can be `master`, or a specific version `vXrY`, or any other valid branch. See the GitLab page to browse projects and branches.

Future reading

The twiki page Git4LHCb is currently the best source for git examples.

Setup Complete

Setting up using pure git

- Set up a development environment.

Prerequisites for building LHCb software

You should have a valid LHCb environment, using one of the ways previously mentioned. You may need to run `source /cvmfs/lhcb.cern.ch/group_login.sh` to set up your environment variables.

Before building, there are a few recommended customisations to prepare in your environment, besides those discussed in the previous lesson.

Ninja (optional, faster replacement for make)

If you have ninja in your path, it will be used to (marginally) speed up the make process, and it is automatic; you still use `make ...` commands. To add it:

```
local:~ $ export PATH=/cvmfs/lhcb.cern.ch/lib/contrib/ninja/1.4.0/x86_64-slc6:$PATH
```

You can also add:

```
local:~ $ export VERBOSE=
```

if you are a fan of reduced noise when building.

CCache (optional, faster rebuilds)

If you rebuild often, you should be using CCache. This program stores the results of compiling and reuses them if nothing changes. You should be able to do something like this after you install CCache:

```
local:~ $ export CCACHE_DIR=$HOME/.ccache
local:~ $ export CMAKEFLAGS=-DCMAKE_USE_CCACHE=ON
```

If you installed CCache from source locally to `$HOME/.local` , then you'll need the local bin in your path, too:

```
local:~ $ export PATH=$HOME/.local/bin:$PATH
```

Debug builds

If you want a debug build, you'll need to change some environment variables with the LbLogin program, which will start a new bash instance with the required variables using the configuration name in `$CMTDEB`:

```
local:~ $ LbLogin -c $CMTDEB
```

Path

You should have a main development directory, and it should be in your CMTPROJECTPATH variable. If you use `$HOME/lhcbdev`, then you would do:

```
local:~ $ export CMTPROJECTPATH=$HOME/lhcbdev:$CMTPROJECTPATH
```

In VMs and Docker, sometimes the root directory `/workspace` is used.

Git for LHCb

You can set up a development environment with complete LHCb packages, and build them from scratch. This allows you to make large scale changes without locating sub-packages online, and is similar to the standard procedures in other projects. The downside to this is that they take a while to compile, due to the size of LHCb projects. Since the following is very similar to standard git procedures, most online git tutorials are also helpful in understanding the meaning of the various git command used.

Help for the git novice

There are a lot of sites available for help with vanilla git commands. A few are listed here.

1. The git documentation.
2. Github's interactive tutorial series. This is an excellent resource for running simple commands to get a feel for how git works on the command line, right in your browser.
3. Atlassian's git guru tutorials.
4. TutorialsPoint's git tutorials.

There are always cheatsheets, too, such as this one. A unique cheatsheet reference from one of the LHCb collaborators can also be downloaded here.

Before using git, you should go to your gitlab.cern.ch account and set up an ssh key. The procedure is the same as with github (and the public key is the same). In short, on Linux or Mac, make sure you have key pair in `~/.ssh/id_rsa.pub` and `~/.ssh/id_rsa`. If you don't have one, run

```
local:~ $ ssh-keygen
```

and use the default location. If you don't put in a passphrase, you can directly push and pull without a password; if you do put a passphrase, you can run

```
local:~ $ ssh-add
```

when you start a terminal to enter your passphrase once per session. Your computer keychain (Mac, Ubuntu, etc) can also store your password once per login.

If you encounter problems

If you get an error about your ssh agent not running, you can execute the output of the ssh-agent command with `eval $(ssh-agent)` and try again.

Once you verify you have an ssh key pair, you need to go to the gitlab website, go to your profile (possibly hidden behind the hamburger button), then edit, then SSH Keys (or just go here), and paste the contents of the `id_rsa.pub` file into the key edit box. You can give it a title that allows you to identify the computer or VM associated with the key.

Setting up the projects

To get a complete project for development, the following command will work:

```
local:~ $ git clone ssh://git@gitlab.cern.ch:7999/lhcb/Project.git
```

where `Project` is the name of the project. The location of this project on your computer is important! It should have the following structure:

- Main LHCb folder (any name, like `$HOME/lhcbdev`)
  - `Project` (git folder here)
  - `AnotherProject`

The package structure is this way to allow the package search system to find names and versions. The main folder should be listed in the `$CMTPROJECTPATH` environment variable.

Once you have cloned the repository, you should check out the branch that you want to build/work on with `git checkout -b BranchName` .

An complete example for the Lbcom package would look like this, using a destination for the git folder, since it will default to the wrong name:

```
local:~ $ mkdir ~/lhcbdev
local:~ $ cd ~/lhcbdev
local:lhcbdev $ git clone ssh://git@gitlab.cern.ch:7999/lhcb/LHCb.git
local:lhcbdev $ cd LHCb
local:LHCb $ git checkout -b upgradeTracking
```

Inside the project, the CMakeLists.txt file may reference other projects, like this:

```
gaudi_project(Project LocalVersion
              USE ReferredToProject RemoteVersion
              DATA OtherStuff)
```

You need to set the LocalVersion to your version (such as upgradeTracking), and you need to set any dependencies to the version names you've set, as well. When `gaudi_project` looks for a package, it will look for ones that follow the above naming scheme and have the correct, matching version.

> Note: You do not need to locally build all dependencies if there is a released version that will work; they will be found if you do not have a local version.

Building

To build, you need to run the following commands in each package directory (starting with dependencies first):

```
local:LHCb $ lb-project-init
```

This will setup the makefile, and some other things. Then run:

```
local:LHCb $ make configure
```

to configure the run (it runs CMake behind the scenes). If this fails, you probably have a problem in your CMakeLists.txt or with your paths.

To build and move the results to the folder that CMT expects, run:

```
local:LHCb $ make install
```

This should take a long time the first time, and should be much faster after that.

Future reading

The twiki page Git4LHCb is currently the best source for git examples.

Building LHCb software

Understand how to modify builds.

- "Learn the basics of building software in LHCb."

Building in CMake

Modern LHCb software builds is stored in git and built with CMake. The procedure to build is given in the example below:

Customizing the build

The build mechanism requires two files in the main directory:

- `toolchain.cmake` : A standard file generated by the steps above (`lb-project-init`)
- `CMakeLists.txt` : The cmake build instructions.

In your CMakeLists.txt, instead of manually configuring CMake, you generally have the following lines:

```
cmake_minimum_required(VERSION 2.8.5)
```

This tells CMake to set policies to the version listed; even if you are in a newer CMake, you will be fully backwards compatible (and will not receive many of the improvements of using a newer CMake, either).

```
find_package(GaudiProject)
```

This is a custom package that contains the very powerful `gaudi_project` command used below. This is CMake's "import" statement. Other common packages are ROOT and Boost. You can add components, using the COMPONENTS keyword. (ROOT's components are not very well defined, but this is useful for BOOST).

```
gaudi_project(MyProject v12r3
              USE BaseProject v45r6
                  AnotherProject v7r8
              DATA Some/DataPackage
                   ExternalData VERSION v2r*
                )
```

Notice the structure of this command. First we have the current project name, and it's version. Then we use the keyword "USE" defined by the function to start listing projects to use. Here we have to specify a specific version. Finally, the keyword "DATA" starts a listing of data packages, which can take wild cards in the version strings.

Internally, this tells CMake to find and load other projects in a specific directory structure.

Gaudi supplies a few other commands, as well.

WIP

Later, explain:

- `gaudi_add_library`
- `gaudi_depends_on_subdirs`
- `gaudi_add_module`
- `gaudi_install_headers`
- `god_build_dictionaries`
- `god_build_headers`
- `gaudi_install_python_modules`
- `gaudi_install_scripts`
- `gaudi_install_tests`
- `gaudi_add_dictionary`
- `gaudi_add_executable`
- `gaudi_env`

Further reading

- CMake LHCb builds TWiki

Running and testing

Run examples and tests

- "Learn about running examples."
- "Learn about running tests."

One of the most important components of writing code is testing. The following section will show you how to run examples in tests in Gaudi; this is also mostly compatible with the packages that are built on top of Gaudi, such as DaVinci and Gauss.

Running an example

The Gaudi examples are available in `Gaudi/GaudiExamples`. On lxplus, you can grab them and build with a partial checkout:

```
local:~ $ lb-dev Gaudi v28r1
local:~ $ cd GaudiDev_v28r1
local:GaudiDev_v28r1 $ git lb-use Gaudi
local:GaudiDev_v28r1 $ git lb-checkout Gaudi/v28r1 GaudiExamples
local:GaudiDev_v28r1 $ make
```

Now, you have Gaudi and just the GaudiExamples subdirectory (project). This is a good place to start looking around to see how Gaudi works; for now we'll try running an example.

```
local:GaudiDev_v28r1 $ ./run gaudirun.py GaudiExamples/options/AlgSequencer.py
```

This will run a a few algorithms in sequence, and will output info messages for ten "events" (no data).

Testing

Searching the source

In this case, you can search the source in the current directory and below by:

```
local:~ $ git grep AlgSequencer
```

To search the online source, you can do:

```
local:~ $ Lbglimpse AlgSequencer Ganga v28r1
```

There are two methods to write tests:

QMT tests

For an old-style QMT test, you add a QMT file in `tests/qmtest/gaudiexamples.qms/` that looks like `testname.qmt`. The file contains

xml:

```
 <?xml version="1.0" ?><!DOCTYPE extension  PUBLIC '-//QM/2.3/Extension//EN'  'http://www. codesourcery.com/qm/dtds/2.3/-//qm/
2.3/extension//en.dtd'>
 <extension class="GaudiTest.GaudiExeTest" kind="test">
   <argument name="program"><text>gaudirun.py</text></argument>
   <argument name="args"><set><text>$GAUDIEXAMPLESROOT/options/AlgSequencer.py</text></    set></argument>
   <argument name="use_temp_dir"><enumeral>true</enumeral></argument>
   <argument name="reference"><text>refs/AlgSequencer_pyopts.ref</text></argument>
 </extension>
```

This contains a few important points. The `AlgSequencer.ref` is a copy of the output that will be used to check the result. It is at `tests/refs/AlgSequencer_pyopts.ref`. The compare is smart enough to mask out numbers like time and date for you. The opts file to run is also here.

Python tests

Another way to create a test is through a python system; a matching example is at `tests/qmtest/newFormat/algsequencer_pyopts_test.py`. The contents:

```
# -*- coding: utf-8 -*-
import BaseTest
from BaseTest import *

class Test(BaseTest):

    def __init__(self):
        BaseTest.__init__(self)
        self.name = os.path.basename(__file__)[:-5]
        self.program="gaudirun.py"
        self.args=["$GAUDIEXAMPLESROOT/options/AlgSequencer.py"]
        self.reference = "refs/AlgSequencer_pyopts.ref"
```

Running a test

You can run all the tests with `make test`. This internally runs the ctest command in the build directory. You can pass on arguments to the ctest file through the make command though the `ARGS` variable. For example, to run a single named test:

```
local:GaudiDev_v28r1 $ make test ARGS="-R algsequencer_pyopts"
```

The arguments that can be passed to `ctest` can be found with `ctest --help`. Some examples:

- `-N` : just print the names of the tests
- `-R <regex>` : select tests by regular expression
- `-V` : verbose printout (extra details, command line)

Note that `gaudirun.py` can take read and run a `.qmt` file directly. This allows you to see the output of the run, as `ctest` hides the output.

See also

- Hackathon introduction

Hello World in the Gaudi Framework

Gaudi in LHCb

- "Learn how to write a simple test Gaudi program."

Hello World in Gaudi

The following is a minimum example to build an algorithm in Gaudi as a new package. This code is part of this repository, in `/code/GaudiMinimal`. If you don't have this repository cloned already, the following line will just get the project you need for this lesson:[1]

```
local:~ $ svn export https://github.com/lhcb/starterkit-lessons/trunk/first-development-steps/code/GaudiMinimal GaudiMinimal
```

Project

First, you will need a Gaudi **project**. The example was setup to match the way that most Gaudi projects work, with **packages** inside. This is the recommended directory structure for this example:

```
└── GaudiMinimal
    ├── CMakeLists.txt
    └── GaudiHelloWorld
        ├── CMakeLists.txt
        ├── src
        │   ├── HelloWorld.cpp
        │   └── HelloWorld.h
        └── options
            └── gaudi_opts.py
```

Here is an example top-level `CMakeLists.txt` for a new LHCb package:

```
cmake_minimum_required(VERSION 2.8.5)

find_package(GaudiProject)

gaudi_project(GaudiMinimal v1r0
              USE Gaudi v27r1)
```

This first line sets the CMake version, and the second loads the GaudiProject CMake module. Then a new project (`GaudiMinimal`) is declared, and Gaudi is set as a dependency. Gaudi automatically looks for directories in the current one for packages. This is done because it makes it simple to grab a few packages, and they are automatically picked up by the project, and if the rest of the project is built and ready elsewhere in the path, the local and remote portions are combined. This makes it easy to build a small piece of a project with rebuilding the entire project, or add a piece to a project.

Note about organization

A further organizational tool are **project groups**; which are simply one more layer of folders. We don't need it for this project, but to add it is as simple as adding one more directory to the path. No extra CMakeLists are needed. It is commonly used to factor out common code between multiple projects, with the subprojects living in separate git repositories. For an often out of date but useful

description, see this TWiki page.

An example of this would be writing an Algorithm for DaVinci; DaVinci is the project, Phys is the subproject, and SomeDaVinciPackage could be the package you are working on, and is the only piece you need to clone in git. The code lives in the Phys git repository, which means it could be used by other projects too.

The package needs to be created in the `GaudiHelloWorld` subdirectory, and has a CMakeLists.txt that looks like this:

```
cmake_minimum_required(VERSION 2.8.5)

gaudi_subdir(GaudiHelloWorld)

gaudi_add_module(GaudiHelloWorld src/*.cpp
                 LINK_LIBRARIES GaudiAlg)
```

This is tagged as a Gaudi subdirectory. The module we are making is added as `GaudiHelloWorld`, and linked to the Gaudi Kernel. More advanced algorithms may need to be linked to more libraries, including some that are discovered by `find_package`.

The header file

To create an algorithm, the following header file is used:

```
#pragma once

#include "GaudiAlg/GaudiAlgorithm.h"

class HelloWorldEx : public GaudiAlgorithm {
public:
    HelloWorldEx(const std::string& name, ISvcLocator* pSvcLocator);
    StatusCode initialize() override;
    StatusCode execute() override;
    StatusCode finalize() override;
};
```

This creates a new algorithm, and overrides all five of the user-accessible functions. Many algorithms will not need to override all of these. The constructor is needed primarily to delegate to the Algorithm constructor, and is also needed if you want to use it to initialize member variables.

Note on inheriting constructors

If you don't need to add anything to the constructor, you can inherit the default constructor by replacing the constructor line above with:

```
using GaudiAlgorithm::GaudiAlgorithm;
```

And if all you need a non-default constructor for is to initialize member variables, that can be done in the inline in their definition instead.

The implementation

The implementation is simple:

```
#include "HelloWorldEx.h"
#include "GaudiKernel/MsgStream.h"

DECLARE_COMPONENT(HelloWorldEx)
```

Here, we include our header file and use the `DECLARE_COMPONENT` macro to register this as a Gaudi factory in Gaudi's Registry singleton.

```
HelloWorldEx::HelloWorldEx(const std::string& name, ISvcLocator* ploc) :
    Algorithm(name, ploc) {
    }
```

The only requirement for the constructor is to pass on the two arguments to the Algorithm constructor, written here with C++ forwarding syntax. Any code here will be run when the C++ object is created, so can be used for local initialization. If you don't need a constructor, be sure you delegate to the base class constructor, either explicitly as is done here or with the `using` statement noted above.

```
StatusCode HelloWorldEx::initialize() {

  StatusCode sc = Algorithm::initialize();
  if(sc.isFailure() ) return sc;

  info() << "Hello World: Inilializing..." << endmsg;
  return StatusCode::SUCCESS;
}
```

This is an optional initialization method. If run, it should first call the base class's initialize method, and return the status code if it was not successful. After that, any initialization code you may need can be added. In this example, we are simply printing a message.

```
StatusCode HelloWorldEx::execute() {
  info() << "Hello World: Executing..." << endmsg;
  return StatusCode::SUCCESS;
}
```

This is the execute method. It will run once per event. This usually is the most important method, and ideally is the only one present. Most of the other methods break the concept of stateless algorithms, making this hard to move into a multithreaded framework.

```
StatusCode HelloWorldEx::finalize() {
  info() << "Hello World: Finalizing..." << endmsg;
  return Algorithm::finalize(); // must be executed last
}
```

This is the final method. If you do implement it, you should end by passing on to the base class finalize method.

Performing the run

An example `gaudi_opts.py` options file that uses our algorithm:

```
from Gaudi.Configuration import *
from Configurables import HelloWorldEx

alg = HelloWorldEx()

ApplicationMgr(
    EvtMax = 10,
    EvtSel = 'NONE',
    HistogramPersistency = 'NONE',
    TopAlg = [alg],
)
```

We first import from the magic `Configurables` module, which contains our algorithm(s). The `EvtSel = 'NONE'` statement sets the input events to none, since we are not running over a real data file or detector. We then set up the Gaudi application manager, with the settings

that we need and with a list of algorithms, which in this case is just one. We could also append algorithms to `.TopAlg` on the application manager instance.

To run, the following commands can be used on LXPlus:

```
local:~ $ cd GaudiMinimal
local:GaudiMinimal $ lb-project-init
local:GaudiMinimal $ make
local:GaudiMinimal $ ./build.x86_64-slc6-gcc49-opt/run gaudirun.py GaudiHelloWorld/options/gaudi_opts.py
# setting LC_ALL to "C"
# --> Including file '/afs/cern.ch/user/w/whoami/tmp/GaudiMinimal/GaudiHelloWorld/options/gaudi_opts.py'
# <-- end="" of="" file="" '="" afs="" cern.ch="" user="" w="" whoami="" tmp="" gaudiminimal="" gaudihelloworld="" options=""
gaudi_opts.py'<="" span="">
ApplicationMgr    SUCCESS
================================================================================
                     Welcome to GaudiMinimal version v1r0
                running on lxplus001 on Fri Nov 11 15:43:06 2016
================================================================================
ApplicationMgr       INFO Application Manager Configured successfully
HelloWorldEx         INFO Hello World: Inilializing...
EventLoopMgr      WARNING Unable to locate service "EventSelector"
EventLoopMgr      WARNING No events will be processed from external input.
HistogramPersis...WARNING Histograms saving not required.
ApplicationMgr       INFO Application Manager Initialized successfully
ApplicationMgr       INFO Application Manager Started successfully
HelloWorldEx         INFO Hello World: Executing...
HelloWorldEx         INFO Hello World: Executing...
HelloWorldEx         INFO Hello World: Executing...
HelloWorldEx         INFO Hello World: Executing...
HelloWorldEx         INFO Hello World: Executing...
HelloWorldEx         INFO Hello World: Executing...
HelloWorldEx         INFO Hello World: Executing...
HelloWorldEx         INFO Hello World: Executing...
HelloWorldEx         INFO Hello World: Executing...
HelloWorldEx         INFO Hello World: Executing...
ApplicationMgr       INFO Application Manager Stopped successfully
HelloWorldEx         INFO Hello World: Finalizing...
EventLoopMgr         INFO Histograms converted successfully according to request.
ApplicationMgr       INFO Application Manager Finalized successfully
ApplicationMgr       INFO Application Manager Terminated successfully
```

You should be able to see your algorithm running, and the output at the various points.

Further reading

There are further examples in the Gaudi repository: Gaudi/GaudiExamples/src.

[1]. Yes, this is checking out GitHub repository. With SVN. ↩

Intro to the Gaudi framework

Learn about Gaudi

- "Learn the basic concepts of Gaudi."

What is the Gaudi framework?

The Gaudi framework runs over a list of events, providing ways to process them and store data in a new format. It creates and manages **Data Objects**, which can hold a variety of data. A **Transient Event Store (TES)** stores data objects in a way to make them accessible to the rest of the framework, and parts of it can be made persistent in a ROOT format file. The data in the TES is created and accessed by **Algorithms**, which produce data objects and process data objects. Gaudi also provides **Services**, which provide access to other parts of the framework, such as histograms. **Tools** are lightweight routines that are also available. The Application Manager manages these components.

Algorithms

This is the most important component of the framework for an user to know. Algorithms are called once per physics event, and (traditionally) implement three methods beyond constructor/destructor: `initialize`, `execute`, and `finalize` (see the details on the upgrade). Also, `beginRun` and `endRun` are available, though be careful not to misuse state.

Properties

Algorithms are a **Configurable**, which means they can be accessed in Python and **Properties** can be manipulated there. In the classic API, a property is declared in the *constructor*, using:

```
declareProperty("PropertyName", f_value, "Description of property");
```

Here, `f_value` is a reference to a variable for an int, string, etc. It is almost always a member variable for the class so that you can access it in the other methods.

To use a property, you can simply access it on the configurable in Python:

```
my_algorithm.PropertyName = 42
```

Data Objects

To place an item on the TES, it must inherit from `DataObject` (in the classic system) and must have a unique `CLID` identifier. This can be done manually, or can be automated with the GaudiObjDesc (GOD) system in the LHCb project.

GaudiObjDesc example

To use GOD, add a directory `xml` to your package, and place a GOD xml file in it, with a structure similar to:

```
xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gdd SYSTEM "gdd.dtd">
<gdd>
    <package name="ThreeVecPackage">
        <class
          name="ThreeVecEx"
          author="tbd"
          desc="Some description"
        >
            <base name="DataObject"/>
            <attribute name="x" type="double" desc="Value x"/>
            <attribute name="y" type="double" desc="Value y"/>
            <attribute name="z" type="double" desc="Value z"/>
        </class>
    </package>
</gdd>
```

GOD will automatically create accessors for you, such as `GetX` and `SetX`, and will specialize as needed if the base class is `DataObject`. The output header file will be accessible as `"Event/ThreeVecEx"`, and the class will be in the `LHCb` namespace.

The project you are in must be one that includes at least LHCb (such as Lbcom, DaVinci, etc) and you need the following lines in your CMakeLists:

```
include(GaudiObjDesc)
god_build_headers(xml/*.xml)
god_build_dictionary(xml/*.xml)
```

Transient Event Store (TES)

The TES is a place where you can store items on a per-event basis. It should be viewed as non-mutable; meaning that once you place an item in it, it should never change. Persistence is optional and will be covered later. The path to an event should always start with `"/Event"`, though Gaudi is smart enough to assume that a path that does not start with a slash is a relative path, and will get `"/Event/"` prepended to it. In the classic framework, you used `get` and `put` functions to access the TES. For this you must use `GaudiAlgorithm` instead of `Algorithm`, which is a specialization to add access to the TES. You will need to add `GaudiAlgLib` to the linked library list for the GaudiAlgorithm.

To place an item in the event store, create a pointer to a new object, and then put it in the event store in an `execute` method:

```
auto data = new DataObject();
put(data, "/Event/SomeData");
```

The event store will take ownership of the object, so do not delete it.

To retrieve it, also in an `execute` method:

```
auto data = get<DataObject>("/Event/SomeData");
```

Challenge: Data communication

Set up two algorithms, one that produces data and one that consumes it. You can copy the above GOD file to give yourself an object to pass.

You can get a blank template to start this project with:

```
local:~ $ svn export https://github.com/lhcb/DevelopKit/trunk/code/GaudiTemplate MyGaudiProject
```

This will give you a directory called MyGaudiProject with the basics ready. Edit the CMakeLists and directory name inside the project, then you will just need to work on the files in `src` and `options`.

To facilitate building, a `build_and_run.sh` script is included in the project. This will automatically run the commands needed to build the project and run the options file.

Solution

Code solving this is in `/code/GaudiClassicAPI/GaudiDataTrans`

Further reading

There are further examples in the Gaudi repository: Gaudi/GaudiExamples/src.

- Gaudi workshop 2016

The Gaudi framework

Learning Objectives

- "Learn about the Gaudi framework."

Goals of the design

Gaudi is being improved with the following goals in mind:

- Improve scaling by making the interface simpler and more uniform. This allows simpler code, thereby allowing more complex code to be written.
- Improve memory usage, to be cache friendly and avoid pointers to pointers. Use C++11 move where possible.
- Thread-safe code, with no state or local threading. This allows Gaudi to be multithreaded in the future.

In order to reach those goals, the following choices should be made when designing Gaudi algorithms:

- Proper `const` usage: Since C++11, `const` and `mutable` are defined in terms of data races (see this video).
- No global state; otherwise it would be impossible to multithread.
- Tools: should be private or public.
- Ban `beginEvent` / `endEvent`, since there might be multiple events.
- No explicit `new` / `delete` usage (C++11 goal).
- Remove `get/put<MyData>("")` for safer constructs.

Gaudi properties

One of the standout features of new Gaudi is in the new properties; they can be directly defined in the class definition instead of the implementation and used everywhere in the class, instead of requiring special calls in the constructor and custom management of the data. So the following now creates a property:

```
Gaudi::Property<int> m_some_int{this, "SomeInt", 0, "Description of some int"};
```

This defines a member variable of type `Gaudi::Property<int>`, and calls the constructor on initialization with several parameters. The first is `this`, a pointer to the current class. The second is the property name. The third is the default value. You can also optionally give a description.

Notes on the new syntax

The usage of `this` is a common feature of the new interfaces, giving Gaudi components access to the algorithms that contain them. The location of `this` is not consistent across the components, however, as you will see with `AnyDataHandle`.

AnyDataHandle

The requirement to inherit from `DataObject` has been lifted with `AnyDataHandle`. This provides a wrapper than can be put into the TES, and can be retrieved from it as well. Assuming you have some object called `Anything`, you can wrap it in `AnyDataHandleWrapper` and then use the traditional get/put syntax:

```
auto vec = new AnyDataWrapper<Anything>(Anything{1,2,3,4});
put(vec, "/Event/Anything");
```

```
auto base = getIfExists<AnyDataWrapperBase>("/Event/Anything");
const auto i = dynamic_cast<const AnyDataWrapper<Anything>*>(base);
const Anything anything = i->getData();
```

This, however, is significantly convoluted. A much cleaner way to add an `AnyDataHandle` is to construct it as a member of your class, just like a `Gaudi::Property`, and then use the `.get` and `.put` methods.

```
AnyDataHandle<Anything> m_anything{"/Event/Anything", Gaudi::DataHandle::Writer, this};
m_anything.put(Anything{1,2,3,4});
```

```
AnyDataHandle<Anything> m_anything{"/Event/Anything", Gaudi::DataHandle::Reader, this};
const Anything* p_anything = m_anything.get();
```

Notes on the new syntax

Adding AnyDataHandle as a member variable breaks compatibility with the `using` statement, so you will need to explicitly define the constructor.

This works by replacing inheritance with *type erasure*.

Gaudi::Functional

Rather than provide a series of specialised tools, the new `Gaudi::` Functional` provides a general building block that is well defined and multithreading friendly. This standardizes the common pattern of getting data our of the TES, working on it, and putting it back in (in a different location). This reduces the amount of code, makes it more uniform, and encourages 'best practice' procedures in coding in the event store. The "annoying details", or scaffolding, of the design is left to the framework.

A functional, in general, looks like this example of a transform algorithm to sum two bits of data:

```
class MySum
 : public TransformAlgorithm<OutputData(const Input1&, const Input2&)> {

     MySum(const std::string& name, ISvcLocator* pSvc)
      : TransformAlgorithm(
           name,
           pSvc, {
              KeyValue("Input1Loc", "Data1"),
              KeyValue("Input2Loc", "Data2") },
           KeyValue("OutputLoc", "Output/Data")) { }

    OutputData operator()(const Input1& in1, const Input2& in2) const override {
        return in1 + in2;
    }
```

This example highlights several features of the new design. This starts by inheriting from a templated class, where the template defines the input and output expected. Different functionals may have no input and/or output, and might have multiple inputs or outputs. The

constructor takes KeyValue objects that define the data inputs and outputs, with multiple data elements input as a initializer list. The `operator()` method is overridden, and is `const` to ensure thread safety and proper design. This takes the inputs and returns an output.

The functionals available in the framework are named by the data they work on (with examples):

- One input, no output: `Consumer`
  - `Rec/RecAlgs` : `EventTimeMonitor` , `ProcStatusAbortMoni` , `TimingTuple`
- No input, one or more output: `Producer`
  - Should be used for File IO, constant data
- True/False only as output: `FilterPredicate`
  - `Phys/LoKiHlt` : `HDRFilter` , `L0Filter` , `ODINFilter`
  - `Phys/LoKiGen` : `MCFilter`
  - `Hlt/HltDAQ` : `HltRoutingBitsFilter`
  - `Rec/LumiAlgs` : `FilterFillingScheme` , `FilterOnLumiSummary`
- One or more input, one output: `Transformer`
- One or more input, more than one output: `MultiTransformer`
- Identical inputs, one output: `MergingTransformer`
  - `Calo/CaloPIDs` : `InCaloAcceptanceAlg`
  - `Tr/TrackUtils` : `TrackListMerger`
- One input, identical outputs: `SplittingTransformer`
  - `Hlt/HltDAQ` : `HltRawBankDecoderBase`
- Converting a scalar transformation to a vector one: `ScalarTransformer`
  - `Calo/CaloReco` : `CaloElectronAlg` , `CaloSinglePhotonAlg`
  - This is used for vector to vector transformations where the same algorithm is applied on each vector element with 1 to 1 mapping.

A note on stateless algorithms

A stateless algorithm (one that does not store state between events) provides several important benefits:

- Thread safety
- Better scalability
- Leaner code

The downside is that a lot of code needs to be migrated.

Producer

The simplest of the functional algorithms, this produces `Out1` to `OutN` given nothing:

```
class MyProducer : public Gaudi::Functional::Consumer<int()> {
public:
    MyProducer(const std::string& name, ISvcLocator* svcLoc)
              : Producer( name, svcLoc,
                KeyValue("OutputLocation", {"MyNumber"})) {}

    int operator()() const override {
        return 314;
    }
}
```

This inherits from the templated Producer, where the template is the signature of the `operator()` function. Here, it is a function that produces an int given nothing. When making the constructor, the base class is called with one more argument, the output(s) of the function

as `KeyValue`, where the first argment is the key (output location), and the second one is a list of locations in the TES that will be produced (here, it is `/Event/MyNumber`).

The important part of the class is the `operator()`, which produces an int (just 314 in this example) that gets placed on the TES. The Producer is currenlty unused in the LHCb codebase, but is intended for use for IO from files, and for random number generation.

If you want to produce several outputs, you can return a tuple, and give a list of KeyValues, one for each tuple member.

Consumer

This is a class that takes in TES data. It looks like:

```
class MyConsumer : public Gaudi::Functional::Consumer<void(const int&)> {
public:
    MyConsumer(const std::string& name, ISvcLocator* svcLoc)
              : Consumer( name, svcLoc,
                KeyValue("InputLocation", {"MyNumber"})) {}

    void operator()(const int& input) const override {
        info << "Printing input: " << input << endmsg;
    }
}
```

The class is created inheriting the templated class `Consumer`, with the signature of the `operator()` function as it's first argument as before.

The constructor should include the `KeyValue` input that it will use. The Consumer in the example above takes the int we put in the TES before and prints it to the info log. The value in the TES is obtained by reference.

FilterPredicate

This blocks algorithms behind it, returns `filterPassed`.

Transformers

Split or merge containers. These take one or more inputs, and produce one or more outputs.

Conversion from the old framework to the new

- Try to convert everything to a `Gaudi::Functional` algorithm. Split into smaller basic algorithms and build a `SuperAlgorithm` from the smaller ones if needed.
- At least try to migrate to data handles + tool handles.
- Make sure the event loop code is `const`; do not cache event dependent data. Interface code, especially for tools, should be `const`.
- Try to migrate away from `beginEvent`, `endEvent` incidents.
- Please add as many tests as possible!

An example

An example with some of these features can be found by running:

```
local:~ $ svn export https://github.com/lhcb/DevelopKit/trunk/code/GaudiNewAPI GaudiNewAPI
local:~ $ cd GaudiNewAPI/GaudiFunctional
local:GaudiFunctional $ source build_and_run.sh
```

This package has two examples, `GaudiFunctional` and `GaudiDataTrans`, which use the new system to transfer data in and out of the TES

using DataHandles and the functional approach. Feel free to explore and modify these examples. If you come up with a new useful example, please submit it as a PR.

Upgrading an Algorithm

Learning Objectives

- "Learn about upgrading algorithms to the new Gaudi framework."

Steps to convert

The following are the five steps to upgrade an algorithm.

Identify TES interactions

First, find and identify the input data and output data. You are looking for `get` , `put` , or `getIfExists` . For example:

Example: Locating TES interactions

```
local:dir $ egrep 'get|put' Pr/PrAlgorithms/src/PrMatchNN.cpp
declareProperty("VeloInput", m_veloLocation=LHC...
declareProperty("SeedInput", m_seedLocation=LHC...
declareProperty("MatchOutput", m_matchLocation=...
put(matchs, m_matchLocation);
LHCb::Tracks* velos = getIfExists(m_veloLocation);
LHCb::Tracks* seeds = getIfExists(m_seedLocation);
```

In this case, the `Velo Tracks` + `Seed Tracks` are producing `Output Tracks` .

Deduce functional algorithm to use

Out/in	0	1-n	vector	
0		`Consumer`		
1	`Producer`	`Transformer`	`MergingTransformer`	
n	`Producer`	`MultiTransformer`		
vector		`SplittingTransform`		
boolean		`FilterPredicate`		
boolean+1-n		`MultiTransformerFilter`		

Possible inputs

- **0** : no input, pure producer

- **1-n** : one or several independent inputs. This number and the type of each input must be known at compile time
- **vector** : any number of inputs (not know at compile time), all of the same type, aka a vector of inputs of the same type

Possible outputs

- **0** : no output, pure consumer
- **1-n** : one or several independent outputs. This number and the type of each output must be known at compile time
- **vector** : any number of outputs (not know at compile time), all of the same type, aka a vector of outputs of the same type
- **boolean** : an additional boolean output, saying whether we should carry on or stop the processing, aka a filter

Modify Algorithm declaration

- Change inheritance from Algorithm to your functional algorithm base class.
- Template the functional algorithm with the "signature" of the algorithm, that is its data flow.

Example: Declaration

In the case of `Velo Tracks` + `Seed Tracks` producing `Output Tracks` :

```
Gaudi::Functional::Transformer
      <LHCb::Tracks(const LHCb::Tracks&, const LHCb::Tracks&)>
```

In the case of `Tracks` producing `Vector of Tracks` :

```
Gaudi::Functional::SplittingTransformer
      <std::vector<LHCb::Tracks> (const LHCb::Tracks&)>
```

- All inputs are now using `const` references.
- All declarations of members storing locations of input/output can be dropped (automatic retrieval from TES).
- Equivalent properties are automatically defined by the functional framework.

Example: Member declaration

Member declaration:

```
std::string m_veloLocation;
```

Used in constructor as:

```
declareProperty("VeloInput", m_veloLocation=...
```

Example: PrMatchNN

Old:

```
#include "GaudiAlg/GaudiAlgorithm.h"
class PrMatchNN : public GaudiAlgorithm {
   ...
   std::string m_veloLocation;
   std::string m_seedLocation;
   std::string m_matchLocation;
```

New:

```
#include "GaudiAlg/Transformer"
class PrMatchNN :
    public Gaudi::Functional::Transformer
    <LHCb::Tracks(const LHCb::Tracks&, const LHCb::Tracks&)> {
    ...
```

Modify constructor/destructor

Changes

- Initialize functional algorithm rather than `Algorithm`
- New arguments are needed
  - Input location / list of input locations in TES
  - Output / list of output locations in TES
- Locations are given using `KeyValue` objects, such as `KeyValue{"VeloInput", LHCb::TrackLocation::Velo}`
- This creates the adequate property in the back
  - So old `declareProperty` lines can be dropped
- For each location, a default might be given

Example: PrMatchNN

Old:

```
PrMatchNN::PrMatchNN(const std::string& name, ISvcLocator* pSvcLocator) :
  GaudiAlgorithm(name, pSvcLocator), ... {
    declareProperty("VeloInput", m_veloLocation=...
    declareProperty("SeedInput", m_seedLocation=...
    declareProperty("MatchOutput", m_matchLocation=...
```

New:

```
PrMatchNN::PrMatchNN(const std::string & name, ISvcLocator* pSvcLocator) :
 Transformer(name, pSvcLocator,
   {KeyValue{"VeloInput", LHCb::TrackLocation::Velo},
    KeyValue{"SeedInput", LHCb::TrackLocation::Seed},
    KeyValue{"MatchOutput", LHCb::TrackLocation::Match}}),
 ... {
```

Convert execute into operator(...)

The declaration of `operator()` will have a signature matching the data flow, and will be made using `const` references. The definition is of the form:

```
Output operator() (const Input&, ...) const override;
```

The input is a `const` reference, the method itself is both `const` and will `override` another method. The output is returned by value. It is a good idea to add `std::move(...)` around the return value to make sure a copy is not made; compilers, especially newer ones, may do this for you.

Compared to execute

- Drop all interactions with TES (`get` / `put`)
- Replace memory allocation of the output by simple creation on the stack
- Thus change accordingly `->` into `.` , drop some `*`
- Return output rather than a `StatusCode`
- Throw an exception when you were returning a bad `StatusCode`

Example: PrMatchNN

To simplify the following example, the status code checks have been removed.

Execute method:

```
LHCb::Tracks* matchs = new LHCb::Tracks;
put(matchs, m_matchLocation);
LHCb::Tracks* velos = getIfExists<LHCb::Tracks>(m_veloLocation);
LHCb::Tracks* seeds = getIfExists<LHCb::Tracks>(m_seedLocation);
StatusCode sc = m_matchTool->match(*velos, *seeds, *matchs);
return sc;
```

Operator ():

```
LHCb::Tracks matchs;
StatusCode sc = m_matchTool->match(velos, seeds, matchs);
return matchs;
```

To convert, first the TES interactions were dropped (now part of the header), memory allocation was removed (created on the stack), dereferencing pointers was removed, and the object was returned instead of the `StatusCode` .

Improving

These changes were made to give faster, more readable code, easy transition to parallel Gaudi, and more checks at compile time. To truly get improvements to the code, a few further steps should be taken to update the algorithms while it is already being modified and reviewed.

Property

`Gaudi::Property` provides a simple, clean way of declaring properties, and almost obsoletes the old `declareProperty`. It is not completely backward compatible, but fixes are trivial and usually result in better code. The property setup will exist in one place (header) instead of two (header and constructor).

Example: Property conversion

Old header

```
bool m_skipFailedFitAtInput;
double m_maxChi2DoF;
```

Old constructor

```
declareProperty("SkipFailedFitAtInput", m_skipFailedFitAtInput=true);
declareProperty("MaxChi2DoF", m_maxChi2DoF=9999, "Max chi2 per track");
```

New header

```
Gaudi::Property<bool> m_skipFailedFitAtInput
    {this, "SkipFailedFitAtInput", true };
Gaudi::Property<double> m_maxChi2DoF
    {this, "MaxChi2DoF", 9999, "Max chi2 per track"};
```

In most cases, this change is transparent and you can use the property the same way. In cases where it does not work, such as when using `->` for a pointer property, you can use `.value()->` instead. Also, printing a property to an iostream is customized, giving name and value. If the old print is needed, use `.value()` to output just the value.

Modern C++

This is a great time to update to modern C++ syntax. Although all C++11 and much of C++14 is readily available, even in the base-line GCC 4.9 compiler, the following changes are especially useful:

- Member initialization: By moving member initialization to the declaration, you can simplify the constructor and discover missing initializations.
- Use `nullptr` instead of `0` or `NULL`: Type safety is always a good thing, and can catch pointer bugs.
- Use `using` constructor if possible: if everything can be moved out of the constructor, you can use a `using` directive instead to further clean up code.
- Use `auto` and range-based loops: Often it is nicer to your reader to be explicit, but in many cases, the type does not matter or is something complex, like an iterator, that really does not need to be known. Those can be moved to using `auto` to clean up code. Also, closely related, moving to range-based for loops can make code much, much easier to read.

- Lambdas: Using the built in lambdas instead of `boost::lambda` and its quirks. Also can be used to move some tasks to the STL that used to be loops.

Thread safety

Moving to the new framework *should* take care of thread safety by using `const`, but there are a few ways around it that, unfortunately, are common in our code base. You can remove `const` with `const_cast`, and a member variable can be changed in a `const` method if it is declared `mutable`. As a general rule, you should only see `mutable` in C++11 when the variable itself is thread-safe (for example, a mutex). It should *not* be used to simply make adapting bad code easier. To remove them, you can usually simply remove the `const_cast` or `mutable`, then trace down the compiler errors to see what is happening.

Often these are introduced because code is not using the TES for data transmission; the new `AnyDataHandle` makes that trivial to do now, so the best solution is to put the data in the TES using an `AnyDataHandle`.

You'll want to remove `Incident`s, as well. You'll see something like this:

```
void handle(const Incident&) override final;
```

And, in the constructor:

```
incSvc()->addListener(this, IncidentType::EndEvent);
```

In most cases, this is only needed to reset a cache at the start of the event. The cache needs to be removed to be thread safe anyway, so that also solves the `Incident` issue. In the other cases, the TES or function arguments should replace the communication that the `Incident` was providing.

Further reading

> - 5th hackthon talk (this lesson was based on the presentation here by S. Ponce.)

New Counters

Learning Objectives

- "Learn about upgrading to the new counters in Gaudi

Old counters

The old counters looked like this:

```
++counter("# valid banks"); // Add one
counter("skipped Banks") += tBanks.size(); // Add a number
++counter("nInAcceptance"); // binomial here !
```

This approach as a plethora of issues. First, counters are looked up by string. If one does not exist with that string, a new one is created. There are special name patterns that indicate different sorts of counters, such as "acc" for binomial. This is all protected by a mutex, which locks on every counter access. And there are five doubles that are tracked for all counters, such as the mean, even if all you need is a simple increment counter.

All together, this meant that 33% of the time in `createUTLiteCluster` was spent on counters alone!

Half way there

This removes the name lookup, but still is not ideal, either for timing or design:

```
class ... {
    mutable StatEntity m_validBanks{this, "# valid banks"};
};
```

New counters

To replace this system, a new counter object was built in Gaudi. Usage looks like this:

```
class ... {
    mutable Counter<> m_validBanks{this, "# valid banks"};
};
```

This is allowed to be mutable, since it locks internally (just like an atomic or mutex member of a class can be mutable). The string here is only used for print out purposes. And there is a family of counters, depending on what you want:

Name	Usage details
`Counter`	A pure counter, counts number of entries (`++` only).
`AveragingCounter`	Keeps count and sum, can compute mean. (`+=` also)
`SigmaCounter`	Also keeps sum of squares, can compute variance, standard deviation.
`StatCounter`	Also keeps min/max values.
`BinomialCounter`	Dedicated to binomial law, compute efficiency.

There are also two template parameters, with the following defaults:

```
Counter<double, atomicity::atomic>
```

The first template parameter is the type (float is usually fine), and the second is the atomicity, with `atomic` and `none` (non-atomic operations, for local counters). If you use `atomicity::none`, the member should not be marked `mutable`!

Temporary counters

This helped improve the speed, partially by using atomics instead of mutex locks, and partially by reducing the number of operations when not needed. But the biggest improvement came from buffered local counters. If you create a buffer counter from a counter, you can update it locally, and then when the object goes out of scope, the counter it was created from updates. You can now do this:

```
AveragingCounter<float> counter{this, "main counter"};
{ // Entering a local scope
    auto buf = counter.buffer();
    for (int i = 0; i < 10000; i++)
        buf += i; // No atomics here, pure local!
} // single atomic update of main counter
```

This allows you to loop over tracks, or some other per-event structure, and still only update your counters once per event (which is much, much faster). This removed the rest of the measurable time spent in counters in `createUTLiteCluster`, giving a 33% speed up!

Design for experts

- Counters are only accumulators + printing
- Many basic accumulator exists
  - count, sum, square, min, max, true, false
- You can easily extend the set, based on `GenericAccumulator` class
  - Templated by type, atomicity, transform (identify, square), `ValueHandler` (sum, maximum, ...)
- They are then merged using `AccumulatorSet`

```
struct AveragingAccumulator :
    AccumulatorSet<Arithmetic, Atomicity, CountAccumulator, SumAccumulator>
```

What's new in C++ for LHCb Physicists

The following lessons cover the latest changes in C++, so that you can help modernize and clean up any code that you come across. The upgrade environment will have at least access to C++14, and probably C++17.

New features in C++11 for LHCb Physicists

Learning Objectives

- "Learn a few of the most useful tools in C++11."

C++11 was the largest change ever made to C++; and due to the changed release schedule, probably will remain the largest single change. It is a well thought out, mostly backward-compatible change that can completely change the way you write code in C++. It is best thought of as almost a new language, a sort of (C++)++ language. There are too many changes to list here, and there are excellent resources available, so this is meant to just give you a taste of some of the most useful changes.

Many of the features work best together, or are related. There already are great resources for learning about C++11 (listed at the bottom of this lesson), and C++11 is already in use in current LHCb software. Therefore, the remainder of this lesson will cover a few of the common idioms in C++11 that a programmer experienced with the older C++ might not immediately think of.

Types

Typing in C++11 is much simpler. If the type is deductible, auto allows you to avoid writing it out. A few examples of uses of auto:

```
// Avoiding double writing on pointers
auto pointer = new SomeNamespace::MyLongType(arg1, arg2);

// Does anyone know the type of an iterator? It's ugly!
auto iterator = some_vector.begin();

// This is useful for prototyping, but probably should be explictly typed in real code to help coders
auto type = function_returns_something();

// Worst use of auto; don't do this
auto a = 1;
```

ROOT macros have a syntax that precedes auto; you can simply assign (`x = Something();`) and if `x` does not already exist, it does the equivalent of `auto x = Something();` . This has the drawback that it is not valid C++ code, but unfortunately is still useful in a RootBook, where you may want to rerun a cell.

The `NULL` keyword, which is equivalent to 0, causes typing issues. A new `nullptr` keyword was added, and is not equivalent to zero. This is significantly better type-safety, and should be used instead of `NULL` .

The definitions of `const` and `mutable` changed a little; see this video. In short: `const` actually means bit-wise constant or thread-safe; mutable means the object is already thread-safe (atomics, mutexs, some queues). This is probably what you thought they meant (but didn't) in C++98.

Containers and iterators

While iterators existed in previous versions, using them is now part of the language, with the iterating for statement (for each). It allows a unintuitive iteration loop to be written more cleanly and compactly. Compare the following two methods of setting the values of a vector to zero:

```
for(vector<double>::iterator iter = vector.begin(); iter != vector.end(); ++iter)
    *value = 0.0;

for(auto &value : vector)
    value = 0.0;
```

To make something iterable, you should define a `begin` and `end` method or function. There are several options, as well. Adding an `&` will give you a reference, to save memory copies and to allow mutation of the original iterable. Adding `const &` avoids the copy but ensures that you don't change the original iterable.

A related improvement is the addition of container constructors. Which means you can *finally* do this:

```
std::vector<int> values = {1,2,3,4,5,6};
```

C++ also has a variety of different initializers; C++11 added a uniform initializer syntax, so it has even more different initializers. The benefit is that it works in cases that the old syntax had issues. It solves the *Most Vexing Parse*, where a new object cannot be followed by parenthesis, since that is a function definition. It also does not explicitly narrow.

Due to the addition of initialiser lists, though, this can be confusing. What does the following statement produce, a list of 42 integers, or one integer with the value 42?

```
std::vector<int> vector{42};
```

Answer

The standard oddly prioritizes the List Initializer, if the object in question supports List Initializer, and so this is one vector of the value 42. The only way to initialize 42 empty values is to use the old syntax, `std::vector<int> vector(42);`, making this Uniform Initializer Syntax definitely not Universal Initializer Syntax.

This might show up most often in member initializers, which were added to C++11 also; these have the same caveat.

- Example

Functional programming

Functions are now easier to refer to and create. A `std::function` type is useful, but usually will be hidden with `auto` unless you are crafting a function to take functions as arguments. The lambda function allows an inline function definition, with some perks. The syntax is `[](){}`, which looks like a normal function definition with the function name and type replaced by the square brackets. For example:

```
auto square = [](double x){return x*x;};
double squared_five = square(5.0);
```

The lambda function gets interesting when you add something to the square brackets; this is called "capture" and allows you to capture the surrounding variables. For example:

```
int i = 0;
auto counter = [&i](){return i++;};
counter(); // returns 0
counter(); // returns 1
```

You can capture by value, by reference, etc. If you use `[=]` or `[&]`, the lambda function will automatically capture (by value or reference, respectively) any variables mentioned inside the function. Note that if you do use capture, you will lose the ability to convert to a

old-style C function pointer.

- Example

Class improvements

Default values for members can be declared in the definition now (as I'm sure you've tried to do in older C++ at least once). You can also call a previous constructor in the initializer list (delegating constructors).

Compile time improvements

The slow removal of the ugly, error-prone macro programming has started in C++11, with `constexpr`. A function or class with this modifier (and lots of restrictions as to what it can contain) can be used by the compiler at compile time to produce a result in your compiled code.

The move operation was added, as well. This allows moving a value through a special `std::move`, and also can be invoked by the compiler automatically if you return a value from a function. This reduces the need to return pointers from functions.

Std library improvements

The powerful `std::shared_ptr` and `std::unique_ptr` remove most of the reasons to fear pointers, though they don't work that well with toolkits like ROOT that try to do their own memory management. A chrono library was added for consistent timekeeping on all platforms. A threading library provides tools that work with the new functional tools and makes threading easy, and also a mutex and atomic library to support it. A regular expression library was added. More algorithms were added. A sized integer library was *finally* added with `stdint.h` (a benefit of being based on C99). Random number generation is finally properly supported, with a good set of algorithms and distributions.

Several container libraries were added. The most notable container library addition is the array library, which provides a compile time replacement for C-arrays, with bounds features. Unordered versions several containers were added. Containers became much more powerful, now with move semantics, as well as `emplace_back`, which can construct a value directly inside a container. Most containers support initializer lists.

A functional library was added to give an explicit type to function pointers, and is used for lambdas as well. A bind function allows you to add function arguments to a function (currying in functional programming terms).

Variadic templates

Variadic templates allow a function or constructor to take an unlimited number of arguments of any type. This allows the `std::tuple` feature, Gaudi's functional algorithms, and other very powerful features. Although the implementation may seem unusual at first, especially if you are used to a scripting language like Python, the implementation allows the parameter expansion to happen at compile time, meaning there is no runtime penalty for using variadic templates.

The syntax uses the following constructs:

- **Declaring a parameter pack.** This is denoted by an ellipses to the *left* of a parameter name. The ellipses are usually placed next to the proceeding type, such as in `template<typename... Ts>` or `Ts... values`, but the meaning is the same. For a class this must be the last parameter; for a function it can occur earlier (but usually does not).
- **Expanding a parameter pack.** This is denoted by an ellipses to the *right* of a parameter pack or expression containing a parameter pack. This is often seen in the body of the functions. These are commonly used to call functions, `funct(values...)`, or perform an expression then call a function `funct(do_something(values)...)`. In the second example, `do_something` is called on each value before calling the `funct` function.
- **Counting the parameters in the parameter pack.** This is done with the `sizeof...()` function, which is a `constexpr` function that returns the number of parameters inside the parameter pack.

Variadic template function example

An example of a Python style print function:

```
void print() {
    std::cout << std::endl;
}
template<typename T, typename... Ts>
void print(T value, Ts... values) {
    std::cout << value << " ";
    print(values...);
}
```

Here, the ellipses serves the same role the first two times it is seen; it is declaring a parameter pack `values` with type `Ts`. Inside the function, the ellipses is expanding the call to `print(first value, second value, etc)`. This recursively calls itself, ending in the final empty argument form; a very common pattern for variadic templates.

Note

There is an old feature called variadic functions from C, which allows an unlimited number of arguments, but is not type safe. This is how the unlimited argument `printf` function is defined. This uses an ellipsis at the end of the parameter list, optionally after a comma.

For more on variadic templates and what they can do, see Cpp Reference's page on paramter packs.

Move semantics

One of the more fundamental changes in the language was the promotion of move semantics to a language feature, as well as stronger guidelines on auto-optimization. This can fundamentally change the way functions are written. What is the problem with this statement in C++03?

```
GiantObject item = GiantObject_returning_function();
```

Here, you create a `GiantObject` inside the function, and then copy it to a new object `item`, then delete the old object. It's horribly wasteful in both time and memory; if you don't have enough memory for two separate copies of `GiantObject`, you can crash your program. In C++11, not only is the idea of a move instead of a copy added, the compiler is generally recommended to do that for you if it can. So the value will simply be moved, with no changes to either the function or the line above, as long as there are no references retained to the object inside the function (though globals, members, parameters, etc). This is a compiler recommendation, rather than a formal requirement of the language, but most compilers implement it. (In C++17, it becomes an official requirement of the language in most cases.) This is part of C++11's move away from the use of pointers, and is huge step in the right direction.

Moving is also a part of the language, but using it explicitly requires some understanding of a fundamental feature of C++ that was not usually talked about before: value categories.

Explanation of value categories

The names rvalues and lvalues historically refer to where the expression tends to be relative to an assignment operation. So for the expression:

```
x = 1+2;
```

The `x` is an lvalue, and the 1+2 is an rvalue. The names can be slighltly misleading, since an l-value could occur on either side of the assignment, and there is no requirement that an assignment be made in every line of C, but every expression (or sub-expression) can be classified under these two categories in C++03. The difference is in memory; the L-value is given a real location in memory, while the rvalue is temporary and cannot be used past the current expression (and the compiler might optimize it away in some cases). An example of an expression that is an rvalue is `*(x+1)`, where x is a pointer. It is quite possible for a function to return an lvalue.

In C++11, rvalues are further complicated because of the desire to move object. To properly grasp the syntax, it is important to understand all 3 unique categories in C++11, from the C++ standard n3055:

- **lvalue**: Anything that can be on the left side of equal sign.
- **xvalue**: An rvalue that about to expire (something being returned from a function, for example). An xvalue can be moved.
- **prvalue**: An rvalue that is not about to expire, like a literal (`12` , `true`) or the result of a non-reference return of a function.

C++11 also defines two combination categories, **glvalue** is an lvalue or xvalue, and the classic **rvalue**, which is a prvalue or an xvalue.

Syntax for the value categories

Functions can return any of the three C++11 categories:

```
int returns_prvalue();
int& returns_lvalue();
int&& returns_xvalue();
```

And, functions can take either of the values:

```
void foo(int any_value);
void foo(int& lvalue);
void foo(int&& rvalue); // rvalue takes an rvalue, but inside the function it is a named lvalue
```

This hopefully gives you an idea of what a move function must look like. Its signature must be:

```
T&& move(T&& input_rvalue);
```

Here, it takes an rvalue and returns an xvalue. It is the same thing as doing a static cast to an rvalue. This xvalue is therefore movable by the compiler.

Example of explicit move

```
std::string a = "original";
std::string b = "new";
b = std::move(a); // a now empty
std::cout << "[" << a << " " << b << "]";
// Will output: [ orignal]
```

The most common uses tend to be in overloads, such as for move constructors, to specialize behavior for moves.

Other features

Tuples allow multiple return values, albeit through `std::tie`. In C++17 they are elevated to a more fundamental part of the language through new syntax.

C++ now supports user defined literals, for strings and for numbers. These were not included in the standard library for common types until C++14, though. One use was added, though; raw string literals `R"..."` were added, with similar meaning and semantics to the Python raw strings, were intended for the same use (regular expressions).

Attributes were added, allowing arbitrary tags to be added to expressions. There are very few official ones, though some get added for each version. These are intended mostly for compiler specific functions, like OpenMP. They are indicated by double square brackets.

The parser is smarter, finally understanding `<one, <two>>` correctly without forcing a space between the greater-thans.

Suffix return type allows the return type to be listed after the function definition line, instead of before; this allows templated functions with complex return types to be written more easily, since the parameters exist after the definition line, but not before.

The override keyword indicates a function is intended to override a virtual function, making the intention clear (and compiler can check and issue errors). The final keyword is also added, enforcing a virtual function to be non-overridable.

Further reading

There are many, many smaller features, as well as details of the larger features that were not covered here. See:

- Bjarne Stroustrup's document - written by the creator of C++.
- SmartBear blog post
- ISO CPP
- Cpp Reference

New features in C++14 for LHCb Physicists

Learning Objectives

- "Learn a few of the most useful tools in C++14."

Unlike C++11, this is a minor release, focused mostly on improvements on top of C++11 changes, with very little that one could call "new". C++14 feels a little more natural than C++11 by expanding the usage of features and implementing common sense additions.

The biggest issue with C++14 is the GNU compiler; full C++14 support was released in version 5.0, but due to some binary changes, many projects, including SL6 and Ubuntu, chose to delay moving from version 4.9. This is finally a smaller issue with some fixes in 6.0, but it will take a while for that version to become commonly used. By run 3, hopefully C++14 and even C++17 will be completely available.

Also, while C++11 is available in ROOT 6, C++14 requires a flag and compatible compiler, so C++14 features are often unavailable.

Auto and lambdas

Type syntax is more consistent in C++14, with `auto` working in more places. You can now write a function returning `auto` and it will deduce the return type from the return statements instead of having to use the peculiar trailing return type syntax and `decltype`. This should be used with caution when designing a function for a third party use; you should always present a clear interface, and auto obscures that. This used to work only with lambdas and only if there was a single return statement. A related improvement with lambdas is that they now support auto for parameters; this allows a template lambda functions to be created and reused (called a *generic lambda*). They have also gained slightly more powerful capture abilities.

Constexpr

This expression has gained quite a bit in C++14; you can now use `if`, `switch`, and loops inside a constexpr, as well as variable definitions and mutating local objects. A bit more of the standard library now includes constexpr, as well, such as `std::array`. Most of the algorithms still do not have constexpr and thus require using a third party library, sadly.

A related change is the addition of variable templates. You can now define a variable with template specializations; for example, you could define pi to provide a double and a string representation all in one variable, depending on how it is used. This could also be used to define a constant variable without being tied to a library.

Smaller changes

The standard library received a few improvements, as well. The C++ literals syntax is finally supported by the standard library. You can now write:

```
using namespace std::string_literals;
some_string_function("This is a std::string"s + " simply by adding an s at the end"s);
```

Other standard library types have literals support for units now too, too, such as the chrono library. As an example where a duration is created:

```
using namespace std::literals::chrono_literals;
auto duration = 1h + 2min + 3s + 4ms + 5us + 6ns;
```

Complex numbers also gained a set of literals, as well. The literal syntax is entirely a C++11 construct, the new feature is just the addition of the predefined literals to the standard library (albeit in an opt-in namespace). These literals, however, are free from the requirement that a user-defined literal must follow; they do not start with an underscore.

Digit separators, using single quotes, can make large numbers more readable. They are ignored by the compiler and are only visual aids. Binary literals are now available, using a `0b` prefix.

- Example

An omission of the C++11 standard was fixed with the addition of `std::make_unique` to mimic `std::make_shared` for unique smart pointers.

The type system has received minor improvements, with `_t` type aliases to reduce typing (pun intended), and a `std::enable_if_t` helper type makes `std::enable_if` slightly less verbose (if you are stuck in C++11, this is easy to define using the possible implementation).

Bugfixes

Several bugs in `C++11` were addressed, as well. Most compilers backport these fixes into C++11 mode, such as GCC4.8/4.9. One example is type overloading with `std::function`; in `C++11`, you could not distinguish between different `std::function` signatures for function or method overloads, but in C++14 and newer C++11 compilers you can.

Further reading

A few other resources:

- Official C++14 Overview, C++14 language additions, and C++14 library additions pages
- Dr. Dobbs
- InfoQ
- A set of general rules for good C++14

New features in C++17 for LHCb Physicists

Learning Objectives

- "Learn a few of the most useful tools in C++17."

The every-three-year cycle has changed the development of C++; we are now getting consistent releases somewhere in-between the major and minor releases of old. The 2017 release may be called minor by some, with a huge portion of the planned improvements being pushed back another 3-6 years, but there were several substantial changes in areas that particle physicists can use. The LHCb software (except for the underlying Guadi) uses C++17 as it exists in GCC 7.

Syntax improvements

GCC 7 should be feature complete for syntax improvements (as is Clang 5).

Constructor argument deduction

Templated constructors (finally!) support template deduction, allowing them to behave like functions. This makes the old function wrappers, like `make_tuple` and `make_shared` obsolete. This makes many templated classes much easier to use for the average user.

```
// C++11
std:::tuple<int, double> my_tuple(my_int, my_double);
// C++17
std::tuple my_tuple(my_int, my_double);
```

Tuple syntax

Tuple syntax is now part of the language, rather than just part of the library. Combined with the previous improvement to constructors, this allows the following to be written in Python-like syntax:

```
// C++11 syntax
std::tuple<double,int> return_two_args() {
    return make_tuple(1.0, 2);
}
double a, int b;
std::tie(a,b) = return_two_args();
```

```
// C++17 syntax
auto return_two_args() {
    return std::tuple(1.0, 2);
}
auto [a, b] = return_two_args();
```

Along with the C++14 auto return type deduction, C++17 allows the tuple to be created directly without a wrapper function or template parameters, and it allows the variables to be created and assigned directly in the assignment statement (structured bindings). This also works for `std::arrays` and some structs, and can be overloaded for custom classes. You can also do this (or any one line initializer) inside an `if` statement followed by a semicolon (like `for`), and it will be valid until the end of the `else` clause. A couple of functions have been added to make using tuples as arguments easier; `std::apply` for functions and `std::make_from_tuple` for classes.

Reducing usage of the preprocessor

The slow removal of the secondary preprocessor language continues with a static `if constexpr` statement. Lambdas now can be `constexpr`, and are allowed in `constexpr` functions. And while we are on the subject of lambda's, they also gained the ability to capture the pointer to the current class by value, with `[*this]`.

The language is now better defined, with less left up to the compiler implementation. The order of expression evaluation is no longer left up to the compiler in most cases, reducing subtle portability bugs. The compiler is now guarantied to avoid a copy (copy elision) in many cases; you can even return a class that does not support moving. (Note: there's a bug with copy elision in GCC 8.1, to be fixed in 8.2). Inline variables are finally allowed, removing the compiler errors that required irritating workarounds and separate `.cpp` files when all you needed was a header file. You still are not supposed to have global variables, but if you do, now they are easier to write correctly.

Variadic macros now have folding syntax, which allows you to perform reduction with them easily. Inside your template function with variadic parameter `args` you can do `0 + ... + args` to sum over `args` (other operators are supported, including `,`. The `...` syntax can be used for using declarations now, too.

`#if __has_include(<headerfile>)` now allows you use the preprocessor to check for availability of header files, greatly simplifying the build system, since you can now directly check to see if a library is available.

New attributes have been added, such as `[[maybe_unused]]`. Attributes are allowed in more places, and there is now a clear requirement that a compiler ignore any attribute that it does not recognise. These hopefully will finally begin to replace the various `#pragmas` currently in use.

Smaller features

You can nest namespaces directly now.

```
namespace A::B {...
```

Template arguments can be `auto`, which is any non-Type argument. Hexadecimal float point literals are allowed. List initialization works for enums. C11 is now the base, instead of C99.

Standard Template Library

These improvements are a bit slower in coming, as usual. Features that are not available yet will be noted.

Utilities

The removal of using raw pointers and unsafe C syntax continues with three tremendously useful classes from Boost. Feel free to check the cppreference page or Boost libraries for examples and usage.

Optional

One of the common uses for pointers is the creation of a value that might not exist (`nullptr`). This has the undesirable side effect of dynamically allocating memory, and requires the optional deletion of the allocated memory. `std::optional` provides a safe, stack based solution that clearly defines your intent, as well.

```
std::optional<Massive> maybe = maybe_make_massive_obj();
if(maybe) {
    std::cout << "Massive object exists: " << *maybe;
}
```

Variant

`std::variant` provides a C++ syntax for a safe C `union` replacement. It can store a value from a predefined list of types. The size of the object is equal to the largest possible contained type.

```
std::variant<int, std::string> either;
either = 2; // Now an int
either = "hi"; // Now a string
try {
    int x = std::get<int>(either);
} catch (std::bad_variant_access&) {
    std::cout << "This is not an int";
}
```

One particularly powerful feature is the `.visit` method, which takes a callable that must be valid for all possible types. You can combine this with auto lambdas to process the contents very generally.

Any

`std::any` should help kill the desire to use unsafe void pointer casts. It can hold anything, and allows access with `any_cast` specialisations.

String view

A new class, `std::string_view`, replaces the usage of `std::string&`, and promises faster string usage. Libraries should use it, and the user will not notice a difference except faster code. (If you use ROOT, you've probably already seen errors mentioning `string_view` when you didn't match C++ versions correctly; this comes from ROOT trying to backport `string_view`.)

Parallel standard library algorithms

Not available in STL yet

Not available in any open source compiler yet! You can use the Intel Parallel STL, however, which is planned to be proposed for merge into both GCC and Clang.

You can now set an execution policy on algorithms in the standard library; sequential, parallel, and vector are allowed. It is integrated into the algorithms library; you just add the execution policy as the first argument to the algorithm. For example:

```
std::vector<double> vals = {1.0, 2.0, 3.0, 4.0};
auto my_square = [](double value){return value*value;}
std::for_each(std::parallel::par_vec, vals.begin(), vals.end(), my_square);
```

Most of the old algorithms support these execution policies, and several new algorithms have been added to cover parallel usage.

Note for Gaudi users

Remember, if you are already in a multithreaded application such as Gaudi, you may not get much benefit from multithreading further.

Also, the Gaudi framework will use a Threaded Building Blocks (TBB) backend to run the components multithreaded, which means that individual components should not be also multithreaded with a different scheme. If these eventually play together more nicely, this warning may go away.

Filesystem

GCC 8

This was just added in libstdc++ that comes with GCC 8, so this is not available to LHCb users yet.

The powerful Boost file system has finally made it into the standard library, after three revisions and promises every three years. This allows object oriented syntax for file manipulation, and works on Windows/Mac/Linux. However, it probably will not be integrated into libraries very quickly, requiring casting to C-style strings to be used. Should still be very useful. If you are familiar with Python's pathlib or a similar library, this will seem familiar.

A simple example of the new filesystem:

```
auto file_in_path = std::filesystem::current_path() / "file.txt";
file_in_path.replace_extension(".log");
if(std::filesystem::exists(file_in_path))
    ...
```

Here we see that the `/` operator joins path segments, the paths have useful member functions, and the non-member functions can operate on the underlying filesystem. There exists a full set of filesystem operations. Generally, any documentation you find for `Boost::filesystem` Version 3 should also be applicable.

Assorted standard library additions

There are a few smaller additions to the standard library too, like the addition of special math functions, such as Bessel functions. If you have ever needed to clamp a value between two limits, `std::clamp(value, min, max)` avoids nested min/max calls or duplication of the value. `std::lcm` and `std::gcd` have been added as mathematical functions, as well.

A few new helper functions make dealing with generalized classes easier, including `std::invoke`, which can call any callable, `std::not_fn` which can negate any callable, and tuple's apply, which applies a tuple as arguments to a callable.

Non-member versions of common tasks, `std::size`, `std::empty`, and `std::data` have been added to join the `std::begin` and `std::end` family.

There are also a set of modifications to existing features, listed in the official document that are too small to list here.

Further reading

There are other changes that are either too small to discuss here, or are not likely to affect the average particle physicist. Like C++14, many of them polish the newer portions of the language. The official overview is excellent. A very good overview can be found on StackOverflow.

Also see Herb Sutter's site for reports on the progress made at the C++ standards meetings for information about C++20 and beyond!

New features in the distant future for LHCb Physicists

Learning Objectives

- "Learn a few of the plans for C++"

C++ is on a 3 year cycle, with groups working on Technical Specifications (TS's). Many TS's were proposed for C++17, but failed to make the cut. They may make it into a later version, however, and may need support from users like you. So here are the most interesting plans:

Parallelism II

The parallel TS that was accepted has an extension that was not; it it planned for the future. This add a few things, like auto-deduction of the best execution policy.

Concepts

This TS was sorely missed; even though it doesn't directly affect code, it does something even more important: It improves error messages and clarifies templated code structure. Although the syntax is a bit dense (it's C++, after all), what it does is simple. If you write a templated function, you probably have some concept of what the parameters should be. This allows you to specify that upfront. You can say something like "I expect the type to be a number" or "I expect to the type to have a `.Save()` and `.Load()` methods", and the error message if this fails will occur in the right portion of the code, and will clearly define the problem. As a library user, you will only notice that error messages improve; as a library author, you will need to think about what template expect and add concepts to them in order to benefit.

Modules

C++'s biggest remaining problem is the horrendous, hack of a system from the 1970's called `#include`. It only knows how to copy and paste code when compiling, causing huge, ugly, long compiles. A real module system would make combining code and compiling cleaner and faster (by a lot). This was not quite ready for 2017, but was close.

Ranges

Ranges uses concepts so heavily that it was not going to go in without it. It can define a range (like 10-20) of an object (like a vector) without wasting memory. It also would allow the standard library to work on ranges, which would be defined for things like `std::vector`, allowing the following notation:

```
std::for_each(vect.begin(), vect.end(), do_something);
```

to change to:

```
std::for_each(vect, do_something);
```

Of course, the point would be to do something like this:

```
std::for_each(vect[{2,5}], do_something);
```

which would take the second up to the fifth elements of `vect` and only operate on those. I'm assuming vectors will support view's iterating and slicing here, otherwise you might need to wrap a vector in a view from Ranges.

Reflection

This is not really even a TS yet, just a Study Group (SG). However, it is a huge topic, and it being pushed forward by the ROOT team among others. It would allow a C++ class to know what it is, just like a Python class does, and would eliminate/fix/simplify most of what ROOT does, like the saving of arbitrary classes. ROOT's dictionary generator is just a hack for the missing Reflection feature. Read more here. Like several of the newest proposals, it uses Concepts.

Compile-time code

This would allow a programmer to execute arbitrary code blocks at compile time; due to similarity with constexpr, this is proposed to look like this:

```
constexpr {
    // The code in the block executes during compilation
}
```

This would be useful for reflection (since you can process the members of a reflection enabled class), and with a third new feature, **injection**, you could even inject new methods at compile time. That brings us to our next new feature...

Metaclasses

This is a radical proposal that is exciting everyone, but requires a series of features to be added (reflection, compile-time code, and injection) that are still being worked on. If this was added, however, it could replace the custom dictionary generators of ROOT and Gaudi Object Description (or just about any other system), it could provide simple ways to add things like Python bindings, and much more. All of this inside a framework that would be easier to debug than templates. It could also unify many of the other concepts in C++, like classes vs. structs, into a single conceptual framework. See Herb Sutter's site for more on the proposal.

This proposal would allow new versions of the "class" keyword to be added that add new functionality and set new defaults; even generate several classes from one definition. For example, you could remake the existing `struct` just using a metaclass definition that sets the defaults to public and adds default constructors.

Smaller changes

The `operator.` proposal for C++17 needed a last minute change, and may or may not make it into C++17 as a special case. It allows smart references just like we currently have smart pointers.

There has been some work on unified call syntax, which would allow `x.something(y)` to fall back to `something(x,y)` and vice versa.

Some experimental libraries that may make it in eventually are Transactional Memory, Concurrency, Networking, Coroutines, Graphics, and Numerics.

"C++ Performance"

Learning Objectives

- "Learn about fast coding."

Optimizing code

One of the goals of coding is to make clean, clear code that closely matches the intent of the user, both in language and syntax. Really, almost all languages and features are truly optional; programs could technically be written with a tiny subset of any modern computing language. The point of classes, exceptions, structures, and the like is to make code match intent in a manor obvious to the user. Another goal of writing code is to be able to use one piece of code in multiple cases, avoiding rewriting or maintaining multiple copies. A related goal is to be able to create it quickly with minimal hassle.

Optimization may seem to be exactly the opposite of these goals, obfuscating your original intent, making performant versions for each use case, and extending the work needed to create the code; but when carefully performed, optimization can have a minimal impact in these. There are several key points to take away:

Know where your code is taking time

This is key; most algorithms have a bottleneck somewhere; it's been said that 1% of your code tend to take 99% of the time. It is important to **profile** your code to understand where time is being spent. Only optimize the portions that take the most time; it is not worth the impact to readability to optimize something that takes less than 5% of the total time.

There are simple tools for profiling, as well as complex. The obvious manual methods, using `std::chrono::high_resolution_clock::now()` and printing times, might be faster for a single use, but in the long run will take far more of your time than learning a profiler.

The following are three common classes of profilers.

Code insertion

These modify the execution of the program, such as `callgrind`. This causes your program to run much slower, and can affect the proportion of time spent in calls. Small calls or IO tend to be heavily impacted by using these tools.

Example of callgrind

The following example shows the procedure for obtaining a graphical result of the time spent in each portion of your program:

```
local:build $ g++ -g -O2 -std=c++11 -pedantic -Wall -Wextra my_program.cpp -o my_program
local:build $ valgrind -tool=callgrind -dump-instr=yes -collect -jumps=yes -cache -sim=yes -branch-sim=yes ./my_program
local:build $ kcachegrind
```

Stack sampling profilers

These sample the program stack during execution (`gprof` , `google-perftools` , `igprof`), providing minimal changes to the runtime of the program. These trade knoledge of every call for a representation of a normal run.

Examples of sample profiling

Examples of sample profiling

To use, you'll want the compiler to add profile info (`-pg`) and optionally debug info (`-g`). You also may want the compiler to avoid inline functions (`-fno-inline`). The binary can now be run through gprof:

```
local:build $ gprof ./my_program
```

Google also has a profiling tool:

```
$ CPUPROFILE=prof.out LD_PRELOAD=/usr/lib/libprofiler.so.0 ./my_program
$ google-pprof -text ./my_program prof.out
```

Another option is igprof, which is available on the CERN stack. In comparison with the other sampling profilers, it provides several unique advantages. The following are benefits and drawbacks compared to gprof:

- Does not require special compilation flags.
- More robust when monitoring programs that do nontrivial stuff such as calling libraries or spawning new processes and threads.
- Has a fancy HTML frontend for reports.
- Much faster sampling rate -> misses less, but produces bigger output.
- It is a bit hard to use outside of CERN (e.g. no package in the Debian repos).
- Does not seem to have an option to produce line-by-line annotated source, a gprof feature which can be quite handy in perf analysis

Compared to valgrind/callgrind:

- Does not run your code in a virtual machine -> enormously faster, however output is less precise (e.g. no cache info).
- Does not bias your performance profile by inflating the relative cost of CPU w.r.t. IO by a factor of 100.

Example of igperf

You can profile a program as follows:

```
local:build $ igprof -d -pp -z -o my_program.pp.gz my_program
local:build $ igprof-analyse -d -v -g my_program.pp.gz >& my_program.pp.out
```

The first line produces a profile, the second line converts it to a readable text report.

Kernel sampling profilers

These are similar to the other sampling profilers, but they use extra information from the Linux kernel when sampling, such as CPU performance counters. This leads them to the following benefits and drawbacks:

- They are hopelessly unportable to non-Linux OS's. Compare them with Xcode Instruments on macOS and the Windows Performance Toolkit.

- The information that they provide is hardware and kernel-specific, and in particular they won't run at all if your Linux kernel is too old.
- They tend to make the system protection features of modern Linux distros (e.g. SELinux) go crazy, making it a pain to get them to run.
- As a compensation for all these drawbacks, they promise much more detailed information, comparable to what one can usually only get through callgrind, thanks to the use of hardware performance counters.
- And they can also do system-wide profiling, which is useful when studying interactions between "unrelated" processes.

Some examples are `linux-perf` and `oprofile`.

Example of Kernel profiling

You can also use the kernel and CPU to help (may require root privileges):

```
local:build $ perf record ./my_program
local:build $ perf report ./my_program
```

Keep clarity a focus

Your code should have well defined, independent blocks that do as few jobs each as possible, and it should be structured to match a user's expected behavior (good documentation can allow minor deviations from this). This is, in almost all cases, something you should not have to sacrifice for speed. Package highly optimized code inside these blocks, with a clear indication of what they are supposed to do. Good function/class/etc names go a long way. Try to avoid "ugly" coding practices, as described below.

Stay general as long as possible

If you need to write an algorithm, think of how it can be broken up. Try to solve everything generally first. If you have code that has to sort a list, it's better to separate the list sorting from your algorithm. Then, you can often find an optimised library version of the general tasks; you can write unit tests that test each part; and you can keep optimizations localised to the code they affect. Sometimes you cannot do this, but always try first.

Use template features when possible to perform tasks at compile time. Modern C++ has `constexpr` expressions that allow compile time computation to be done, which allow you to avoid hard coding calculated numbers in the code, but to leave the original expressions. You can also use macros as a last resort.

Use the language

Some common issues with writing C++ are:

- Using globals. Don't.
- Forgetting `const`. This should be used for every interface, and in C++11, is an indication of multithread safety.
- Custom memory management inside a framework (like Gaudi) that does that for you. This can be made into a more general rule: if a framework provides something, use it. Then the framework can be upgraded, and your code will receive the benefits.
- Try to support and use pass by reference and move semantics. This allows you to avoid copying large data structures many times.
- Using `iterator++` instead of `++iterator`, the first is often making a copy wastefully.
- Forgetting to use `noexcept` if you can't throw an exception; this can allow the compiler to do more optimization for you.

Common speedups

The following are common ways to speed up computation.

- Precompute expressions, and reuse the results. If you use the expression `sin(x)*cos(y)` multiple times, you can save processor steps by precomputing it to a temporary variable. Note that if you do this excessively, you might end up with temporary variables in the main memory instead of the cache, so look for many usages or something that takes many processor cycles.
- C++ allows return value optimization; if you return an object created in the function, and it otherwise would go out of scope and be destroyed, it is moved out of the function instead of copied. (Note: This is the case with most compilers for C++11, and is required by the language in most cases for C++17). Make sure all your return statements return the same object for this optimization. (Unnamed object get this automatically, too)
- Perfect forwarding: this is a handy trick in several cases; as a common example, you can construct an object inside a vector instead of copying it in, using `emplace_back`. The method forward the arguments after the first to the constructor (the first argument), but makes it in-place inside the vector instead of a move or copy.
- Watch for slow general functions, like `pow(x,2)`, that are designed to be flexible at runtime, when a simple compile time version `x*x` is available. Profile!

Approximations

Pay attention to the level of precision you need; if you can make an approximation to an expensive calculation, this can reduce your computing time significantly. Be careful, though, often it can be difficult to speed up functions in the standard library or high-performance libraries; test and measure any improvements you try to make.

To multithread or not to multithread

One of the hottest topics in programming is the use of multiple threads; on a multithreaded computer or a GPU, the promise of speedup by factors of 2+ is enticing. But there are conditions to these gains besides the obvious requirement that the procedure must be able to be done in parallel; if you have a memory bottleneck for example (very common), that will remain the limiting factor. Another common bottleneck is I/O; that is often improved by threads even on a single core processor.

If you are working inside a framework, like much of the code for LHCb, and that framework is already multithreaded, it is often slower to then try to introduce more threads; the other CPUs are already busy. Much of the work for these components is in making them **thread-safe**, that is, making them able to run in parallel with other algorithms or with themselves without conflicts for writing/reading memory.

Multithreading basics in C++11

Multithreading is difficult for most languages to handle well, since we write source code in a linear fashion. There are several common methods for multithreading in C++11:

Thread

With `std::thread`, you can run a function immediately in parallel, with perfect forwarding. Values are not returned; you will need to pass in pointers, etc. to get results.

```
std::thread t1(my_function, my_argument1);
// Starts running along side code
t1.join(); // Finish the function and return.
```

Future

You often want to start a function, go do something else, and then get the result of that function. Futures and promises allow you to do that. A promise is a value that you promise to give at some point, but does not necessarily have a value right now. A future is an object associated with that promise that lets you wait for the result and retrieve it when it is ready. Here is some pseudocode (see similar example here):

```
void some_slow_function(std::promise<int> p) {
    std::this_thread::sleep_for(5s);
    p.set_value(42);
}

std::promise<int> p;
std::future<int> f = p.get_future();

std::thread t(some_slow_function, std::move(p));
// Could do other things here
std::cout << "The result is " << f.get() << std::endl;
t.join();
```

Async

This usage of futures can be made much easier in some cases using `std::async`. Notice that the last example had to have a special `some_slow_function` that worked with a promise. Often you will just want to wrap an existing function that slow, and returns a value. The previous example then becomes:

```
int some_slow_function(int time) {
    std::this_thread::sleep_for(time * 1s);
    return 42;
}

std::future<int> f = std::async(std::launch::async, some_slow_function, 5);
// Could do other things here
std::cout << "The result is " << f.get() << std::endl;
```

Data Races

Multithreading's biggest issue tends to be data races. Let's make a pseudocode example:

```
int x = 0
void thread_1() {
    x += 1;
}
void thread_2() {
    x -= 1;
}
// Run thread_1 and thread_2 in parallel
```

What is the value of `x` after running? The expected answer is 0, since 1 is being added, and 1 is being subtracted. But when you run this, you'll randomly get -1, 0, and 1 as an answer. This is because you have to read the value of x in (one operation) and then add 1 to it in the register (one operation) then return it to the `x` memory location (one operation). When the operations are running in parallel, you might load the value in the second thread before the write happens in the first thread, giving you the unexpected results seen previously.

There are several ways to manage these values. Besides futures and promises, which can be used between threads, the following low level constructs are available.

Mutexes

A mutex such as `std::mutex` allows a lock to be placed around segments of code that must run sequentially. For example, you may notice that `std::cout` tends to get mangled when multiple threads are printing to the screen. If you place a mutex around each output, the mangling will no longer be an issue. For example:

```
std::mutex m; // Must be the same mutex in the different threads

m.lock(); // This line only completes when m is not locked already
std::cout << "A line" << " that is not interleaved as long as protected by m" << std::endl;
m.unlock();
```

If you forget to unlock the mutex, your code will stall forever when it comes on a new lock statement. A `std::lock_guard` will accept a mutex, locking it immediately, and then unlocking when the lock guard goes out of scope. If you have if statements or code that can throw exceptions, this might be easier than making sure `.unlock` is called every place it is needed.

Mutexes solve the issue of linearising a piece of code, but at a huge cost: they make that part of your code completely sequential and also have some small overhead too.

Atomics

If you are worried about setting and accessing a single value, a faster and simpler method than mutexes are atomics. These often have hardware level support, allowing them to run faster than a matching mutex. These are special version of values that insure that reads and writes never collide. The only difference vs. a normal variable is the use of `.load()` to access the value (other operations are overloaded to behave as expected). The data race example becomes:

```
std::atomic<int> x = 0
void thread_1() {
    x += 1;
}
void thread_2() {
    x -= 1;
}
// Run thread_1 and thread_2 in parallel
```

And the code always gives a result of `x.load() == 0`.

Conditions

The final method that can be used to communicate between threads is condition variables. This behaves a bit like an atomic, but can be waited on until a thread "notifies" that the variable is ready.

Further reading

- Performance in C++ for LHCb
- LHCb Code Analysis Tools

Python in the upgrade era

Learning Objectives

- "Learn about the new Python."

A new Python

About ten years ago, Guido Van Rossum, the Python author and Benevolent Dictator for Life (BDFL), along with the Python community, decided to make several concurrent backward incompatible changes to Python 2.5 and release a new version, Python 3.0. The main changes were:

- Using unicode strings as default, with the old string type becoming a full featured binary type
- Changing several builtins, for example
  - The print statement became a function, allowing more consistent syntax and the use of the word print as a name
  - The confusing `input` removed, and `raw_input` now renamed to `input`
  - Simpler `exec`
  - Division is now split between `/` float division and `//` truncating division
- Improved exception tracing, with chaining
- Improved function call syntax with annotations and keyword only arguments, replacing little used tuple parameter unpacking
- More class constructor features, such as nicer metaclass syntax, keyword arguments, `__prepare__`
- Renamed standard libraries, to be more consistent
- Removal of a lot of depreciated features, including old-style classes
- Removal of a lot of depreciated syntax that had become learner stumbling blocks
- Adding `nonlocal` variables
- Extended tuple unpacking, like `first, *rest = makes_a_tuple()`
- Removing the proliferation of `.pyc` files, instead using `__pycache__` directories
- Automatic selection of C-based standard library modules over pure Python ones if available
- Unified the `int` and `long` types into one unlimited length integer type

Unfortunately, this list was comprehensive enough to break virtually every python script ever written. So, to ease the transition, 3.0 and 2.6 were released simultaneously, with the other, backward compatible new features of 3.0 being also included in 2.6. This happened again with the releases of 3.1 and 2.7. Not wanting to maintain two Pythons, the BDFL declared that 2.7 was the last Python 2 series release.

Side note about speed

These changes (mostly the unicode one) also made Python much slower in version 3.0. Since then, however, there have been many speed and memory improvements. Combined with new C extensions for some modules, Python 3 is now usually as fast or faster than Python 2.

The original, officially sanctioned upgrade path was one of the biggest issues with moving to Python 3. A script, `2to3`, was supposed to convert code to Python 3, and then the old version could be eventually dropped. This script required a lot of manual intervention (things like the unicode strings require knowledge of the programmer's intent), and required library authors to maintain two separate versions of the code. This hindered initial adoption with many major libraries unwilling to support two versions for Python 3 support.

Unofficial authors tried making a new script, `3to2`, which worked significantly better, but still was hindered by the dual copies of code issue.

Another decision also may have slowed adoption. Part way through the development of Python 3.2 up to 3.4, the decision was made to avoid adding any new features, to give authors time to adopt code to a stable Python 3. This statement could be taken in reverse; why update to Python 3 when it does not have any new features to improve your program? The original changes (as listed above) were not enough to cause mass adoption.

This dreary time in Python development is now drawing to a close, thanks to a change in the way authors started approaching Python compatibility. There is such a good overlap between Python 2.6 or Python 2.7 and Python 3.3+ that a single code base can support them both. The reason for this is the following three things:

- Good Python 2 is almost the same as Python 3. The things that were dropped were mostly things you shouldn't do in Python 2 anyway.
- Several changes in syntax are available in Python 2 using `__future__`
- The remaining changes can mostly be wrapped in libraries

These were capitalized by the unofficial library authors, and now almost every library is available as a single code base for Python 2 and 3. Most of the new standard libraries, and even a few language features, are regularly backported to Python 2, as well.

Libraries to ease in the transition

Six

The original compatibility library, six (so named because 2 times 3 is 6), provides tools to make writing 2 and 3 compatible code easy. You just import six, and then access the renamed standard libraries from `six.moves`. There are wrappers for the changed features, such as `six.with_metaclass`.

These features are not hard to wrap yourself, so many libraries implement their own `six` wrapper to reduce dependencies and overhead.

Future

This is a newer library with a unique approach. Instead of forcing a usage of a special wrapper, the idea of future is to simply allow code to be written in Python 3, but work in Python 3. For example, `from builtins import input` will do nothing on Python 3 (`builtins` is where `input` lives), but on Python 2 with `future` installed, `builtins` is part of `future` and will import the `future` version. You can even patch in the Python 3 standard library names with a `standard_library.install_aliases()` function.

Future also comes with it's own version of the conversion scripts, called `futurize` and `pasteurize`, which use the future library to make code that runs on one version run on both versions. An alpha feature, the `autotranslate` function, can turn a library that supports only Python 2 into a Python 3 version on import.

Backports

Several of the new libraries and features have been backported to Python 2. I'm not including ones that were backported in an official Python release, like `argparse`.

- `pathlib`: A simple, object oriented path library from Python 3.4
- `enum`: A python package for enumerations from Python 3.4
- `mock`: A version of `unittest.mock` from Python 3.3
- `futures`: This is the `concurrent.futures` package in Python 3.2
- `statistics`: From Python 3.4
- `selectors34`: The `selectors` package from Python 3.4
- `typing`: Type hints from Python 3.5
- `trollius`: The `asyncio` package, with a new syntax for `yield from`, from Python 3.4

- Smaller changes: `configparser`, `subprocess32`, `functools32`, and the various `backports`-dot-something packages.

New features in modern Python

These are features that have been released in a version of Python after 3.0 that are not in the older Python 2 series:

- Matrix multiplication operator, `@` (3.5)
- Special `async` and `await` syntax for asynchronous operations (3.5, 3.7)
- Unpacking improvements, so that the `*` and `**` operators work in more places like you'd expect (3.5)
- Function signatures now in easy to use object (3.3)
- Improvements to Windows support (Windows launcher, recent versions of Visual C++) (3.2, 3.4, 3.5, 3.6, 3.7)
- Delegation to a subgenerator, `yield from`, finally allows safe factorisation of generators (3.3)
- Context variables and AsyncIO improvments, include a simple run function (3.7)
- `importlib.resources`, which allows files that don't end in `.py` to be accessed (FINALLY!) (3.7, backports available)
- `breakpoint()` built-in function for debugging (3.7)
- Modules can have custom `__dir__` and `__getattr__` (3.7)
- Lots of new debugging options in CPython for developers, like timing module import and better stacktraces (3.7)

Formatted string literals (3.6)

Finally! You can write code such as the following now:

```
x = 2
print(f"The value of x is {x}")
```

This is indicated by the `f` prefix, and can take almost any valid python expression. It does not have the scope issues that the old workaround, `.format(**locals())` encounters.

Syntax for variable annotations (3.6)

This will be great for type hints, IDE's, and Cython, but the syntax is a little odd for Python. It's based on function annotations. A quick example:

```
an_empty_list_of_ints: List[int] = []
will_be_a_str_later: str
```

This stores the variable name and the annotation in an `__annotations__` dictionary for the module or the class that they are in.

Simi-ordered dictionaries (3.6 and 3.7)

Python dictionaries are now partially ordered; due to huge speedups in the C definition of ordered dicts, the `dict` class is now guarantied to iterate in order as long as nothing has been changed since the `dict` creation. This may sound restrictive, but it enables many features; you can now discover the order keyword arguments were passed, the order class members were added, and the order of `{}` dicts. If you want to continue to keep or control the order, you should move the `dict` to an `OrderedDict`, as before. This makes ordered dictionaries much easier to create, too.

Warning

Only class member order and keyword argument order are ensured by the language; the ordering of `{}` is an implementation detail. This detail works in both CPython 3.6 and all versions PyPy, however. This became language mandated in Python 3.7.

DataClasses (3.7)

Most programmers coming from other languages want some form of class designed to store data. Creation of these data-centric classes is verbose and ugly in python, since you hace to put all the setup in the `__init__` method rather than directly in the class like other languages, and you have to manage initilization, print, comparison, etc. yourself. Now, with DataClasses, you can do it with a nice syntax:

```
from dataclasses import dataclass

@dataclass
class Vector:
    x: float
    y: float
    z: float
```

This will create (by default) `__init__`, `__repr__`, and `__eq__`. You can also ask for `order`, `unsafe_hash`, and `frozen`.

This is similar to, and less powerful than, the popular attrs library (available for all versions of Python). This library module, like many others, was also backported to older versions of Python. However, the variable type annotations are not available in older versions.

Other smaller features:

- Underscores in numeric literals. You can add arbitrary spacers to numbers now, such as `1_000_000`.
- Windows encoding improvements.
- Simpler customization of class creation, using `__init_subclass__` class method.
- Descriptor access to the name of the class and the descriptor, using `__set_name__`.
- A file system path protocol, `__fspath__()`, allows any object to indicate that it represents a path. Finally pathlib works without wrapping it in a `str()`!
- Better support for `async` list comprehensions, and `async` generators.
- A `secrets` module for password related randomization functions.

Status of Python

The current status of the python releases is as follows:

- Python 2.5: Dead.
- Python 2.6: Most libraries are dropping support, officially discontinued, but still on some legacy systems, like the default environment in SL6.
- Python 2.7: The officially supported Python 2 release, critical security flaws fixed till roughly 2020. PyPy supports 2.7.13. Windows version is stuck requiring Visual Studio 2008 for builds (Careful memory design can allow use of new VS)
- Python 3.0-3.2: Never used significantly, no library support.
- Python 3.3: Better backwards compatibility makes this the first generally used Python 3, with Windows downloads outpacing Python 2.7 for the first time. `u""` was added back in as a no-op.
- Python 3.4: Addition of asyncio features and pathlib provided even more interest.
- Python 3.5: New features, such as matrix multiplication, are accelerating the transition from Python 2. Note that PyPy3 is currently based on Python 3.5.3.
- Python 3.6: The addition of format strings make simple scripts much easier and cleaner.
- Python 3.7: Big performance improvements make this the fastest CPython ever; dataclasses, typing, and threading improvements.

Further reading

- The old Python wiki page

- What's new in Python
- 10 awesome features of Python that you can't use