Програмні коди для фізики високих енергій

Денис Лонтковський

Зміст лекції

- Огляд ролі програмного забезпечення в HEP
  - Історія та еволюція програмних засобів
  - Ознайомлення з обчислювальними задачами в HEP
- Огляд мов програмування, які часто використовуються в HEP (C++, Python)
- Контроль версій коду та спільна розробка з Git.

Introduction to software in HEP

- HEP encompasses the study of fundamental constituents of matter and their interactions
- The research field is mostly focused on three fundamental forces: electromagnetic, weak and strong
- Particle accelerators, such as Large Hadron Collider, enable researchers to study fundamental particles interactions in high-energy collisions.
- Software forms the integral part of all HEP experiments and is used in all aspects, including, data acquisition, analysis and simulations

Domains of software development in HEP. Detector control and data acquisition

- Detectors control (slow and fast): Software enables automated control of multiple detectors comprising detector systems such as CMS/ATLAS

- Data management software handles massive influx of the data generated from collisions as well as from simulations

- Software plays crucial role in high-level triggers to determine which events to be recorded

Domains of software development in HEP. Data analysis

- LHC experiments, CMS and ATLAS generate petabytes of data which require advanced algorithms for efficient handling and analysis.
- State-of-the-art techniques and software frameworks are used to extract physically significant information from massive amount of data.
- Multiple detectors outputs must be combined using efficient algorithms to obtain as complete as possible information about high-energy collisions.

Domains of software development in HEP. Simulations

- Simulation software holds a pivotal role in High-Energy Physics to unfold underlying physics.
- Often Monte Carlo methods are used to simulate both hard collisions and interactions of the collision products with detectors.
- Simulations are also typically the only source of the information for estimating uncertainties in experimental results.

History of the software in HEP

- The history of software in HEP is closely tied to the evolution of computing technologies

- **1960s-1970s: The Birth of HEP Software**
  - The 1960s saw the introduction of the first digital computers in HEP. FORTRAN (Formula Translation) was the dominant programming language, due to its efficiency in handling numerical calculations. The simulations, implemented in FORTRAN, allowed physicists to model particle interactions and detector responses, providing a critical tool for interpreting experimental results.

- **1980s-1990s: The Rise of General-Purpose Software Frameworks**
  - The 1990s witnessed the emergence of object-oriented programming languages, particularly C/C++. This shift was driven by the need for more modular and maintainable code in increasingly complex HEP experiments. During this period, the ROOT data analysis framework was developed at CERN, providing an object-oriented replacement for older FORTRAN-based tools. ROOT, written in C++, became the standard for data analysis in HEP and remains widely used today.

- **2000s-Present: Modern Software Challenges**
  - As experiments like the LHC at CERN generated unprecedented amounts of data. This necessitated advancements in data management, parallel computing, and grid computing. The Worldwide LHC Computing Grid (WLCG) was established to provide the necessary computational resources, involving a global network of data centers.
  - In recent years, Python has gained popularity in HEP for its ease of use and extensive ecosystem of scientific libraries. While C++ remains dominant in performance-critical applications, Python is increasingly used for prototyping, data analysis, and scripting.

C++ and Python crash-course

C++ and python. Common programming languages in HEP

- **Common Applications:**
  - C++ is most often used in high-performance computing, while python is predominant as scripting language for data analysis.
  - C++ typically offers superior performance for resource-intensive tasks, whereas Python prioritizes rapid development.
- **Syntax:**
  - C++ features a more complex syntax requiring explicit memory management compared to Python's intuitive simplicity.
  - C++ uses braces and semicolon ; as statements/instructions delimiters
  - Python relies on proper indentation in control structures

Basic Syntax Comparison

- C++ Syntax Example:
- To print 'Hello, World!' in C++, use:

```cpp
#include <iostream>

int main()
{
    // Prints hello world
    std::cout << 'Hello, World!'
              << std::endl;
    return 0;
}
```

```
> g++ main.cpp && ./a.out
```

- Python Syntax Example:
- To print 'Hello, World!' in python, use:

- `print('Hello, world')`

- `python3`

Variables and Data Types

- In C++, variables must be explicitly declared with their type before usage.
  - For example:
  - `double pion_mass = 139.570;`
- However, C++ compiler can also infer variable type during parsing/translation
  - For example:
  - `auto n_flavors = 5;`
- Basic data types include:
  - `int`, `float`, `double`, `char`
  - `std::string`, `std::vector<>`, `std::map<>`

Variables and Data Types

- Python has dynamic typing, meaning variables can be assigned without explicit type declarations.
  - Examples:
  - `pion_mass = 139.570`
  - `collider = 'LHC’`

- Python basic objects types:
  - `int, float, str, list, dict`

C++ control structures (conditionals): If-Else

- In C++, conditional structures like **if, else if**, and **else** allow decision-making based on conditions
- Example:

```cpp
std::string particleFlavor = "up";

if (particleFlavor == "up") {
    std::cout << "Particle is an Up quark";
} else if (particleFlavor == "down") {
    std::cout << "Particle is a Down quark";
} else if (particleFlavor == "strange") {
    std::cout << "Particle is a Strange quark";
} else {
    std::cout << "Unknown particle flavor";
}
```

C++ control structures (conditionals): Switch

- Alternative for long **if, else if, else** conditions is **switch**
- Example:

```
enum Flavor {UP, DOWN, STRANGE};
Flavor particleFlavor = UP;

switch (particleFlavor) {
    case UP:
        std::cout << "Particle is an Up quark";
        break;
    case DOWN:
        std::cout << "Particle is a Down quark";
        break;
    case STRANGE:
        std::cout << "Particle is a Strange quark";
        break;
    default:
        std::cout << "Unknown particle flavor";
        break;
}
```

Python control structures (conditionals): If-elif

- Python equivalent of conditional control structure has less variations
- Example:

```python
particle_flavor = "up"

if particle_flavor == "up":
    print("Particle is an Up quark")
elif particle_flavor == "down":
    print("Particle is a Down quark")
elif particle_flavor == "strange":
    print("Particle is a Strange quark")
else:
    print("Unknown particle flavor")
```

Python control structures (conditionals): match/case

- Python3.10 equivalent of switch control structure
- Example:

```python
particle_flavor = "up"

match particle_flavor:
    case "up":
        print("Particle is an Up quark")
    case "down":
        print("Particle is a Down quark")
    case "strange":
        print("Particle is a Strange quark")
    case _:
        print("Unknown particle flavor")
```

C++ control structures (loops):

- Iterations in c++ can be described in different ways

```
int numberOfEvents = 5;

for (int event = 0; event < numberOfEvents; event++) {
    std::cout << "Processing event number: " << event <<
std::endl;
}
```

C++ control structures (loops):

- Iterations in c++ can be described in different ways

```
auto particleEnergy = 100.0; // Starting energy in GeV
auto particleEnergyLoss = 10.0; // Energy lost per step

while (particleEnergy > 0) {
    std::cout << "Particle energy: " << particleEnergy << " GeV" << 
std::endl;
    particleEnergy -= particleEnergyLoss;
}
```

C++ control structures (loops):

- Iterations in c++ can be described in different ways

```
int hits = 0;
int maxHits = 10;

std::srand(std::time(0)); // Seed for random number generator

do {
    hits = std::rand() % 20; // Generate random number of hits (0-19)
    std::cout << "Number of hits in detector: " << hits << std::endl;
} while (hits < maxHits);
```

Python control structures (loops):

- Similar control structures available in python too
- Example:

```
collision_energies = [100, 200, 300, 150, 250]

# Iterate through each event and print the energy
for i, energy in enumerate(collision_energies, 1):
    print(f"Event {i}: Energy = {energy} GeV")
```

Python control structures (loops):

- Similar control structures available in python too
- Example:

```
electron_energy = 100.0
bremsstrahlung = 10.0  # Energy lost per photon emission

# Simulate decay until the particle energy reaches zero or
below
while electron_energy > 0:
    print(f“Electron energy: {electron_energy:.1f} GeV")
    particle_energy -= bremsstrahlung
```

Functions declaration

- In both C++ and Python, functions are fundamental building blocks for structuring programs and reusable code
- Example C++:

```cpp
// Function to calculate energy from particle's four-momentum vector (E, px, py, pz)
double calculateEnergyFromFourVector(double px, double py, double pz, double mass) {
    // E^2 = (px^2 + py^2 + pz^2) + m^2
    return std::sqrt(std::pow(px, 2) + std::pow(py, 2) + std::pow(pz, 2) + std::pow(mass, 2));
}
```

Example python:

```python
def calculate_energy_from_four_vector(px, py, pz, mass):
    # E^2 = (px^2 + py^2 + pz^2) + m^2
    return math.sqrt(px**2 + py**2 + pz**2 + mass**2)
```

Elementary object-oriented programming

- Both languages support object-oriented paradigm based on **classes**

- C++ and python support inheritance, and polymorphism. C++ manages memory through explicit constructors and destructors. Python offers dynamic duck-typing.

C++ class declaration

```
class Particle {
private:
    double px, py, pz; // Momentum components in GeV/c
    double mass;       // Mass in GeV/c^2

public:
    // Constructor
    Particle(double px, double py, double pz, double mass)
        : px(px), py(py), pz(pz), mass(mass) {}

    // Method to calculate energy from four-momentum vector (E^2 = p^2 +
m^2)
    double calculateEnergy() {
        return std::sqrt(px * px + py * py + pz * pz + mass * mass);
    }

};
```

Python class declaration

```
class Particle:
    def __init__(self, px, py, pz, mass):
        # Initialize momentum components (GeV/c) and mass (GeV/c^2)
        self.px = px
        self.py = py
        self.pz = pz
        self.mass = mass

   # Method to calculate energy from four-momentum vector (E^2 = p^2 +
m^2)
    def calculate_energy(self):
        return math.sqrt(self.px**2 + self.py**2 + self.pz**2 +
self.mass**2)
```

Memory management

- Memory management is a critical aspect of programming, especially in languages like C++ where developers need to manually allocate and free memory.
- In contrast, Python provides automatic memory management through garbage collection, making it easier for developers to focus on logic without worrying about memory leaks.
- In C++, dynamic memory is allocated using the **new** keyword, which reserves space on the heap. The programmer **must** explicitly free this memory using the **delete** keyword to prevent memory leaks.
- Python abstracts away memory management from the user. It uses **automatic memory management** through a garbage collector, which automatically handles allocation and deallocation of memory. When objects are no longer referenced, Python's garbage collector frees up the memory, preventing memory leaks

C++ dynamic memory allocation

```
int* arr = new int[10];  // Dynamically allocate an array of 10 integers
// ... Use the array ...
delete[] arr;  // Free the allocated memory
```

However, there are much better ways of memory management in C++

```
// Using std::unique_ptr to manage a dynamically allocated array
std::unique_ptr<int[]> uniqueArray = std::make_unique<int[]>(10);
// Now create a shared_ptr that shares ownership of a different array
std::shared_ptr<int[]> sharedArray(new int[10], std::default_delete<int[]>());
// Create a weak_ptr that observes the shared_ptr without owning it
std::weak_ptr<int[]> weakArray = sharedArray;
```

Comparison of smart pointers in c++

Smart Pointer	Ownership	Copyable	Movable	Thread-Safe	Use Case
std::unique_ptr	Exclusive	No	Yes	N/A	Managing a resource with sole ownership
std::shared_ptr	Shared	Yes	Yes	Yes	Sharing ownership of a resource
std::weak_ptr	Non-owning	Yes	Yes	Yes	Non-owning reference to a shared_ptr to avoid circular dependencies

Collaborative software development.
Git

Introduction

- Git is a version control system for code/text.
- Git's allows multiple users to work on code simultaneously.
- Tracking modifications in source code ensures traceability, accountability and facilitates incremental development.
- Git streamlines collaborative efforts by allowing branching, merging, and conflict resolution among contributors effectively.

Git init/clone

- Initializing a Repository: Executing `git init` prepares a new Git repository, establishing all essential files for tracking.
- Directory Modification: When `**git init**` is run, the current directory becomes monitored by Git for version control purposes.
- Alternatively, when project code already exists at SCM server, the repository can be cloned to the development station
  `**git clone <url>**`

Git Commit

- Recording Changes: The `**git commit**` command records changes, storing snapshots of project history within the repository.

- Descriptive Messages: A meaningful message, provided with `-m`, is crucial for understanding the context of changes made. Typically, WHY code has been
added or modified.

- Example Syntax: Use the syntax:
- ```bash git commit -m 'Initial commit' ```
- to initiate your project's first commit.

Git Branch

- The `**git branch**` command facilitates the creation, listing, and deletion of branches in repositories.
- Feature Development Isolation: Branches enable developers to work on distinct features independently, preserving the stability of the production codebase.
- For instance, executing
```bash git branch new-feature ```
creates a branch for a new feature development.

Git Pull

- The `git pull` command updates the local repository by combining fetched changes with local branches.
- Usage Example: To implement, use: ```bash git pull origin main ``` to update the main branch from remote.

Git Log

- The `git log` command displays comprehensive commit history, including commit IDs, authors, dates, and messages.

- Usage Example:
```bash git log```
to view a detailed timeline of repository commits.

Git Push

- The `git push` command transfers local commits to a specified remote repository, enabling collaboration.
- Using `git push` enables contributors to synchronize their changes
- Usage Example:
  ```bash git pull origin main ```
  to update the main branch on remote git server