C++ AND ROOT

J. STROLOGAS

GOAL OF THESE LECTURES

- Provide the basic C++ functionality that is useful for ROOT
- Provide the basic ROOT functionality that is useful for Data Analysis
- Provide a couple of Data-Analysis tasks

GOAL OF THESE LECTURES

- Provide the basic C++ functionality that is useful for ROOT
  - First Lecture
- Provide the basic ROOT functionality that is useful for Data Analysis
  - Second Lecture
- Provide a couple of Data-Analysis tasks
  - Third Lecture

SOME PREREQUISITES

- LINUX
  - We will work at a Linux environment
- C
  - We will not cover basic C syntax
- A short overview will be provided as well

LINUX BASIC COMMANDS (1)

- pwd
  - Prints working directory (directory=folder)
- ls
  - Lists files and directories in the current directory, ls –lt lists them in chronological order
- cd …/…/…
  - Changes directory (the path ../../../ has to be provided, or just the local directory)
- mkdir …
  - Creates the directory … (a full path can be provided)

LINUX BASIC COMMANDS (2)

- rm …
  - Removes a file, e.g., rm a.txt
- cp … …
  - Copies one file to another name, e.g. cp a.txt b.txt
- mv … …
  - Renames files/directories or moves them to new location, e.g, mv a/ b/ or mv a.txt b.txt (file a.txt disappears)

C BASIC SYNTAX (1)

- Variable types: double/float/int/char/string
  - e.g., int a=4;
- Flow control: if, if-else, switch
  - e,g., if(a==0){}
- Loops: for, while-do, do-while
  - e.g., for(int i=0; i<N; i++){}
- Functions
  - e.g., double f(double x, int y){… return a;}
- Arrays: [], [][]
  - e.g., double a[3] or int b[4][6]

C BASIC SYNTAX (2)

- Structures
  - e.g., struct point{double x; double y;}; struct point a; a.x=10.;
- Input from keyboard/file: scanf, fscanf
  - e.g., scanf("%lf",&a) or fscanf(fp,"%lf",&a)
- Output to monitor/file: printf, fprintf
  - e.g., printf("a = %f",a) or fprintf(fp,"a = %f",a)
- Pointers
  - double x=10; double *a = &x;

(If you want to review the C syntax in a couple of hours → https://www.w3schools.com/cpp/default.asp)

C++ ADDITIONS WE WILL SEE (1)

- Output stream to monitor: cout <<
  - e.g., cout << a << endl;
- Output stream to file: f <<
  - e.g., ofstream f("outFile.txt"); f << a << endl;
- Input stream from keyboard: cin >>
  - e.g., cin >> a;
- Input stream from file: f >>
  - e.g., ifstream f("inputFile.txt"); f >> a;

(One can use the corresponding C commands as well (printf, scanf, etc.))

C++ ADDITIONS WE WILL SEE (2)

- Most importantly → CLASSES
  - A way to group variables of all kind, including functions (methods) and other classes
  - The information is organized in a tree structure because mainly of inheritance
  - Inheritance allows for non duplication of information
  - Everything becomes an object (which may have other objects etc.)
    - Object is a programming entity that includes both data and functions
    - The objects allow for data to be hidden and be accessed with the object's methods
    - Very useful in ROOT. For example, the axis of a histogram is an object that has other objects (e.g. title of histogram), which has other objects e.g. the method that controls the size of the text
  - A CLASS is a description of an object
    - *Each object that is created from a class is called an instance of the class*

ADVANTAGE OF CLASSES

- Better organization of information and communication with computer
  - It's an important step towards AI (you say "car" and the computer knows what you mean)
- Code is easier to maintain and to upgrade
  - The methods of the classes can change → this is transparent to the user
- Better handling of memory
  - All the information that is related is grouped in the same area in memory
- There is no duplication of code, because of inheritance
  - If you create a more specific class it can inherit from a more general class
    - the variables and methods of the general class can be used by the specific class

STRUCTURE OF CLASS

```
class className{

private:
  variables and methods that cannot be accessed directly
public:
   variables and methods that can be accessed directly
}
```

- The private members cannot be accessed directly, but ONLY through public methods (called “accessors”)

METHODS

- Inline implementation of a method
  - The body of the method is written inside the class declaration

- Out-of-line implementation of a method
  - The body of the method is written outside the class declaration
    - In that case the syntax is ReturnType ClassName::MethodName(method arguments)

EXAMPLE → CLASS CAR

```
#include <iostream>
#include <string>

using namespace std;

class car{

private:

   int year;
   string brand;
   string model;

public:

   int getYear(){return year;}
   string getBrand(){return brand;}
   string getModel(){return model;}

   void setYear(int y);
   void setBrand(string b);
   void setModel(string m);

   void printCar(){

      cout << brand << " " << model << " " << year << endl;

   }

};

void car::setYear(int y){
   year = y;

}
void car::setBrand(string b){
   brand = b;

}
void car::setModel(string m){
   model = m;

}
```

private members of class

public members of class

accessor methods

methods defined inline

methods defined out of line

EXAMPLE → CLASS CAR USAGE (SAME FILE)

```
#include <iostream>
#include <string>

using namespace std;

class car{

private:

  int year;
  string brand;
  string model;

public:

  int getYear(){return year;}
  string getBrand(){return brand;}
  string getModel(){return model;}

  void setYear(int y);
  void setBrand(string b);
  void setModel(string m);

  void printCar(){

    cout << brand << " " << model << " " << year << endl;

  }

};

void car::setYear(int y){
  year = y;

}
void car::setBrand(string b){
  brand = b;

}
void car::setModel(string m){
  model = m;

}
```

pointer pointing to instance of the class

```
int main(){

  car myCar_1;
  car *myCar_2 = new car;

  myCar_1.setBrand("Toyota");
  myCar_1.setModel("Corolla");
  myCar_1.setYear(2020);
  myCar_1.printCar();

  myCar_2->setBrand("Mitsubishi");
  myCar_2->setModel("Eclipse");
  myCar_2->setYear(2009);
  myCar_2->printCar();



}
```

We will compile with
g++ code1.cc –o code1
and we will run with
./code1

Result of run:

```
Toyota Corolla 2020
Mitsubishi Eclipse 2009
```

SEPARATING CLASS SPECIFICATION FROM IMPLEMENTATION

- Having the class declaration, methods implementation, and class usage in one file IS NOT good programming practice
- For that reason, we include the class declaration in a header file (car.h) which is the class-specification file
- The methods definitions are in a separate file (car.cc) called class-implementation

CREATE SEPARATE FILES FOR CLASS

car.h

```
#include <iostream>
#include <string>

using namespace std;

class car{

private:

  int year;
  string brand;
  string model;

public:

  int getYear(){return year;}
  string getBrand(){return brand;}
  string getModel(){return model;}

  void setYear(int y);
  void setBrand(string b);
  void setModel(string m);

  void printCar(){

    cout << brand << " " << model << " " << year << endl;

  }

};
```

car.cc

```
#include <iostream>
#include <string>
#include "car.h"

using namespace std;

void car::setYear(int y){
  year = y;

}
void car::setBrand(string b){
  brand = b;

}
void car::setModel(string m){
  model = m;

}
```

We will first create the object car.o with

g++ –c car.c

and then we will produce the executable with

g++ code2.c car.o –o code2

code2.cc

```
#include <iostream>
#include <string>
#include "car.h"

using namespace std;

int main(){

  car myCar_1;
  car *myCar_2 = new car;

  myCar_1.setBrand("Toyota");
  myCar_1.setModel("Corolla");
  myCar_1.setYear(2020);
  myCar_1.printCar();

  myCar_2->setBrand("Mitsubishi");
  myCar_2->setModel("Eclipse");
  myCar_2->setYear(2009);
  myCar_2->printCar();



}
```

CONSTRUCTORS AND DESTRUCTORS

- Constructors are methods that are called every time you create an instance of a class
- They have the same name as the class (and they are public)
- They can be “overloaded” (the same function can have versions with different number and kind of arguments)
  - The constructor with no arguments is called “default constructor”

- The Destructor is called when the instance of the class is destroyed (memory is freed)
  - The name of the destructor is the name of the class with ~ in front

INHERITANCE

- To make the communication with the computer even more natural (and avoid duplication of code) a class can inherit from another class
  - For example, the class car can inherit from the class vehicle, which can have its own methods and variables
- The member variables and methods of the parent class (and grandparent class etc.) become members of the children classes
  - If the class vehicle has a member “fuel” then an instance of the class car has also this member

CREATE A NEW CLASS → VEHICLE

```
#include <string>

using namespace std;

#ifndef VEHICLE_H
#define VEHICLE_H

class vehicle {

private:

  float fuel;//lt
  float engineVolume;//cm^3

public:

  vehicle(){fuel=0;engineVolume=0;}
  ~vehicle(){}

  float getFuel(){return fuel;}
  float getEngineVolume(){return engineVolume;}

  void setFuel(float f){fuel=f;}
  void setEngineVolume(float ev){engineVolume=ev;}


};
#endif
```

- All methods inline
  - Otherwise, I had to have the corresponding .cc file and compile to create the object
- Include guard so that the include is not loaded more than once

USE INHERITANCE (VEHICLE)

car_2.h

```
#include <iostream>
#include <string>
#include "vehicle.h"

using namespace std;

class car : public vehicle{

private:

  int year;
  string brand;
  string model;

public:

  car(){

    year=0; brand=""; model="";

  }
  ~car(){};

  int getYear(){return year;}
  string getBrand(){return brand;}
  string getModel(){return model;}

  void setYear(int y);
  void setBrand(string b);
  void setModel(string m);

  void printCar(){

    cout << brand << " " << model << " " << year << endl;
    cout << this->getEngineVolume() << " mm^3 " << this->getFuel() << " lt " << endl;

  }

};
```

Inheritance

Constructor and Destructor

Access the pointer to the class

car_2.cc

```
#include <iostream>
#include <string>
#include "car_2.h"

using namespace std;

void car::setYear(int y){
  year = y;

}
void car::setBrand(string b){
  brand = b;

}
void car::setModel(string m){
  model = m;

}
```

We will first create the object car_2.o with

g++ –c car_2.c

and then we will produce the executable with

g++ code3.c car_2.o –o code3

code3.cc

```
#include <iostream>
#include <string>
#include "car_2.h"

using namespace std;

int main(){

  car myCar_1;
  car *myCar_2 = new car;

  myCar_1.setBrand("Toyota");
  myCar_1.setModel("Corolla");
  myCar_1.setYear(2020);
  myCar_1.setEngineVolume(1600);
  myCar_1.setFuel(23.4);
  myCar_1.printCar();

  myCar_2->setBrand("Mitsubishi");
  myCar_2->setModel("Eclipse");
  myCar_2->setYear(2009);
  myCar_2->printCar();


}
```

Result of run:

```
Toyota Corolla 2020
1600 mm^3 23.4 lt
Mitsubishi Eclipse 2009
0 mm^3 0 lt
```

WE WILL TRY ALL THIS CODE IN PRACTICE

- First create a directory for this lab
  - mkdir lab_1
- Go to this directory
  - cd lab_1
- Open emacs
  - emacs code1.cc &