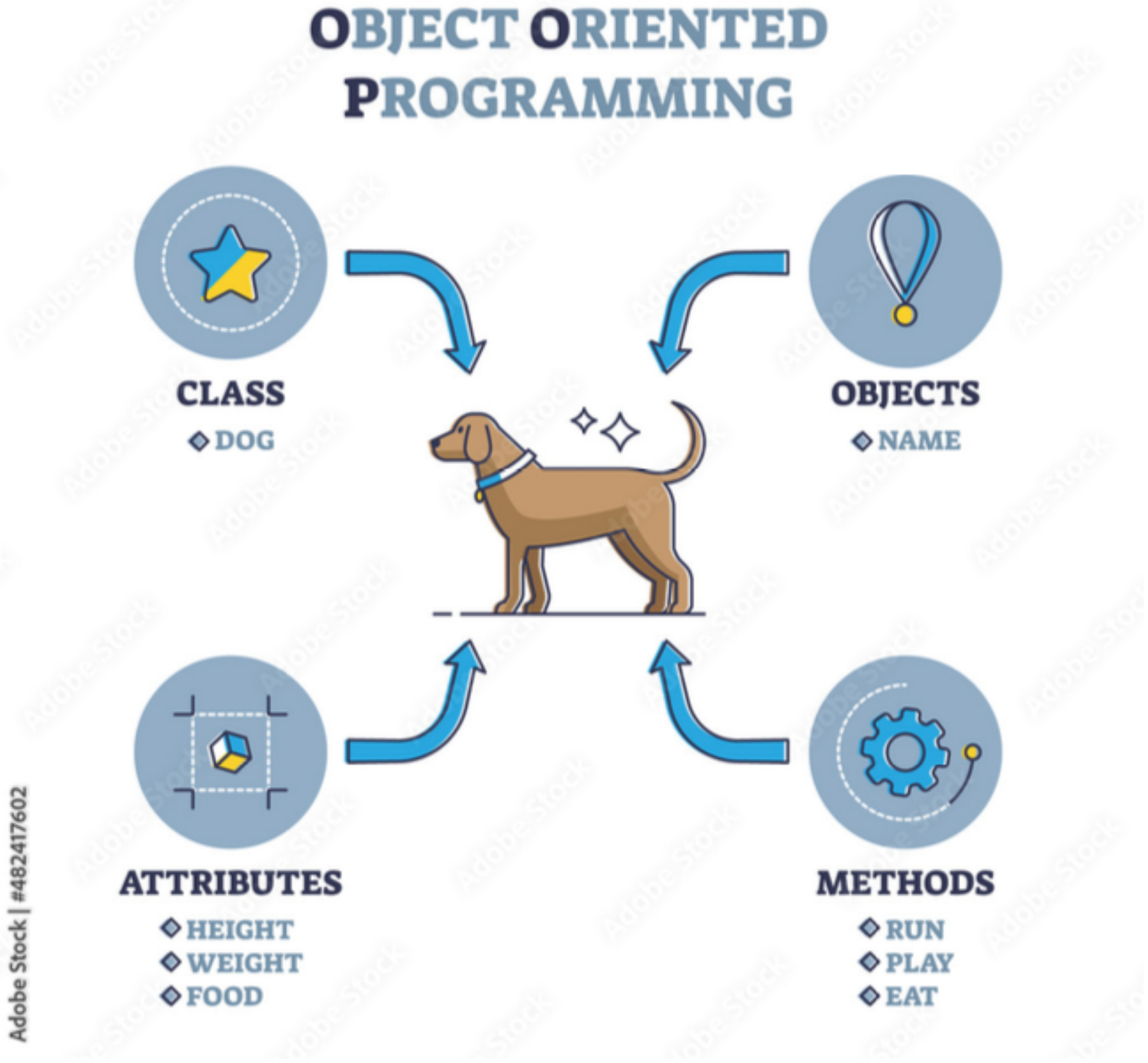
Queen Mary
University of London

Graduate Computing Course

Gitanjali Poddar
21/11/2025

Part I: Intro to C++


OBJECT ORIENTED PROGRAMMING
CLASS
DOG
OBJECTS
NAME
ATTRIBUTES
HEIGHT
WEIGHT
FOOD
METHODS
RUN
PLAY
EAT
Adobe Stock | #482417602

A First C++ Code

```
#include <iostream>

using namespace std;

class BasicRectangle
{
public:
    // width ;
    float W ;
    // length
    float L ;
};

int main()
{
    cout << "Hello world!" << endl;

    BasicRectangle rectangle ;
    rectangle.W = 1.0 ;
    rectangle.L = 2.0 ;

    return 0;
}
```

BASIC SYNTAX

- Curly braces are used to denote a code block
- Statements end with a semicolon
- Comments are marked with //
- Case-sensitive

A First C++ Code (II)

```
#include <iostream>

using namespace std;

class BasicRectangle
{
public:
    // width ;
    float W ;
    // length
    float L ;
};

int main()
{
    cout << "Hello world!" << endl;

    BasicRectangle rectangle ;
    rectangle.W = 1.0 ;
    rectangle.L = 2.0 ;

    return 0;
}
```

Importing *header* that stores pre-defined functions. `<iostream>` is for input-output functions

Loading *namespace* to prevent naming conflicts in large projects

- Without "`using namespace std`", this would have been called as "`std::cout`". It is defined in the `iostream` header file.
- `<<` is the insertion operator
- `endl` is the newline character

A First C++ Code (III)

```
#include <iostream>

using namespace std;

class BasicRectangle
{
public:
    // width ;
    float W ;
    // length
    float L ;
};

int main()
{
    cout << "Hello world!" << endl;

    BasicRectangle rectangle ;
    rectangle.W = 1.0 ;
    rectangle.L = 2.0 ;

    return 0;
}
```

- `class` keyword to create a *class*
- `public` keyword is *access specifier*. It specifies that variables and functions are accessible from outside the class.
- *Variables* are declared with type (`int, double, float, char, bool, string`) and name.
- Class definition ends with a semicolon

→ Create *object* of class and access variables

A First C++ Code (IV)

```
#include <iostream>

using namespace std;

class BasicRectangle
{
public:
    // width ;
    float W ;
    // length
    float L ;
};
```

```
int main()
{
    cout << "Hello world!" << endl;

    BasicRectangle rectangle ;
    rectangle.W = 1.0 ;
    rectangle.L = 2.0 ;

    return 0;
}
```

→

Functions have a return type, name and list of arguments. Here,

- `int` is the return type
- `main` in function name
- No arguments

As in Python

Operators

Operators	Type
`++, – –`	Increment/decrement
`+, –, *, /, %`	Arithmetic
`<, <=, >, >=, ==, !=`	Relational
`&&, ||, !`	Logical

Other operators also exist. These are the most common ones.

As in Python

Control Statements

if-else blocks

```
if (condition)
{
    // Executes this block if
    // condition is true
}
else
{
    // Executes this block if
    // condition is false
}
```

loops

```
for(initialization; check/test expression; updation)
{
     // body consisting of multiple statements
}
```

```
while (test expression)
{
   // body consisting of multiple statements
}
```

- `break:` jump out of loop
- `continue:` control moves to next iteration of loop

Other control statements also exist.
These are the most common ones.

As in Python

Arrays

- Used to store multiple values in a single variable
- Arrays are declared by the variable type, name and number of elements in square brackets
  - `int num[4]={2,5,8,8};`
  - `string words[2][3]={{“cat”, “mat, “hat”},{“bat”, “fat”, that”}};`
  -
```
int numbers[5];
for (int i=0; i<5; i++){
        numbers[i]=10;}
```
 → 1) `for` loops are used to traverse array
 2) array indices start at 0
- `sizeof()` operator can be used to get array size

Datatype Modifiers

- Used with built-in data types to modify the length of data stored

- `signed`: target type will have signed (+/-) representation
  `unsigned`: target type will have unsigned representation
  short: target type will have at least 16 bits
  long: target type will have at least 32 bits

Data Type	Size (in bytes)	Range
`short int`	2	-32768 to 32767
`unsigned short int`	2	0 to 65535
`long long int`	8	-2^{63} to $2^{63}-1$
`unsigned char`	1	0 to 255

References and Pointers

- A *reference* to a variable is created using the & operator

```
string particle=“up”;
string &quark=particle;

cout<<particle<<endl;
cout<<quark<<endl;
cout<<&particle<<endl;
```

→ What do they output?

References and Pointers

- A *reference* to a variable is created using the & operator

```
string particle="up";
string &quark=particle;

cout<<particle<<endl; gives "up"
cout<<quark<<endl; gives "up"
cout<<&particle<<endl; gives 0c6ed54 (memory address of the variable)
```

- *Pointer* is a variable that stores memory address. It has the same data type as the variable and created with * operator

```
string particle="up";
string *ptr=&particle;
cout<<ptr<<endl; gives 0c6ed54

cout<<*ptr<<endl;
*ptr="down";
cout<<*ptr<<endl;
cout<<particle<<endl;
```

→ What do they output?

References and Pointers

- A *reference* to a variable is created using the & operator

```
string particle=“up”;
string &quark=particle;

cout<<particle<<endl; gives “up”
cout<<quark<<endl; gives “up”
cout<<&particle<<endl; gives 0c6ed54 (memory address of the variable)
```

- *Pointer* is a variable that stores memory address. It has the same data type as the variable and created with * operator

```
string particle=“up”;
string *ptr=&particle;
cout<<ptr<<endl; gives 0c6ed54

cout<<*ptr<<endl; gives “up”
*ptr=“down”;
cout<<*ptr<<endl; gives “down"
cout<<particle<<endl; gives “down”
```

References and Pointers

- A *reference* to a variable is created using the & operator

```
string particle=“up”;
string &quark=particle;

cout<<particle<<endl; gives “up”
cout<<quark<<endl; gives “up”
cout<<&particle<<endl; gives 0c6ed54 (memory address of the variable)
```

References and pointers give the ability to **manipulate data in computer’s memory**, which can reduce the code and improve performance

- *Pointer* is a variable that holds the memory address of another variable

```
string particle=“up”;
string *ptr=&particle;
cout<<ptr<<endl; gives 0c6ed54

cout<<*ptr<<endl; gives “up”
*ptr=“down”;
cout<<*ptr<<endl; gives “down"
cout<<particle<<endl; gives “down”
```

Summary

- C++ is an **object oriented programming** language
- **Typical typos**: missing semi-colon, case-sensitive..
- Variables, functions, arrays are **declared by data type and name**
- Typical to C++:
  - **Data modifiers** that change the range/length of built-in data types
  - **References and pointers** that access computer's memory

Part I: Intro to ROOT

Introduction

- Framework developed at CERN for
  - data visualisation: graphs, histograms, trees
  - data analysis: statistical tools (`RooStats, RooFit`), multivariate analysis (TMVA)
  - data storage: store **any C++ object**
- Based on C++
  - Python bindings are provided (PyR00T)
- R00T:
  - Install locally: https://root.cern/install/
  - Use remote machines: CERN cluster, SWAN notebooks, etc

First Steps

To launch ROOT

```
[gipoddar@lxplus977 ~]$ root
   ------------------------------------------------------------------
  | Welcome to ROOT 6.32.06                                https://root.cern |
  | (c) 1995-2024, The ROOT Team; conception: R. Brun, F. Rademakers |
  | Built for linuxx8664gcc on Sep 23 2024, 00:00:00                 |
  | From tags/6-32-06@6-32-06                                        |
  | With g++ (GCC) 11.4.1 20231218 (Red Hat 11.4.1-3)                |
  | Try '.help'/'.?', '.demo', '.license', '.credits', '.quit'/'.q'  |
   ------------------------------------------------------------------

[root [0] .q
 [gipoddar@lxplus977 ~]$
```

To quit ROOT

Data Types

- Storage space for standard data types like `int, bool, char`, etc depend on machine and compiler. ROOT data types are **machine independent.**

- First letter is capitalised and ends with "`_t`"

`int`→`Int_t` `float`→`Float_t` `double`→`Double_t`

Signed	Unsigned	Size (in bytes)
`Char_t`	`UChar_t`	1
`Short_t`	`UShort_t`	2
`Int_t`	`UInt_t`	4
`Long64_t`	`ULong64_t`	8
`Float_t`		4
`Double_t`		8
`Double32_t`		

Takeaway: You can use `int` or `Int_t`, but the latter is preferable

Classes

- All classes start with a ’T’ (type)
  - `TString`: class to handle strings. It has more features than `std::string`. Note: no `String_t` in ROOT
  - TH∗: class to handle 1D, 2D and 3D histograms
  - `TTree`: class to handle large datasets
  - `TObject`: class to handle objects
  - `TFile`: class to handle files
  - `TDirectory`: class to handle directory like structure of files

These are some common classes only

Classes (II)

Typical syntax:

```
class_name object_name (arguments)
class_name *object_name = new class_name (arguments)
```

- **Dot**: used to access members of objects

```
[root [0] TString s("particle")
 (TString &) "particle"[8]
[root [1] s.Length()
 (int) 8
 root [2]
```

Declaring object

- **Arrow**: used to access members of pointers to objects

```
[root [2] TString *s_ptr=new TString("particle")
 (TString *) 0x560fb9779ad0
[root [3] s_ptr->Length()
 (int) 8
```

Declaring object pointer

https://root.cern.ch/doc/master/classTString.html

Typical syntax:
`class_name object_name (arguments)`

Histograms

- TH∗ classes represents histograms
  - TH1∗ and TH2∗ represents 1-dimensional and 2-dimensional histograms
  - The final letter represents the variable type stored in each histogram. Eg: TH1D is double, TH2F is float

Object name Key name Title name n_{bins} x_{min} x_{max}

```
root [11] TH1F example("h1","Histogram",5,10,50)
 Warning in <TROOT::Append>: Replacing existing TH1: h1 (Potential memory leak).
 (TH1F &) Name: h1 Title: Histogram NbinsX: 5
root [12] example.Fill(12)
 (int) 1
root [13] example.Fill(26)
 (int) 3
root [14] example.Fill(35)
 (int) 4
root [15] example.Fill(33,0.25)
 (int) 3
root [16] example.Fill(25,0.01)
 (int) 2
root [17] example.Draw()
 Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
```

Declaring histogram

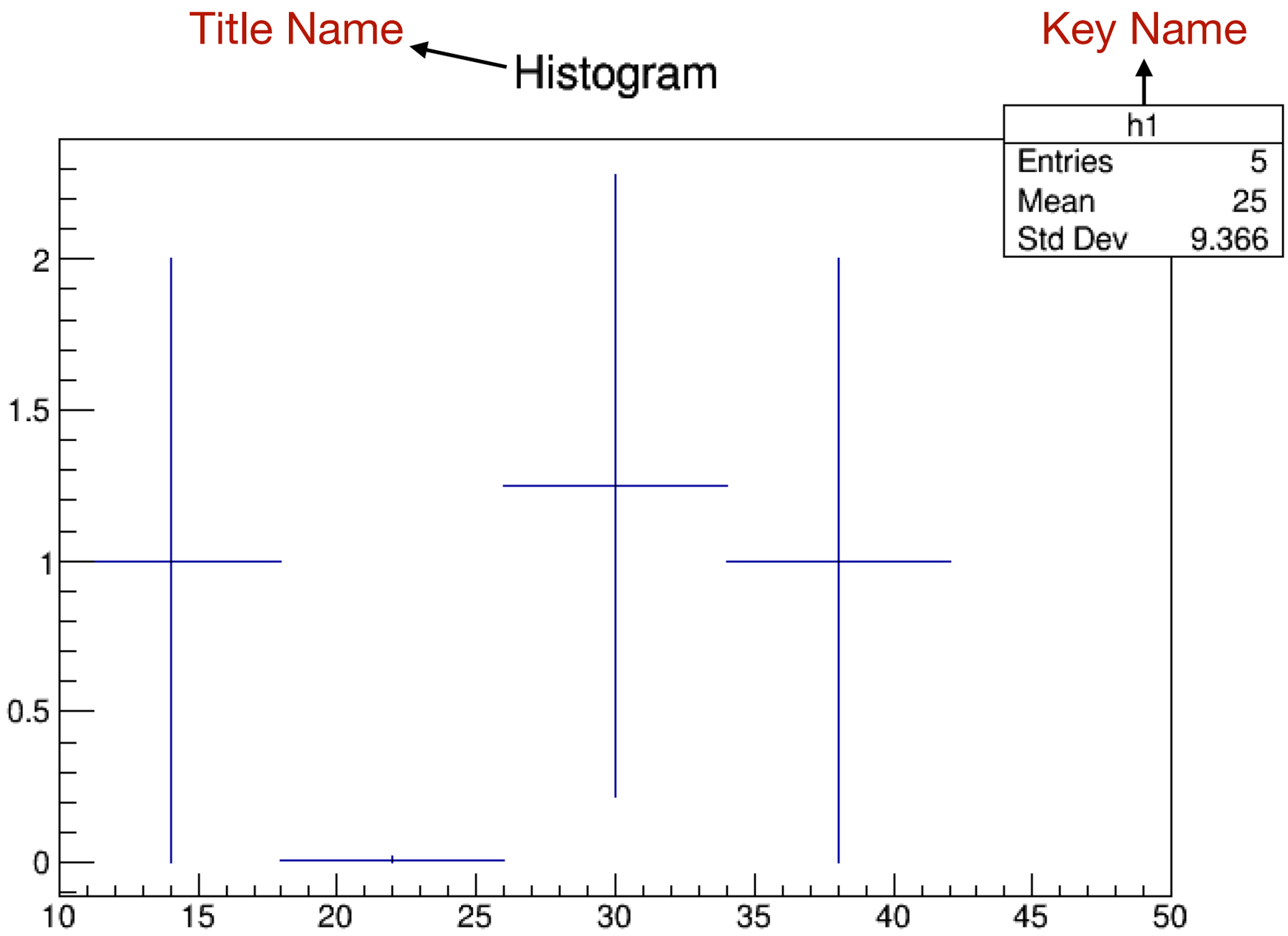
Filling histogram

Filling histogram with weight

Drawing histogram

https://root.cern.ch/doc/master/classTH1.html

Histograms (II)



Typical syntax:
`class_name object_name (arguments)`

File: Input

Object name File name

```
[root [18] TFile out_file("example.root","NEW")
 (TFile &) Name: example.root Title:
[root [19] TH1D h("h1","Histogram",5,10,500)
 (TH1D &) Name: h1 Title: Histogram NbinsX: 5
[root [20] h.Write() Writing/saving object to file
 (int) 266
[root [21] out_file.Close() Closing file
```

- "RECREATE": create a ROOT file, replacing it if it already exists
- "CREATE" or "NEW": create a ROOT file
- "UPDATE": updates the ROOT file
- "READ": opens an existing ROOT file for reading

https://root.cern.ch/doc/master/classTFile.html

Typical syntax:
`class_name object_name (arguments)`

File: Read

```
root [1] TFile f1("example.root","READ")
 (TFile &) Name: example.root Title:
root [2] .ls
TFile**         example.root
 TFile*         example.root
  KEY: TH1D     h1;1    Histogram
root [3] h1->Print("all")
TH1.Print Name  = h1, Entries= 0, Total sum= 0
  fSumw[0]=0, x=-39
  fSumw[1]=0, x=59
  fSumw[2]=0, x=157
  fSumw[3]=0, x=255
  fSumw[4]=0, x=353
  fSumw[5]=0, x=451
  fSumw[6]=0, x=549
root [4]
```

Reading file

Inspecting contents of file

Printing object in file

Directory

`TFile` behaves like file system, inheriting methods from `TDirectory`

```
[root [6] f1.mkdir("directory1")
 (TDirectory *) 0x56006f6e6660
[root [7] .ls
 TFile**         example.root
  TFile*         example.root
   OBJ: TH1D     h1       Histogram : 0 at: 0x56006fb78970
   TDirectoryFile*                 directory1      directory1
   KEY: TH1D     h1;1     Histogram
[root [8] f1.cd("directory1")
 (bool) true
[root [9] .ls
 TDirectoryFile*         directory1      directory1
 root [10]
```

Making directory

Changing directory

https://root.cern.ch/doc/master/classTDirectory.html

Objects

- Mother of all ROOT objects (histograms, trees, n-tuples..)
- Common functions used frequently:
  - `Draw`: to visualise the object
  - `Print`: to print contents of the object
  - `Write`: to save contents of the object
  - `Clone`: to copy contents of the object. This is also one of the ways to create an object.

Trees

- It is made of *branches* (sub-directories) that can correspond to the different variables

- `Print("all")`: print all branches of the tree

```
[TFile**         Zllyjj_EW_reco.root
  TFile*         Zllyjj_EW_reco.root
   KEY: TTree    overlap;1       overlap
root [2] overlap->Print("all")
[******************************************************************************
*Tree    :overlap   : overlap                                                  *
*Entries :    18654 : Total =          863679 bytes  File  Size =     321307 *
*        :          : Tree compression factor =   2.69                       *
******************************************************************************
*Br    0 :event_number : event_number/l                                       *
*Entries :    18654 : Total  Size=     150163 bytes  File Size  =      54886 *
*Baskets :        5 : Basket Size=      32000 bytes  Compression=   2.73     *
*.............................................................................*
*Br    1 :run_number : run_number/I                                           *
*Entries :    18654 : Total  Size=      75361 bytes  File Size  =        720 *
*Baskets :        3 : Basket Size=      32000 bytes  Compression= 103.97     *
*.............................................................................*
*Br    2 :region    : vector<int>                                             *
*Entries :    18654 : Total  Size=     337437 bytes  File Size  =      37409 *
*Baskets :       13 : Basket Size=      32000 bytes  Compression=   9.00     *
*.............................................................................*
*Br    3 :centrality : centrality/D                                           *
*Entries :    18654 : Total  Size=     150145 bytes  File Size  =      86123 *
*Baskets :        5 : Basket Size=      32000 bytes  Compression=   1.74     *
*.............................................................................*
*Br    4 :weights   : weights/D                                               *
*Entries :    18654 : Total  Size=     150118 bytes  File Size  =     141042 *
*Baskets :        5 : Basket Size=      32000 bytes  Compression=   1.06     *
*.............................................................................*
root [3]
```

Printing contents of tree

Branches of tree

https://root.cern.ch/doc/master/classTTree.html

Trees (II)

- It is made of *rows* that can correspond to the different entries
- `Scan("branch_name")`: print rows of branch name

```
root [3] overlap->Scan("event_number:centrality")
************************************
*    Row   * event_num * centralit *
************************************
*        0 *     70966 * 0.3314801 *
*        1 *     70018 * 0.3017931 *
*        2 *     71988 * 0.5246904 *
*        3 *     70583 * 0.0860082 *
*        4 *     71713 * 0.2576033 *
*        5 *     71814 * 0.2144374 *
*        6 *     70877 * 0.2528469 *
*        7 *     70083 * 0.1753834 *
*        8 *     71320 * 0.2671529 *
*        9 *     71171 * 0.0335564 *
*       10 *     70531 * 0.0695489 *
*       11 *     71861 * 0.2872475 *
*       12 *     71601 * 0.0534878 *
*       13 *     70491 * 0.0570811 *
*       14 *     71329 * 0.0379767 *
*       15 *     71167 * 0.0949674 *
*       16 *     70220 * 0.0979169 *
*       17 *     71458 * 0.0502022 *
*       23 *     70104 * 0.0366933 *
*       24 *     71051 * 0.0353275 *
Type <CR> to continue or q to quit ==>
```

Scan multiple variables at same time

```
root [4] overlap->Scan("centrality<0.2")
************************
*    Row   * centralit *
************************
*        0 *         0 *
*        1 *         0 *
*        2 *         0 *
*        3 *         1 *
*        4 *         0 *
*        5 *         0 *
*        6 *         0 *
*        7 *         1 *
*        8 *         0 *
*        9 *         1 *
*       10 *         1 *
*       11 *         0 *
*       12 *         1 *
*       13 *         1 *
*       14 *         1 *
*       15 *         1 *
*       16 *         1 *
*       17 *         1 *
*       18 *         0 *
*       19 *         1 *
*       22 *         0 *
*       23 *         1 *
*       24 *         1 *
Type <CR> to continue or q to quit ==>
```

Make cut on variables

Exercise

- Open “signal.root”
- List its contents
- Plot any one of the histograms
- Print contents of tree
- Scan any branch of the tree
- Scan any two branches of the tree
- Quit ROOT

Feel free to play around with the file!

Solutions

- Open “signal.root”: `root -l signal.root`
- List its contents: `.ls`
- Plot any one of the histograms: `cossphericity->Draw()`
- Print contents of tree: `selectedtree->Print(“all”)`
- Scan any branch of the tree: `selectedtree->Scan("pi0pi0Tagger")`
- Scan any two branches of the tree: `selectedtree->Scan(“pi0pi0Tagger:rhoPiFisher")`
- Quit ROOT: `.q`

Macros

- The standard procedure is to write code in macros/scripts and then run it.
  Note: '**;**' can be ignored when working with ROOT on shell, but not in macros

- To run a macro:

- `root -l Signal_Macro.C`

- `root`
  `.x Signal_Macro.C`

Macros (II)

```
using namespace std;
#include <iostream>

#include "TMath.h"
#include "TTree.h"
#include "TTreeReader.h"
#include <TFile.h>
#include <TH1D.h>
```

Standard C++ header and namespace

Import ROOT classes

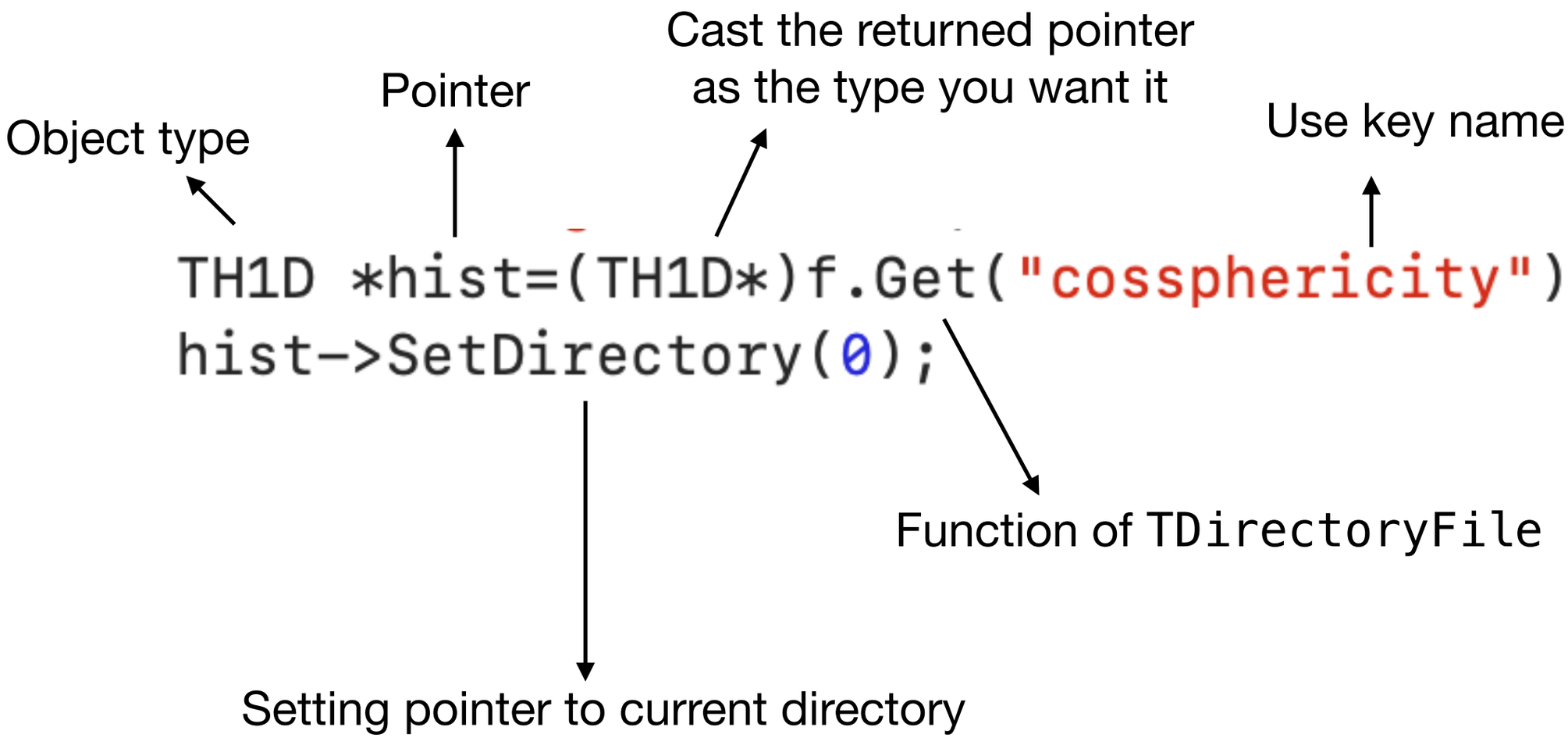
Note: can be imported with <TClass.h> or “TClass.h”

```
void Signal_Macro(){

 TFile f("signal.root");
 TH1D *hist=(TH1D*)f.Get("cossphericity");
 hist->SetDirectory(0);

 hist->Rebin(5);
 hist->Scale(1/hist->Integral());
 hist->Print("all");
 hist->Draw();

 f.Close();

}
```

→ main function has same name as macro

Note: use of . or –> depending on object or pointer

Macros (II)



Note: standard syntax to extract `TObject` from a file

Macros (III)

```
using namespace std;
#include <iostream>

#include "TMath.h"
#include "TTree.h"
#include "TTreeReader.h"
#include <TFile.h>
#include <TH1D.h>


void Signal_Macro2(){

 TFile f("signal.root");

 TTree *tree=(TTree*)f.Get("selectedtree");
 tree->SetDirectory(0);

 Float_t mass;       // a mass variable
 Float_t energy;        // an energy difference variable
 tree->SetBranchAddress("mes", &mass);
 tree->SetBranchAddress("de", &energy);

 TH1F *hist   = new TH1F("h1","mass_hist",10,5.2,5.29);

 for (int i=0; i<tree->GetEntries(); i++)
 {
    tree->GetEntry(i);
    if((mass > 5.2) && (mass < 5.29))
     {
       if(fabs(energy)<0.4) hist->Fill(mass);
     }
 }

 hist->SetDirectory(0);
 hist->Draw();

 f.Close();

}
```

Set branch to fill local variable

`GetEntries()`: number of entries in tree

`GetEntry(i)`: load `i`-th entry into local variables

Note: use of `int` or `Float_t`, both are okay

Summary

- ROOT is a data processing framework
- It has its in-built classes and data types. Eg: `TFile, Double_t`, etc. Reference documentation on CERN ROOT pages
- Tip: as first step, better to follow existing code and work on it..