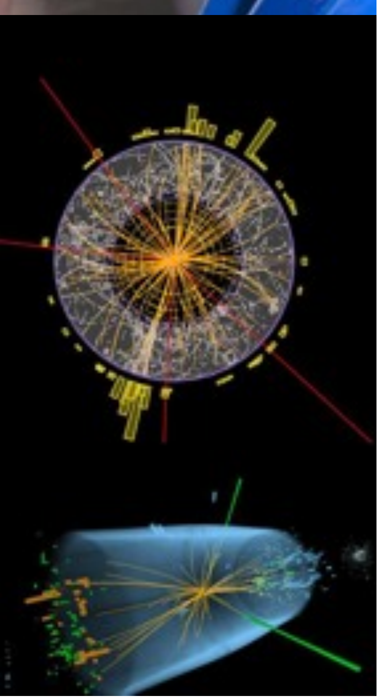
Introduction To C++: All You Need To Know To Use ROOT



ROOT Tutorial at La Plata

27-30 November 2013

Dr Lorenzo Moneta
CERN PH-SFT
CH-1211 Geneva 23
sftweb.cern.ch
root.cern.ch

C++ Introduction

- C++ is an object-oriented programming language
- C++ is one of the most complicated programming languages around
- But as well one of the most powerful ones
  - ROOT uses C++ for its purposes
  - In this presentation we focus only on the basics needed for standard data analysis and plotting

Hello World!

– Our first C++ program

```
#include <iostream>

int main() {
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

Important! Each statement has to end with a semicolon!

Hello World!

– Our second C++ program

```
#include <iostream>

// A comment line
int main() {
  double aNumber(3);
  aNumber = 10 + aNumber;
  std::cout << aNumber << std::endl;
  return 0;
}
```

Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions here

https://twiki.cern.ch/twiki/bin/view/Main/RootIRMMTutorial2013CppExercises

and solve exercises 1 and 2.

Variables and basic C++ types

- Every variable used has to be declared

```
type name(initial value);

double temperature(20.5);
```

- Many numerical built-in types available

C++ type	Meaning	Range
int	Integer	+/- 2147483648
float	Floating-point	+/- 3 * 10**38
double	Floating-point	+/- 2 * 10**308
bool	Boolean value	true,false
short	Integer	+/- 32768
long long	Integer	9*10**18
char	Character Integer	-128 to 127

Operations on variables

- Assignment

```
name = new value;

int i(1);
i = i + 1;   //now i is 2!
```

- Arithmetic operations

Operator	Meaning
-a	Sign change
a*b	Multiplication
a/b	Division
a%b	Modulus
a+b	Addition
a-b	Subtraction

Operations on variables

- Special operators

```
int i(1);
i += 3;
i *= 3;
++i;
```

- Usual operator precedence

```
a = b+2*-c + d %e;

a = (b+ (2*(-c) ) ) + (d%e);
```

Control Structures - if-then-else

- Non-trivial computations are possible as well

```
if (some condition) { what to do; }
```

```
...
double result;
a = some value;
if (a == 0) {
   std::cout << "something" << std::endl;
   result = a;
} else {
   result = 12/a;
}
...
```

- Conditions can be much more complex

```
if ( (a > 4 && b < 3) || c < 5) { ... }
```

Logical operations

- Relational (comparison) operators

Operator	Meaning
==	Equal
!=	Not equal
<	Less than
<=	Less or equal
>	Greater than
>=	Greater or equal

- Be careful! “==” and “=” are different !
- Logical operations

Operator	Meaning
!	Not
!=	Exclusive Or
&&	And
||	Or

Control Structures - loops

- Sometimes an operation needs to be repeated a certain number of times

```
...
double result(1);
for (int i = 0; i < 42; ++i)
{
  result *= i;
}
...
```

increment i after each step

repeat while this is true

what to start with

Control Structures - loops II

- Sometimes an operation needs to be repeated as long as a certain condition is true

```
...
double result = 0;
int i = 0;
while ( i < n) {
  result *= 4;
  ++i;
}
```

check first, then do

```
...
double result = 0;
int i = 0;
do {
  result *= 4;
  ++i;
} while (i < n);
```

first do, then check

Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions here

https://twiki.cern.ch/twiki/bin/view/Main/RootIRMMTutorial2013CppExercises

and solve exercises 3 and 4.

Using Functions

- Programs can be split into logical pieces
  - these are called **functions**

```
// Declaration
double calculateSquare(double input);
```

name of the function

type of the output

type of the input (arguments)

```
// Implementation / Definition
double calculateSquare(double input)
{
  return input*input;
}
```

calculate and return the output

Passing Arguments around

- Normal case in C/C++ **is passing by value**
  - Only the value of a variable is passed to a subroutine
  - For objects a **copy** is passed
  - If the subroutine changes the object, only the copy is changed
    - usually not what is intended
- To pass the variable itself, we can pass a **pointer** to the variable
  - technically, a pointer contains the address where to find the object in memory

Pointers

– A pointer points to some position in memory and keeps track of the variable type stored therein

```
int i(1);
std::cout << "Address of i: " << &i << std::endl;

// declare a pointer to an integer
int* intPointer = &i;

std::cout << "Address of i: " << intPointer << std::endl;
std::cout << "Value of i: " << *intPointer << std::endl;
```

– The following is valid C++ syntax but dangerous

```
// declare a pointer but forget to set it properly
int* intPointer;
std::cout << "Value: " << *intPointer << std::endl;
```

References

– Passing pointers works, but makes code hard to write and read

```
void sort (double* d1, double* d2) {
   if (*d2 > *d1) {
      double d = *d1;
      *d1 = *d2;
      *d2 = d;
   }
}
```

– There is usually a better choice - using references

- A reference is another name for any kind of variable

```
...
double  a = 1.1;
double  b = 2.2;
double& c = a;
a = 5;
std:: cout << c << std::endl;
```

References II

– Let's look at the sort function again

```
void sort (double& d1, double& d2) {
   if (d2 > d1) {
      double d = d1;
      d1 = d2;
      d2 = d;
   }
}
```

– Passing a reference is like passing a pointer, but:
  - you don't need to be careful on passing the arguments in
  - the code is cleaner to read
  - the reference behaves like the object itself
  - Less error-prone on initialisation

Classes and Objects

- Often several variables and several functions only make sense if used together
  - The combination of data and functions is called a **class**
  - The provided functions are called *“methods”* and the data called *“members”*
  - Each individual class is a new data type and can be used as follows:

```
Person aPerson(“name”,20);
std::cout << aPerson.getAge() << std::endl;
std::cout << aPerson.getName() << std::endl;
```

  - Two ways of creating and using an object of a certain class
    - Using a variable

```
Person aPerson(“name”,20);
aPerson.getAge();
```

    - Using a pointer

```
Person* aPersonPointer = new Person(“name”,20);
aPerson->getAge(); //short for (*aPerson).getAge()
...
delete aPersonPointer;
```

    - When creating using “new” you have as well to “delete” the object yourself!

Objects and Classes

- Class: a certain kind of object (e.g. cat)
- Object: a concrete instance of a class (like the cat of your neighbour)

- With classes we have
  - A close coupling between data and functions that work on the data
  - the possibility to hide **how** some piece of code works, we see only **what** it does
    - You want to know how to get your money from an ATM, not build one your own
    - What is made available to the user is called *“interface”*
  - the possibility to divide our code into small pieces that are individually simple and therefore easier to maintain

- Object-oriented programming is the paradigm followed in modern applications and libraries

Objects, Constructors, =

Look at this code:

```
TNamed myObject("name", "title");
TNamed mySecond;
mySecond = myObject;
cout << mySecond.GetName() << endl;
```

Look at this code:

```
TNamed myObject("name", "title");
TNamed mySecond;
mySecond = myObject;
cout << mySecond.GetName() << endl;
```

Creating objects:

1. Constructor **TNamed::TNamed(const char*, const char*)**
2. Default constructor **TNamed::TNamed()**

Look at this code:

```
TNamed myObject("name", "title");
TNamed mySecond;
mySecond = myObject;
cout << mySecond.GetName() << endl;
```

Assignment:

Look at this code:

```
TNamed myObject("name", "title");
TNamed mySecond;
mySecond = myObject;
cout << mySecond.GetName() << endl;
```

Assignment: creating a twin

Objects, Constructors, =

Look at this code:

```
TNamed myObject("name", "title");
TNamed mySecond;
mySecond = myObject;
cout << mySecond.GetName() << endl;
```

New content

```
mySecond
TNamed:
fName "name"
fTitle "title"
```

output:

```
"name"
```

Pointers

Modified code:

```
TNamed myObject("name", "title");
TNamed* pMySecond = 0;
pMySecond = &myObject;
cout << pMySecond->GetName() << endl;
```

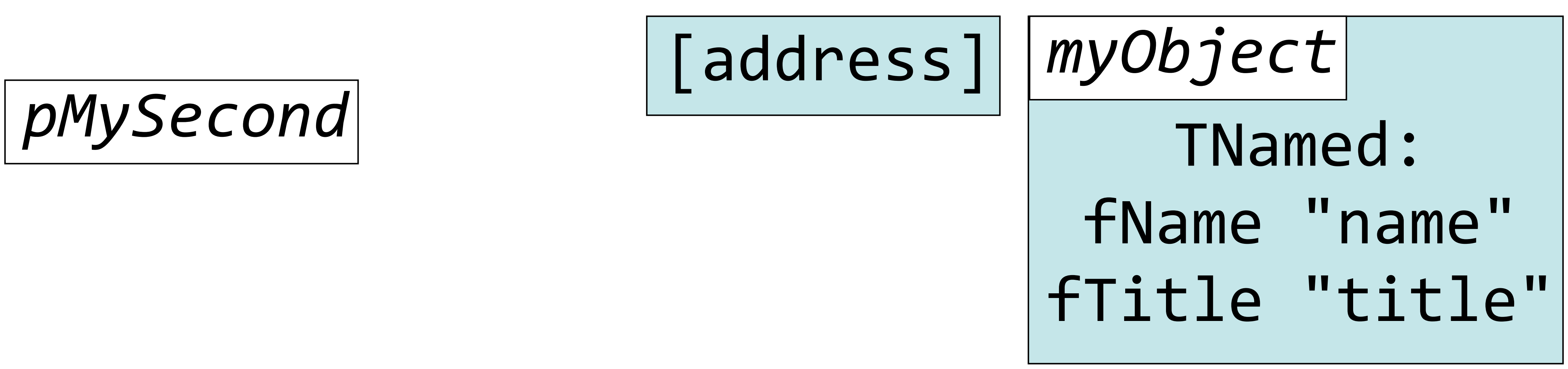
Pointer declared with "*", initialize to `0`

Pointers

Modified code:

```
TNamed myObject("name", "title");
TNamed* pMySecond = 0;
pMySecond = &myObject;
cout << pMySecond->GetName() << endl;
```

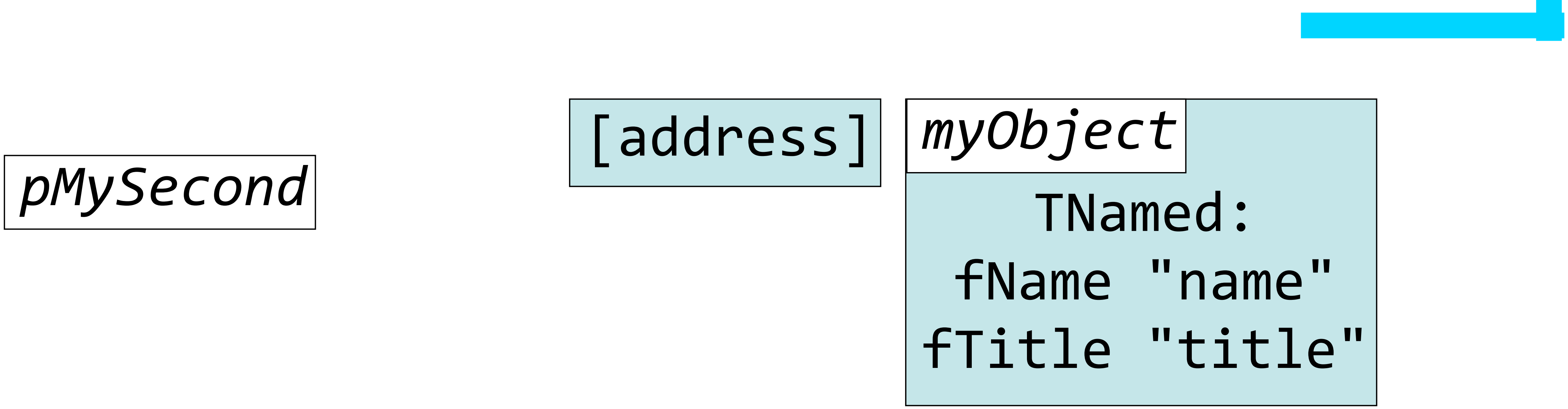
"&" gets address:



Modified code:

```
TNamed myObject("name", "title");
TNamed* pMySecond = 0;
pMySecond = &myObject;
cout << pMySecond->GetName() << endl;
```

Assignment: point to myObject; no copy



Modified code:

```
TNamed myObject("name", "title");
TNamed* pMySecond = 0;
pMySecond = &myObject;
cout << pMySecond->GetName() << endl;
```

Access members of value pointed to by "->"

Pointers

Changes propagated:

```
TNamed myObject("name", "title");
TNamed* pMySecond = 0;
pMySecond = &myObject;
pMySecond->SetName("newname");
cout << myObject.GetName() << endl;
```

Pointer forwards to object

Name of object changed – prints "newname"!

Object vs. Pointer

Compare object:

```
TNamed myObject("name", "title");
TNamed mySecond = myObject;
cout << mySecond.GetName() << endl;
```

to pointer:

```
TNamed myObject("name", "title");
TNamed* pMySecond = &myObject;
cout << pMySecond->GetName() << endl;
```

Object vs. Pointer: Parameters

Calling functions: object parameter obj gets copied for function call!

```
void funcO(TNamed obj);
TNamed myObject;
funcO(myObject);
```

Pointer parameter: only address passed, no copy

```
void funcP(TNamed* ptr);
TNamed myObject;
funcP(&myObject);
```

Object vs. Pointer: Parameters

Functions changing parameter: funcO can only access copy!

`caller` not changed!

```
void funcO(TNamed obj){
  obj.SetName("nope");
}
funcO(caller);
```

Using pointers (or references) funcP can change **`caller`**

```
void funcP(TNamed* ptr){
  ptr->SetName("yes");
}
funcP(&caller);
```

Scope

Scope: range of visibility and C++ "life".

Birth: constructor, death: destructor

```
{ // birth: TNamed() called
  TNamed n;
} // death: ~TNamed() called
```

Variables are valid / visible only in scopes:

```
int a = 42;
{ int a = 0; }
cout << a << endl;
```

Scope

Functions are scopes:

```
void func(){ TNamed obj; }

func();
cout << obj << end; // obj UNKNOWN!
```

must not return pointers to local variables!

```
TNamed* func(){
   TNamed obj;
   return &obj; // BAD!
}
```

Stack vs. Heap

So far only stack:

```
TNamed myObj("n","t");
```

Fast, but often < 10MB. Only survive in scope.

Heap: slower, GBs (RAM + swap), creation and destruction managed by user:

```
TNamed* pMyObj = new TNamed("n","t");
delete pMyObj; // or memory leak!
```

Stack vs. Heap: Functions

Can return heap objects without copying:

```
TNamed* CreateNamed(){
  // user must delete returned obj!
  TNamed* ptr = new TNamed("n","t");
  return ptr; }
```

ptr gone – but TNamed object still on the heap, address returned!

```
TNamed* pMyObj = CreateNamed();
cout << pMyObj->GetName() << endl;
delete pMyObj; // or memory leak!
```

Inheritance

Classes "of same kind" can re-use functionality

E.g. plate and bowl are both dishes:

```
class TPlate: public TDish {...};
class TBowl: public TDish {...};
```

Can implement common functions in TDish:

```
class TDish {
public:
  void Wash();
};
```

```
TPlate *a = new TPlate();
a->Wash();
```

Inheritance: The Base

Use TPlate, TBowl as dishes:
assign pointer of derived to pointer of base "every plate is a dish"

```
TDish *a = new TPlate();
TDish *b = new TBowl();
```

But not every dish is a plate, i.e. the inverse doesn't work. And a bowl is totally not a plate!

```
TPlate* p = new TDish(); // NO!
TPlate* q = new TBowl(); // NO!
```

Virtual Functions

Often derived classes behave differently:

```
class TDish { ...
  virtual bool ForSoup() const;
};
class TPlate: public TDish { ...
  bool ForSoup() const { return false; }
};
class TBowl: public TDish { ...
  bool ForSoup() const { return true; }
};
```

Pure Virtual Functions

But TDish cannot know! Mark as "not implemented"

```
class TDish { ...
  virtual bool ForSoup() const = 0;
};
```

Only for virtual functions.

Cannot create object of TDish anymore (one function is missing!)

Calling Virtual Functions

Call to virtual functions evaluated at runtime:

```
void FillWithSoup(TDish* dish) {
  if (dish->ForSoup())
    dish->SetFull();
}
```

Works for any type as expected:

```
TDish* a = new TPlate();
TDish* b = new TBowl();
FillWithSoup(a); // will not be full
FillWithSoup(b); // is now full
```

Virtual vs. Non-Virtual

So what happens if non-virtual?

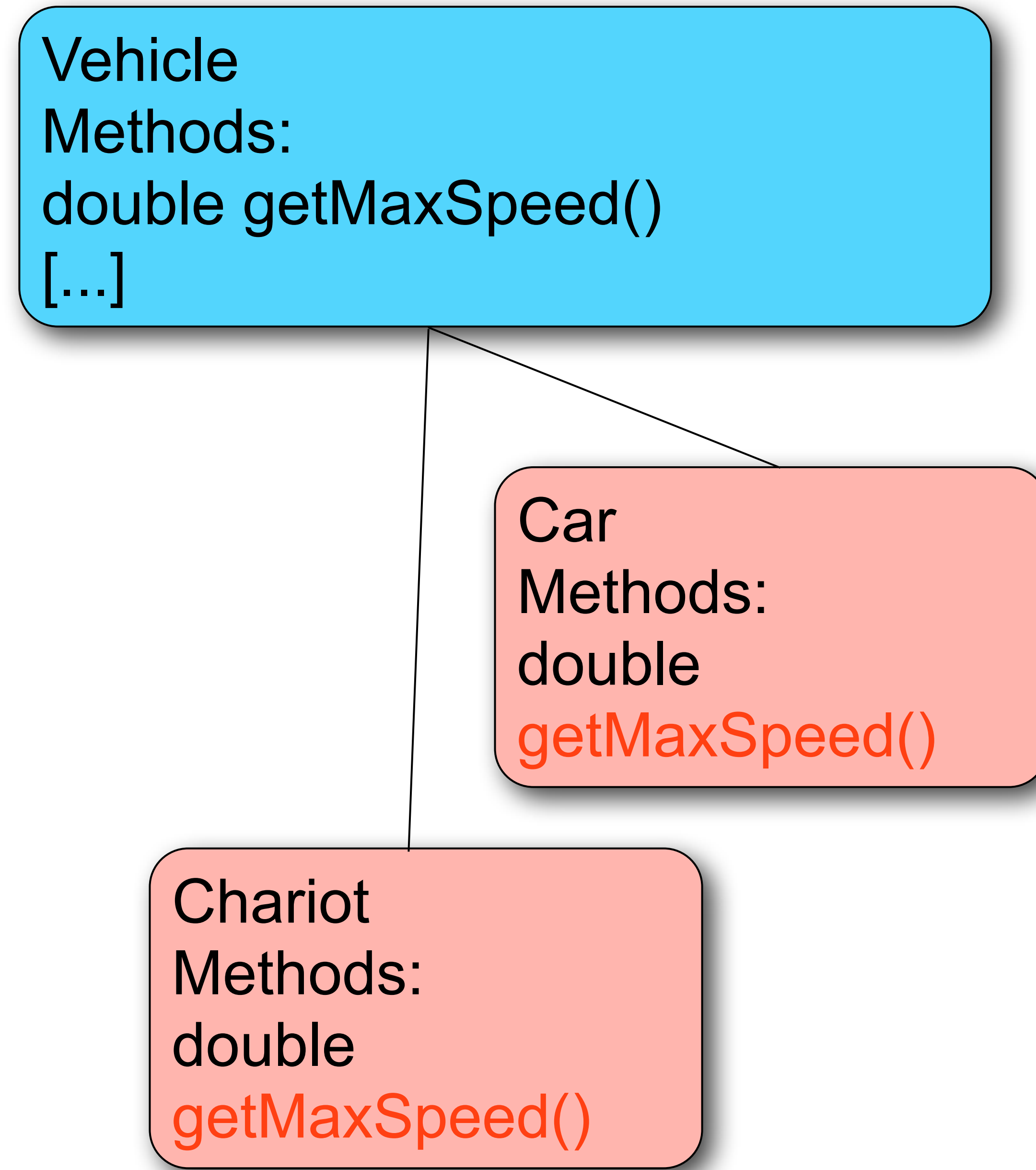
```
class TDish { ...
  bool ForSoup() const {return false;}
};
```

Will now always call TDish::ForSoup(), i.e. false

```
void FillWithSoup(TDish* dish) {
  if (dish->ForSoup())
    dish->SetFull();
}
```

A Taste of Inheritance

- Inheritance: a fundamental concept in C++
  - Used basically *everywhere*
- A derived (“daughter”) class can “inherit” methods and members from the mother class
- Suppose to have a mother **class**: **vehicle**
  - **vehicle** provides a method, double **getMaxSpeed**()
- Suppose2 derived classes: **chariot** and **car**
  - Both are vehicles. Inheritance makes sense: they share functionalities
  - It is possible to call the method getMaxSpeed() from the inherited classes as well - not always needed to re-implement it!
- Specialisation of derived classes is natural:
  - **chariot** could implement getHorsesNumber() while **car** getFuelType()

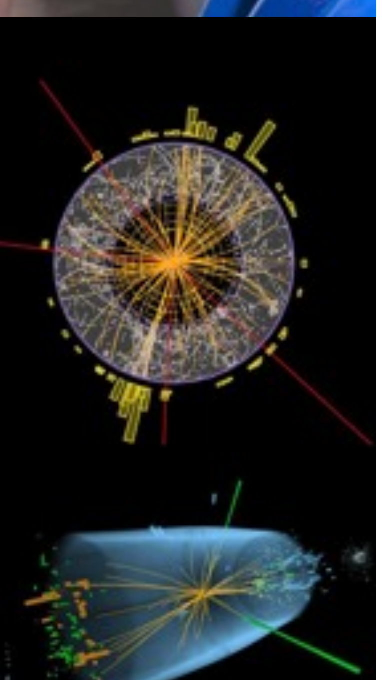


Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions here

https://twiki.cern.ch/twiki/bin/view/Main/RootIRMMTutorial2013CppExercises

and solve exercises 5 and 6.



BACKUP SLIDES

Dr Lorenzo Moneta
CERN PH-SFT
CH-1211 Geneva 23
sftweb.cern.ch
root.cern.ch

Mathematical functions

Function	Meaning
sin(x)	Sine
cos(x)	Cosine
tan(x)	Tangent
asin(x)	Arc sine
acos(x)	Arc cosine
atan(x)	Arc tangent
atan2(x,y)	Arc tangent (x/y)
exp(x)	Exponential
log(x)	Natural logarithm
log10(x)	Logarithm, base 10
abs(x)	Absolute value
sqrt(x)	Square root
pow(x,y)	x to the power of y

Type Conversions

– C++ has many pre-defined type conversions that are applied automatically, when necessary
  - integers to floating point (e.g. on addition)
  - floating point to integer
    - no rounding, but truncation of digits
  - Numbers to bool
    - 0 to false; non-zero to true
  - ...

– You can as well explicitly ask for type conversion (called cast).

Compiling C++ code using ROOT

- Command "`root-config`" tells you necessary compiler flags:

```
root-config --incdir
/Users/moneta/root/5.34.04/include
root-config --libs
-L/Users/moneta/root/5.34.04/lib -lCore -lCint -lRIO -lNet -lHist
-lGraf -lGraf3d -lGpad -lTree -lRint -lPostscript -lMatrix -
lPhysics -lMathCore -lThread -lpthread -Wl,-rpath,/Users/moneta/
root/5.34.04/lib -lm -ldl
```

- To compile a file `example.cxx` that uses root, use:

```
g++ -c -I `root-config --incdir`example.cxx
```

- To compile and link a file `example.cxx` that uses root, use:

```
g++ -I `root-config --incdir` -o example
example.cxx `root-config --libs`
```

The inverted quotes tell the shell to run a command and paste the output into the corresponding place.
On Windows, if you are using Visual Studio, the compiler is `cl` and not `g++`