Introduction to ROOT: part 1

Mirco Dorigo
mirco.dorigo@ts.infn.it

LACD 2023-2024
March 20th, 2024

Contacts

- Researcher @INFN Trieste
- mirco.dorigo@ts.infn.it
  mirco.dorigo@cern.ch
- Worked in CDF (UniTS, 2009-2013)
  and LHCb (EPFL, CERN, 2013-2020)
- In Belle II since 2020
  https://web.infn.it/Belle-II/index.php/our-research

Class plan

Wed 20/03 aula A: setup, basics commands and (very) little C++ tour

Fri 22/03 aula D: reading and storing data (histograms, tuples)

Wed 27/03 aula C edificio B: manipulating data (inspecting distributions, making selections, making graphs)

Wed 03/04 (TBD): fitting data

Fri 05/04 aula D: spare, Q&A

https://root.cern.ch

ROOT
Data Analysis Framework

About | Install | Get Started | Forum & Help | Manual | Blog Posts | Contribute | For Developers

ROOT: analyzing petabytes of data, scientifically.

An open-source data analysis framework used by high energy physics and others.

Learn more | Install v6.22/08

Get Started | **Reference** | **Forum & Help** | **Gallery**

√-1

ROOT enables *statistically sound* scientific analyses and visualization of large amounts of data: today, more than 1 exabyte (1,000,000,000 gigabyte) are stored in ROOT files. The Higgs was found with ROOT!

As *high-performance* software, ROOT is written mainly in C++. You can use it on Linux, macOS, or Windows; it works out of the box. ROOT is open source: use it freely, modify it, contribute to it!

$ _

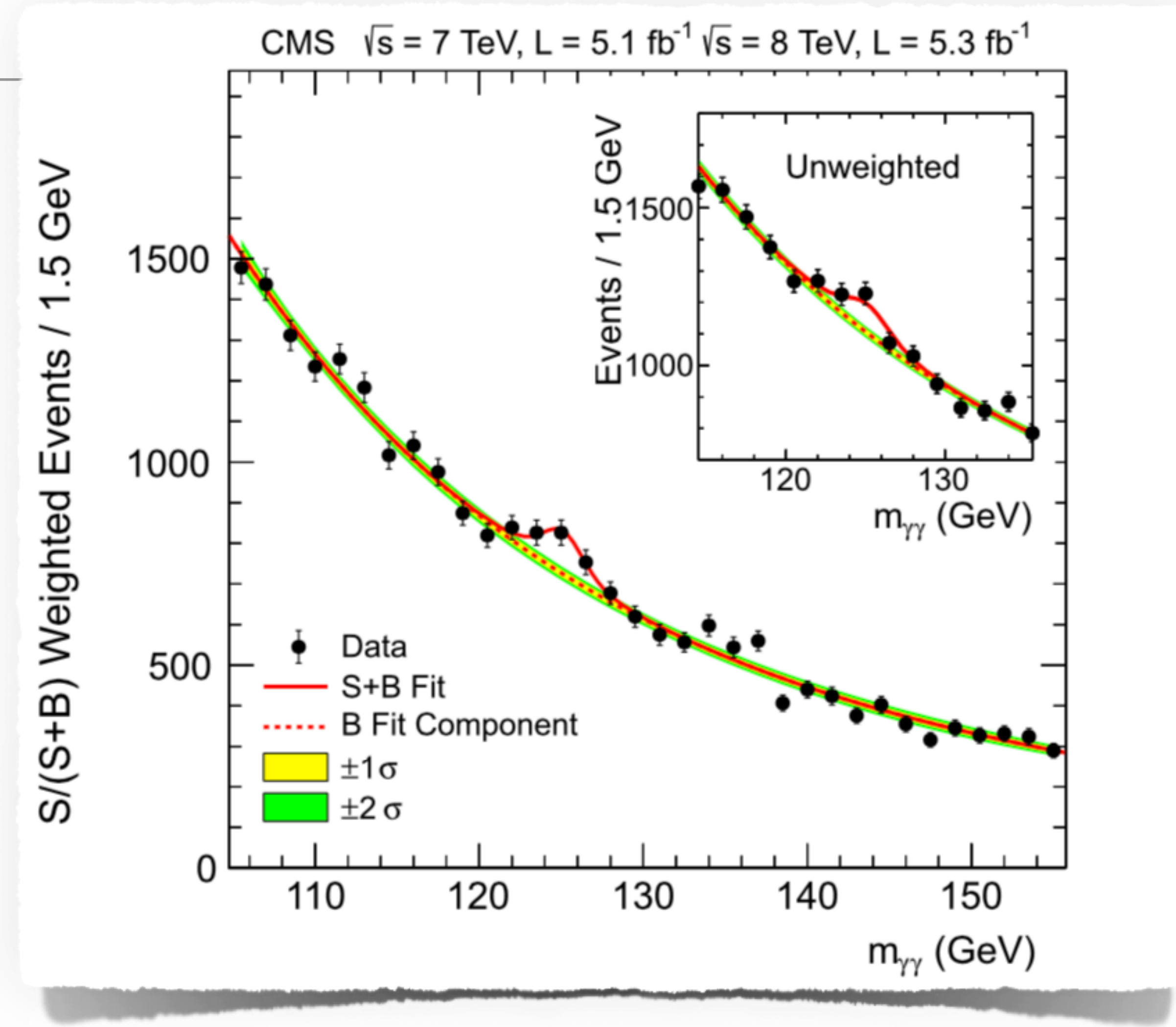
ROOT comes with an incredible C++ interpreter, ideal for *fast prototyping*. Don't like C++? ROOT integrates super-smoothly with Python thanks to its unique dynamic and powerful Python ⇄ C++ binding. Or what about using ROOT in a Jupyter notebook?

Installation

- Some instructions (a few years old, but should still work)
  https://www.unibo.it/sitoweb/gabriele.sirri2/contenuti-utili/df5f946d
  - For Windows, follow the instructions under “**run Ubuntu natively on Windows 10/11 without Virtual Machines.**”
  - For Mac: in addition to the instructions in the link, you can also use Homebrew (https://brew.sh/index_it) or MacPort (https://www.macports.org/install.php), see https://root.cern/install/#macos-package-managers
- This is the root page for installation, where you can find the link to pre-compiled binaries: https://root.cern.ch/downloading-root
- In case you need, a bash guide (get familiar with Sect. 1, 2 and 3): https://swcarpentry.github.io/shell-novice/

https://root.cern.ch

- Open-source analysis framework with building blocks for:
  - ✓ Data processing
  - ✓ Data analysis
  - ✓ Data visualisation
  - ✓ Data storage



Physics Letters B 716 (2012) 30–61

- Widely use in high-energy physics (but not only): > 1EB of data in ROOT format at CERN, thousands of plots from ROOT in papers…

- Written mainly in C++ (bindings for Python available)

`C++` and the interpreter

- `C++` is a coding language to program (writing instructions for your pc to execute).
- Here we won't learn `C++`: just very basic concepts to tell ROOT what to do.
- `C++` is a compiled language: a compiler translates ASCII files with code into machine instructions. A compiler is `gcc`.
- ROOT comes with an interpreter (CLING), don't need to compile code to run it
    - it's not a `C++` feature, its ROOT
    - CLING features just in time (JIT) compilation
    - CLNG provides an interactive `C++` shell
- Very convenient: rapid prototype/check (drawback: learn sloppy `C++`...)

Let’s start ROOT

- To start ROOT just type `root` in your shell

```
[mb-md-01:~ dorigo$ root
   ------------------------------------------------------------------
  | Welcome to ROOT 6.22/02                        https://root.cern |
  | (c) 1995-2020, The ROOT Team; conception: R. Brun, F. Rademakers |
  | Built for macosx64 on Aug 17 2020, 12:46:52                      |
  | From tags/v6-22-02@v6-22-02                                      |
  | Try '.help', '.demo', '.license', '.credits', '.quit'/'.q'       |
   ------------------------------------------------------------------

root [0] 
```

- `.q` to quit ROOT
- `.?` to obtain a list of command
- .!<command> (e.g. `.!pwd`) to access shell command
- Can start ROOT also with flags (eg. `root -l`).
  - `-l` (do not show the root banner)
  - `-b` (batch mode, no graphics)
  - `-q` (run and quit)
- A few examples below, try `man root` for full list.

Using the prompt

- As a simple calculator

```
mb-md-01:~ dorigo$ root -l
root [0] 2*3 + 10 - 36
(int) -20
root [1] 2*3.
(double) 6.0000000
root [2] pow(2,8)
(double) 256.00000
root [3] sqrt(144)
(double) 12.000000
```

- Accessing complex functions (via `TMath` library)

```
root [10] TMath::Gaus(2)
(double) 0.13533528
root [11] exp(-0.5*2*2)
(double) 0.13533528
root [12]
```

- Can run also `C++` instructions

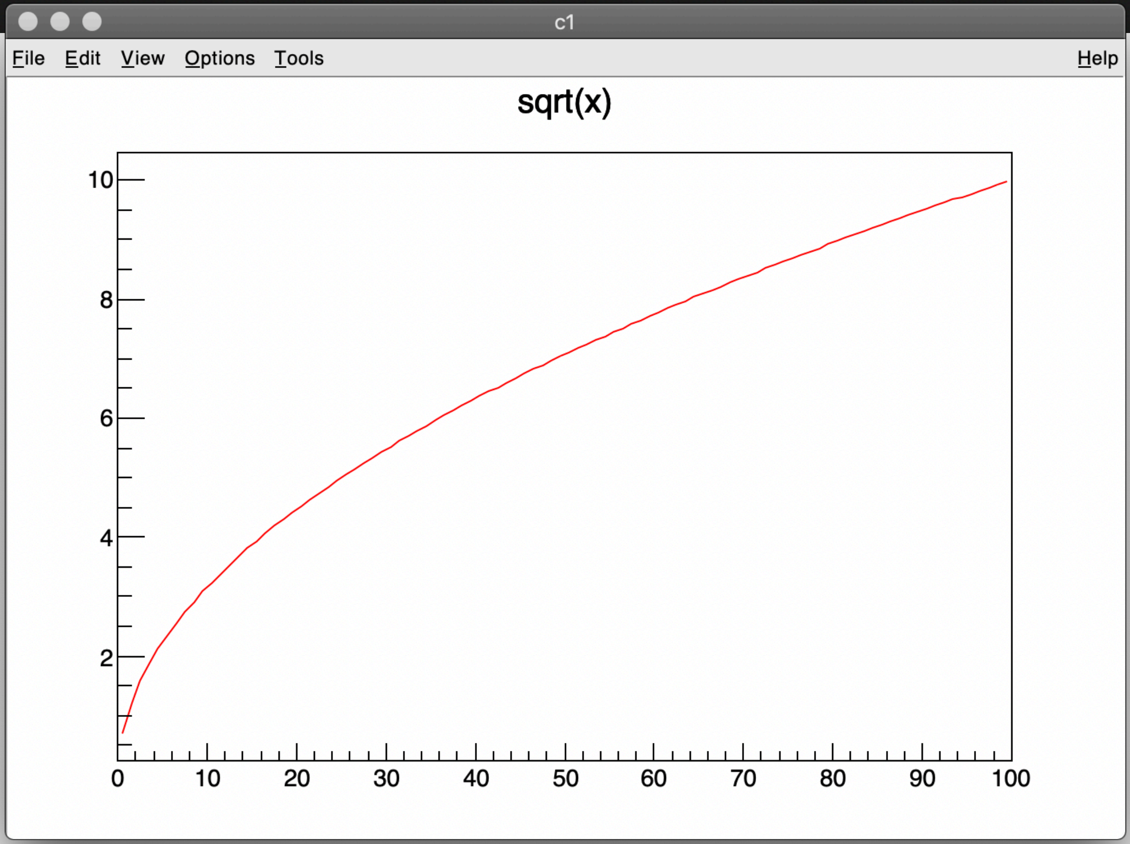
```
mb-md-01:~ dorigo$ rootl
root [0] double x = 0.127;
root [1] int N = 20;
root [2] double g_series = 0;
root [3] for(int i=0; i< N ; ++i) g_series += pow(x,i);
root [4] cout << "Value after 20 iterations: " << g_series << endl;
Value after 20 iterations: 1.14548
root [5] fabs(g_series - (1./(1.-x)))
(double) 0.0000000
```

$$
\begin{aligned}
\frac{1}{1-x} &= 1 + x + x^2 + x^3 + x^4 + \dots \\
&= \sum_{n=0}^{\infty} x^n
\end{aligned}
$$

Using the prompt

- To access ROOT classes

```
mb-md-01:~ dorigo$ root -l
root [0] TF1 f_sqrt("f","sqrt(x)",0,100);
root [1] f_sqrt.Eval(9)
(double) 3.0000000
root [2] f_sqrt.Eval(65.7)
(double) 8.1055537
root [3] f_sqrt.Derivative(9.)
(double) 0.16666667
root [4] f_sqrt.Integral(4,16)
(double) 37.333333
root [5] f_sqrt.Draw()
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
root [6]
```



- Draw the function `1/(1-x)`

Running a macro

- The prompt is powerful, but not convenient to (re)run several lines of code. Let's put them in a "macro", a bunch of lines of code in a ASCII file.

```
void myMacro(){

 //several lines of codes here

 return;
}
```

- Go back and put in a macro the example of the geometrical series.

- Notice: the name of the macro must be the same of the function

- To run your macro, type `root -l myMacro.C`, or

```
mb-md-01:~ dorigo$ root -l
root [0] .x myMacro.C
Value after 20 iterations: 1.145475
root [1]
```

Compiling a macro

- Not only JIT compilation, ACLIC can make libraries from your code
- Just load the macro adding a '+' at the end: `.L myMacro.C+`

```
root [0] .L myMacro.C+
Info in <TMacOSXSystem::ACLiC>: creating shared library /Users/dorigo/./myMacro_C.so
In file included from input_line_12:6:
./myMacro.C:7:44: error: use of undeclared identifier 'pow'
 for(int i=0; i<iterations; ++i) result += pow(variable,i);
                                           ^
./myMacro.C:16:2: error: use of undeclared identifier 'cout'
 cout << "Value after " << N << " iterations: " << g_series(x,N) << endl;
 ^
./myMacro.C:16:69: error: use of undeclared identifier 'endl'
 cout << "Value after " << N << " iterations: " << g_series(x,N) << endl;
```

- What's the problem?

Need to be C++ compliant

- Add some “headers”; make explicit the use of std (standard) library

```
#include <math.h>
#include <iostream>

void myMacro(){

 double x = 0.127;
 int N = 20;
 double g_series = 0;
 for(int i=0; i< N ; ++i) g_series += pow(x,i);

 std::cout << "Value after " << N << " iterations: " << g_series << std::endl;

 return;
}
```

- Should be OK now

```
[mb-md-01:~ dorigo$ nano myMacro.C
[mb-md-01:~ dorigo$ root -l
[root [0] .L myMacro.C++
Info in <TMacOSXSystem::ACLiC>: creating shared library /Users/dorigo/./myMacro_C.so
[root [1] myMacro()
Value after 20 iterations: 1.145475
root [2]
```

Going full C++

- ROOT libraries can be used to produce standalone compiled applications. Need to make our macro `C++` standard code, by adding the `main` function

```
#include <math.h>
#include <iostream>

void myMacro(){

 double x = 0.127;
 int N = 20;
 double g_series = 0;
 for(int i=0; i<N ; ++i) g_series += pow(x,i);

 std::cout << "Value after " << N << " iterations: " << g_series << std::endl;

 return;
}

int main(){
 myMacro();
 return 0;
}
```

- Compile and run the binary `example`.

```
mb-md-01:~ dorigo$ gcc -o example myMacro.C
mb-md-01:~ dorigo$ ./example
Value after 20 iterations: 1.145475
mb-md-01:~ dorigo$
```

Note: not using ROOT libraries here, otherwise: `` g++ -o example myMacro.C `root-config --cflags --libs` ``

Language considerations

- Our code will be simple macros that can run on-the-fly, without compilation. We can afford being sloppy with the language…
- Anyway, a minimum knowledge of `C++` basics is needed.
- Will have a look but you will mostly learn by copying examples. If you are completely unfamiliar, there are many good tutorials and guides on the web (e.g. http://www.cplusplus.com).
- Let's do a quick tour

Fundamental types

```
#include <math.h>
#include <iostream>

void myMacro(){

 double x = 0.127;
 int N = 20;
 double g_series = 0;
 for(int i=0; i< N ; ++i) g_series += pow(x,i);

 std::cout << "Value after " << N << " iterations: " << g_series << std::endl;

 return;
}
```

Variable declaration:
every name and every expression has a type that determines the operations that may be performed on it.

C++ Fundamental Types		Machine Independent Types	
C++ type	Size (bytes)	ROOT types	Size (bytes)
(unsigned)char	1	(U)Char_t	1
(unsigned)short	2	(U)Short_t	2
(unsigned)int	2 or 4	(U)Int_t	4
(unsigned)long	4 or 8	(U)Long_t	8
float	4	Float_t	4
double	8 (>=4)	Double_t	8
long double	16 (>=double)		

Operators

```
#include <math.h>
#include <iostream>

void myMacro(){

 double x = 0.127;
 int N = 20;
 double g_series = 0;
 for(int i=0; i< N; ++i) g_series += pow(x,i);

 std::cout << "Value after " << N << " iterations: " << g_series << std::endl;

 return;
}
```

Make actions on the variables, functions, output..

(Some) operators

Arithmetic operators

C++	Purpose
x++	Postincrement
++x	Preincrement
x--	Postdecrement
--x	Predecrement
+x	Unary plus
-x	Unary minus
x*y	Multiply
x/y	Divide
x%y	Modulus
x+y	Add
x-y	Subtract
Pow(x,y) or TMath::Power(x,y)	Exp
x = y	Assignment
X += y	Updating assignment
X -=, *=, /=, %=, …, Y	

Logic/comparison operators

C++	ROOT extension
false or 0	kFALSE
true or nonzero	kTRUE
!x	
x && y	
x || y	
x < y	
x <= y	
x > y	
x >= y	
x == y	
x != y	

Loops et al. (statements)

```
#include <math.h>
#include <iostream>

void myMacro(){

 double x = 0.127;
 int N = 20;
 double g_series = 0;
 for(int i=0; i< N ; ++i) g_series += pow(x,i);

 std::cout << "Value after " << N << " iterations: " << g_series << std::endl;

 return;
}
```

Repeat the instructions N times

- There are other types of loops (eg. `while`).
  They can be combined with other kind of statement, like `if`, `if … else …`, `switch …` and so on
- We will see them with the examples throughout the lessons.

Functions

- Very convenient to write functions in our macros

```
#include <math.h>
#include <iostream>

double g_series(double variable, int iterations){

 double result=0;
 for(int i=0; i<iterations; ++i) result += pow(variable,i);
 return result;
}

void myMacro(){

 double x = 0.127;
 int N = 20;
 std::cout << "Value after " << N << " iterations: " << g_series(x,N) << std::endl;

 return;
}
```

- Notice: `myMacro()` was used as a function in `main` in slide 9.

Functions — overloading

- Parameters are important. Can overload functions.

```
#include <math.h>
#include <iostream>

double g_series(double variable){

 double result=0;
 for(int i=0; i<3; ++i) result += pow(variable,i);
 return result;
}

double g_series(double variable, int iterations){

 double result=0;
 for(int i=0; i<iterations; ++i) result += pow(variable,i);
 return result;
}

void myMacro(){

 double x = 0.127;
 int N = 20;
 std::cout << "Value after " << N << " iterations: " << g_series(x,N) << std::endl;
 std::cout << "Value after 3 fixed iterations: " << g_series(x) << std::endl;

 return;
}
```

Functions — overloading

- Parameters are important. Can overload functions.

```
#include <math.h>
#include <iostream>

double g_series(double variable){

 double result=0;
 for(int i=0; i<3; ++i) result += pow(variable,i);
 return result;
}

double g_series(double variable, int itera

 double result=0;
 for(int i=0; i<iterations; ++i) result +=
 return result;
}

void myMacro(){

 double x = 0.127;
 int N = 20;
 std::cout << "Value after " << N << " iterations: " << g_series(x,N) << std::endl;
 std::cout << "Value after 3 fixed iterations: " << g_series(x) << std::endl;

 return;
}
```

```
mb-md-01:~ dorigo$ root -l myMacro.C
root [0]
Processing myMacro.C...
Value after 20 iterations: 1.14548
Value after 3 fixed iterations: 1.14313
root [1] .q
```

Defining new types

- The first step to define new types is to create a structures to group elements (members)

```cpp
#include <iostream>

struct ComplexNumber{

 double re;
 double im;

};


void macro(){

 ComplexNumber z;
 z.re = 1.;
 z.im = 3 ;

 std::cout << "real part: " << z.re << endl;
 std::cout << "imaginay part: " << z.im << endl;

}
```

A structure to define a new type, complex numbers

An object of the new type.
Access the members `re` and `im` using a dot.

Defining new types

- The first step to define new types is to create a structures to group elements (members)

```
#include <iostream>

struct ComplexNumber{

 double re;
 double im;

};


void macro(){

 ComplexNumber z;
 z.re = 1.;
 z.im = 3 ;

 std::cout << "real part: " << z.re << endl;
 std::cout << "imaginay part: " << z.im << endl;

}
```

```
root [0] .x macro.C
real part of z 1
imaginay part of z 3
```

Classes

- Classes are structures on steroids: add functionalities (methods)

```cpp
#include <iostream>

class ComplexNumber{

protected:
 double re;
 double im;

public:
 ComplexNumber(double x, double y) { re = x; im= y; }

 double GetRe(){ return re; }
 double GetIm(){ return im; }

 void cPrint(){
   std::cout << "Re: " << re << " " << "Im: " << im << std::endl;
 }

  //can continue...
  //for instance, define sum, product, ...

};

void macro(){

 ComplexNumber z(3,4);
 std::cout << "real part of z " << z.GetRe() << std::endl;
 std::cout << "imaginay part of z " << z.GetIm() << std::endl;

 z.cPrint();

}
```

class “constructor”

Can define all operations that you want with the members of the class

Initialise an object

Access the methods with the dot.

Classes

- Classes are structures on steroids: add functionalities (methods)

```
#include <iostream>

class ComplexNumber{

protected:
 double re;
 double im;

public:
 ComplexNumber(double x, double y) { re = x; im= y; }

 double GetRe(){ return re; }
 double GetIm(){ return im; }

 void cPrint(){
   std::cout << "Re: " << re << " " << "Im: " << im << std::endl;
 }

  //can continue...
  //for instance, define sum, product, ...

};

void macro(){

 ComplexNumber z(3,4);
 std::cout << "real part of z " << z.GetRe() << std::endl;
 std::cout << "imaginay part of z " << z.GetIm() << std::endl;

 z.cPrint();

}
```

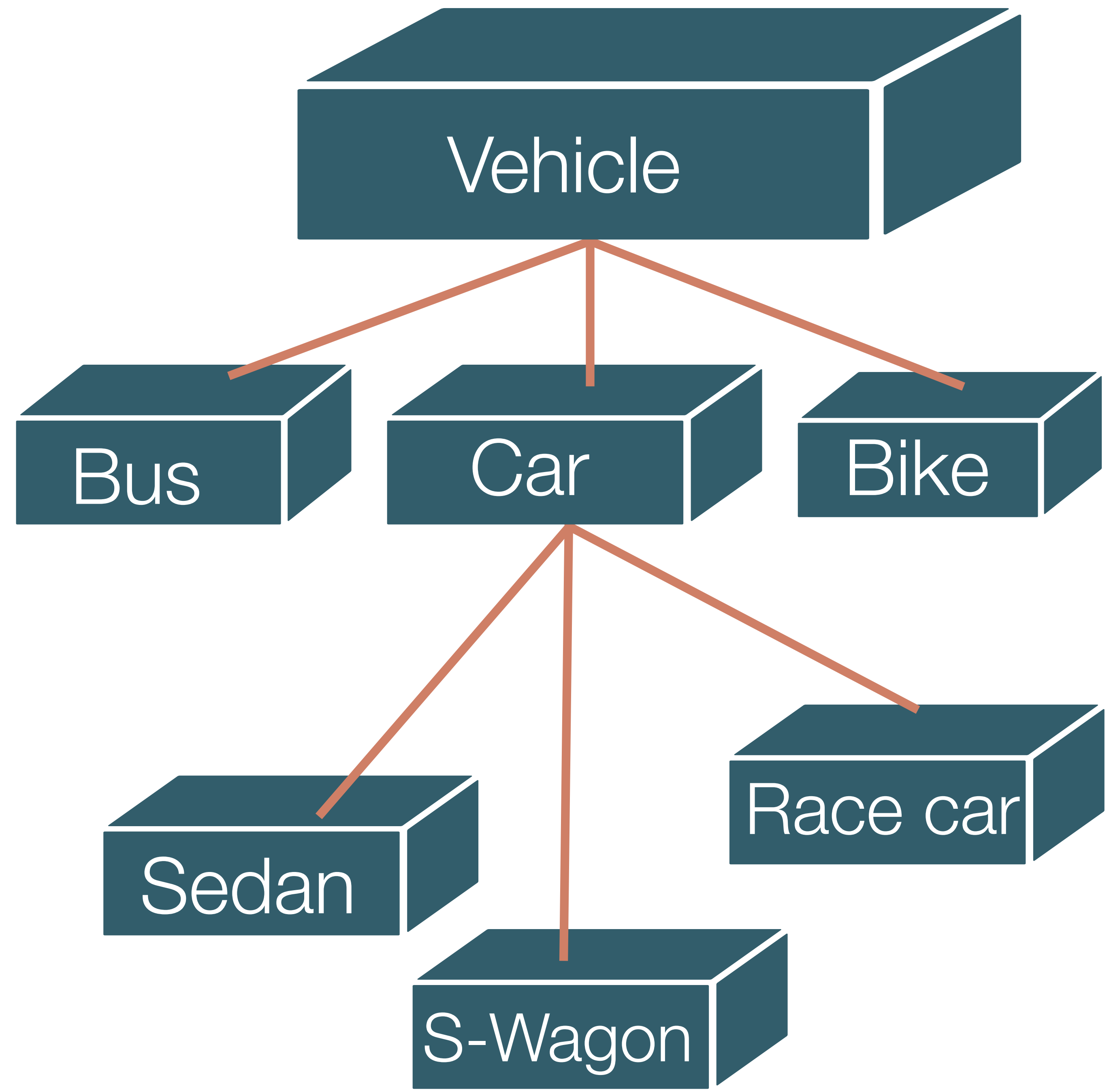
```
root [0] .x macro.C
real part of z 3
imaginary part of z 4
Re: 3 Im: 4
root [1]
```

Object oriented

- Classes have members (variables) and methods (functions)

- An instance of a class is an object, created by a special method, the constructor (can be overloaded).

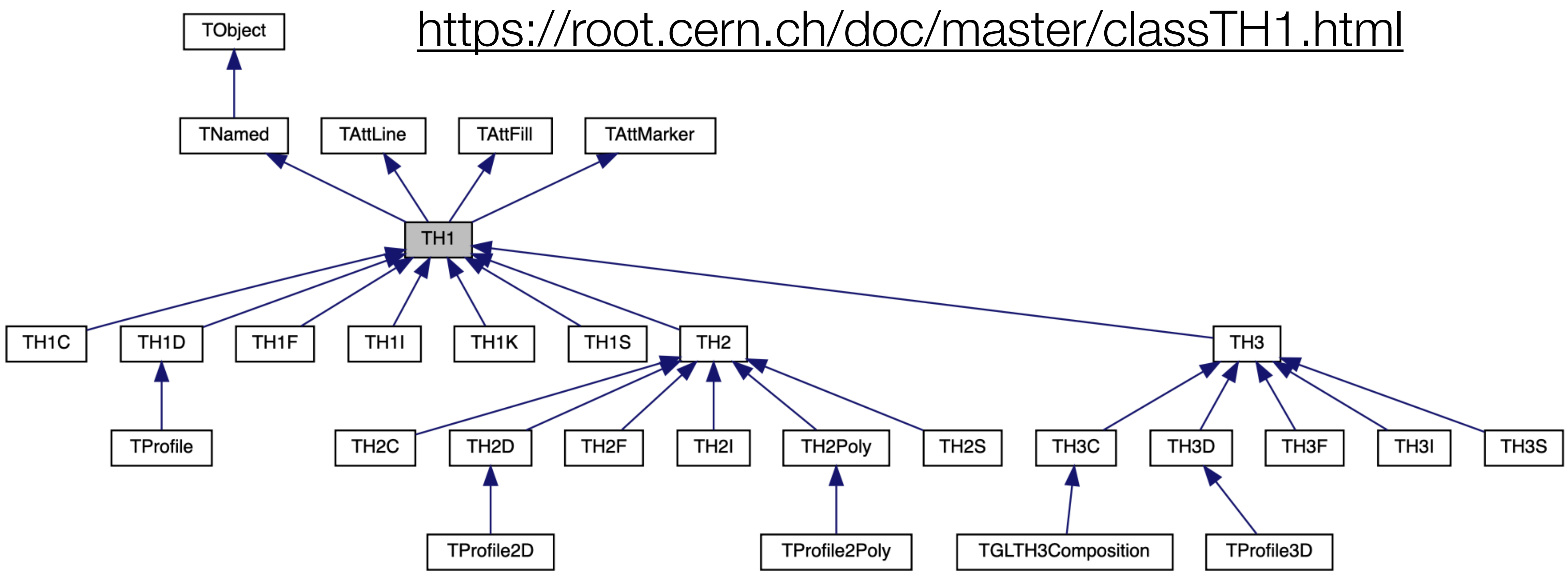
- We can define very abstract classes, and then add derived classes that inherit from them to go more specific with what we need to do.



Going back to ROOT

- ROOT is organised in classes: you will use objects and methods
- All classes begin with a “`T`” in ROOT (`TGraph,TH1,TF1`...)
- All methods begin with a capital letter (`Draw(),GetX(),Derive()`...)
- Classes inherited from more general (abstract) classes

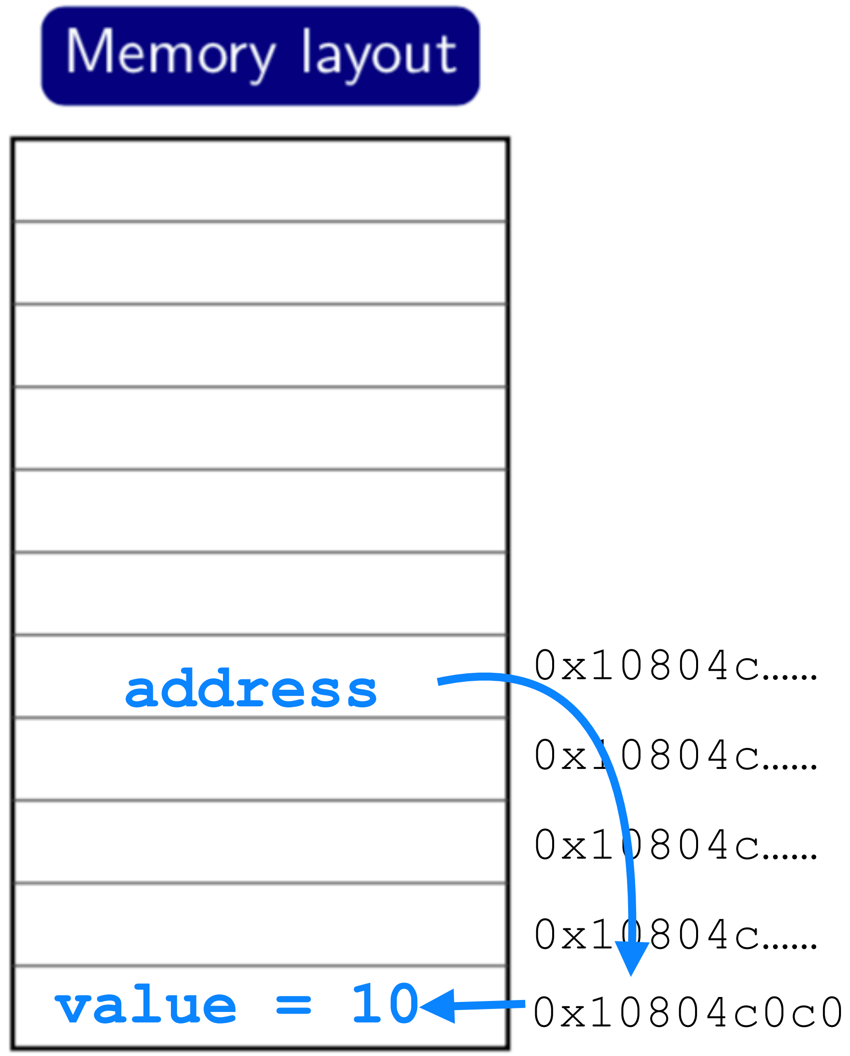
https://root.cern.ch/doc/master/classTH1.html



Pointers

- Values are in memory, at a location (an address).

```
root [0] double value = 10.;
root [1] double* address = &value;
root [2] cout << value << endl;
10
root [3] cout << &value << endl;
0x10804c0c0
root [4] cout << address << endl;
0x10804c0c0
root [5] cout << *address << endl;
10
root [6]
```



- `&` takes the address of `value`
- `address` now contains the memory-address of `value`
- `*address` accesses the content

Pointers and objects

- Can use pointers with objects: create with `new`

```
mb-md-01:~ dorigo$ root -l
root [0] .L macro.C
root [1] ComplexNumber z(1,2);
root [2] z.cPrint();
Re: 1 Im: 2
root [3] w = new ComplexNumber(3,2);
root [4] w.cPrint();
ROOT_prompt_4:1:2: error: member reference type 'ComplexNumber *' is a pointer;
did you mean to use '->'?
w.cPrint();
~^
 ->
root [5] w->cPrint();
Re: 3 Im: 2
```

normal object

w is a pointer to an object

Methods cannot be called by '.'

Use '->', which is a shorthand for '`(*w).cPrint()`'

- Make explicit in code:
  `ComplexNumber* w = new ComplexNumber(3,2);`

- Should need also a destructor to `delete`, but for simple classes like that the compiler takes care for us (important when you have pointers in the class, to free allocated memory).

Scope

- Every variables has a lifetime. It is defined only within a scope.
- It is determined by the { … }

```
#include <math.h>
#include <iostream>

void myMacro(){

 double x = 0.127;
 int N = 20;
 double g_series = 0;
 for(int i=0; i< N ; ++i) g_series += pow(x,i);

 std::cout << "Value after " << N << " iterations: " << g_series << std::endl;

 return;
}
```

`C++` overview wrap-up

- Done a very quick (and incomplete) tour of `C++`.
  This is *NOT* sufficient `C++` for real-life.

- Sufficient to follow the course. We will do very simple coding (might not be really `C++` kosher…).

- Important to understand basic concepts, such that you are not lost when navigating the ROOT class reference
  (eg. https://root.cern.ch/doc/master/classTH1.html)

- Writing macros will come with examples…

Some exercises

- Start ROOT. From the prompt look at the content of your folder, and look at the content of the folder above.

- Write a macro to compute the integral of x^2 between -1 and 1.
  Don't use `TF1`, but compare your results with that of `TF1`.

- Compile the macro in ROOT (.L macro.C+) and run it.

- Explore the `TF1` class. Look at the type 2, expression using variable x with parameters. Using this, write a normal Gaussian function in the range -5 and 5, set the mean to 0 and the std deviation to 1, and draw it.
  Get the value of the 2nd derivative at x = 0. Put all in a macro and run it.

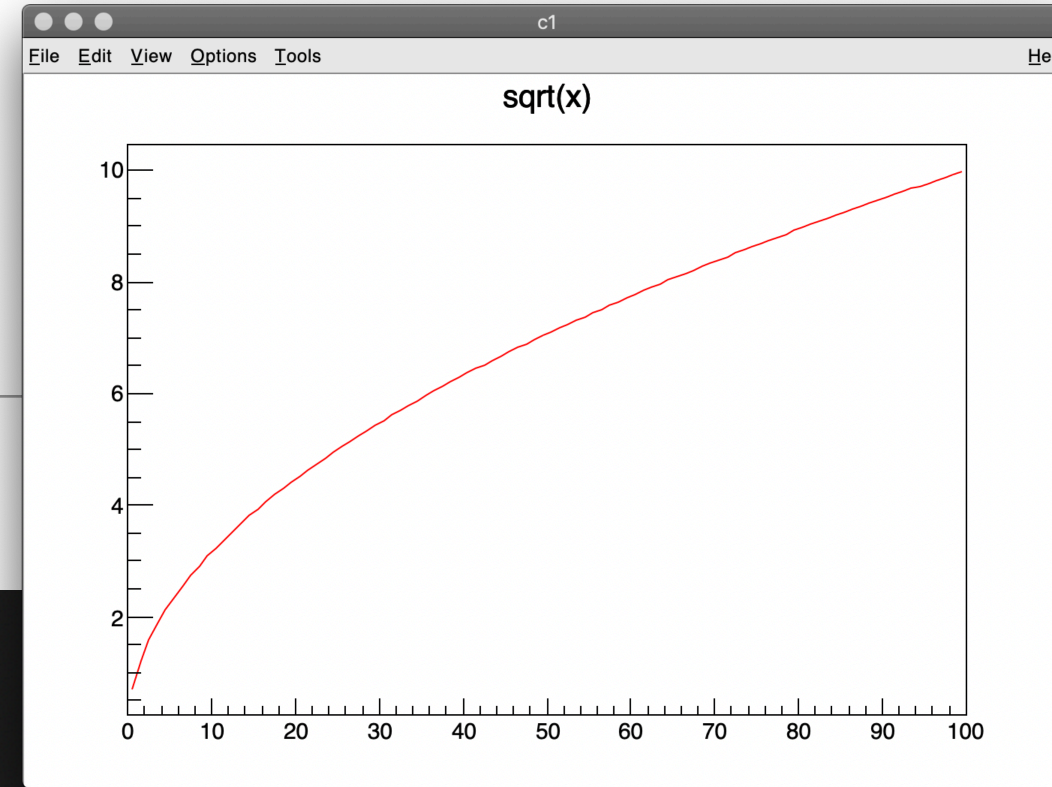
- From the ROOT prompt: draw the `Landau` function.

Extra

Going back to ROOT

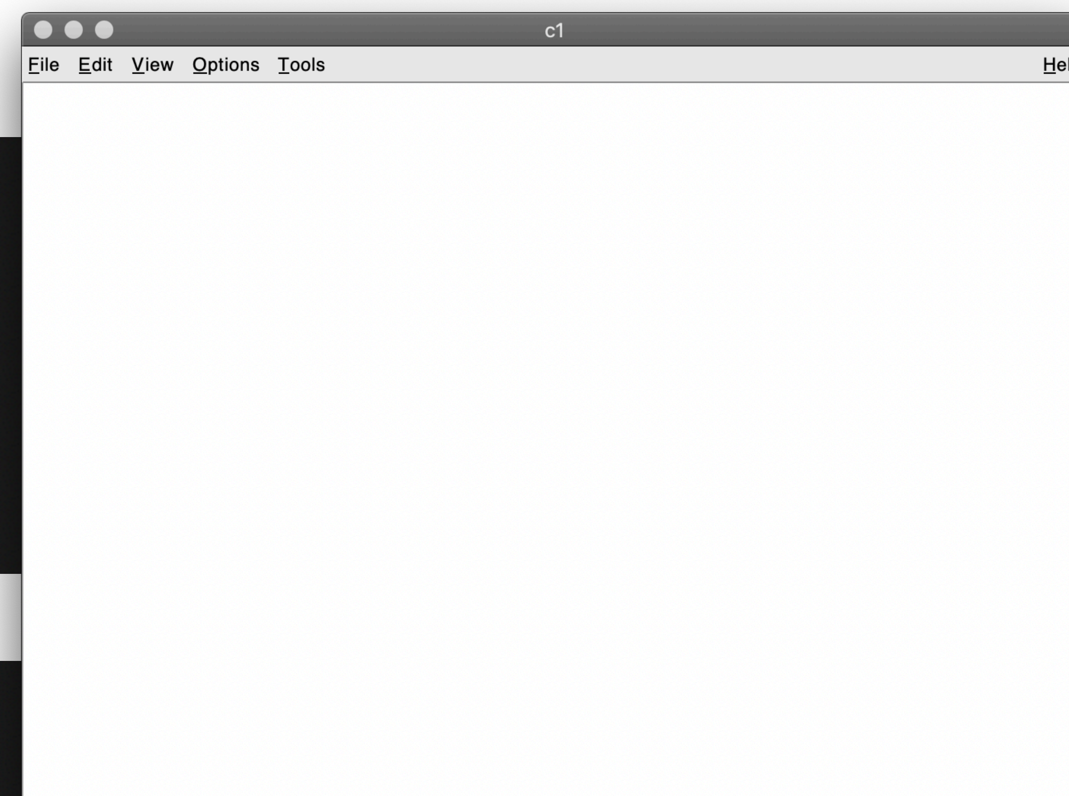
- Remember this?

```
root [0] TF1 f_sqrt("f","sqrt(x)",0,100);
root [1] f_sqrt.Eval(9)
(double) 3.0000000
root [2] f_sqrt.Draw()
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
```



- Put it on a macro and run it.

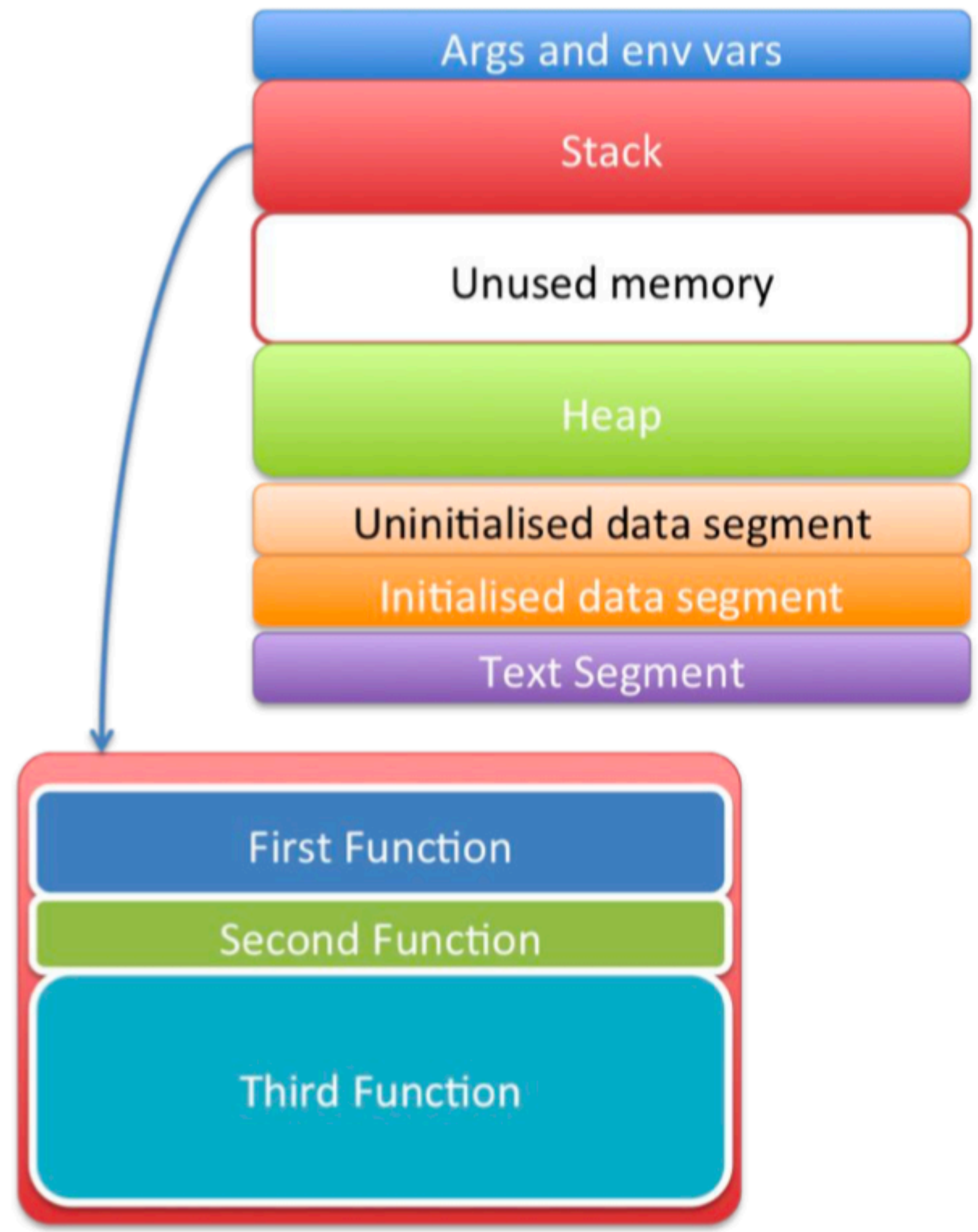
```
void func(){

 TF1 f_sqrt("f","sqrt(x)",0,100);
 cout << f_sqrt.Eval(9) << endl;
 f_sqrt.Draw();

}
```




```
root [0]
Processing func.C...
3
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
root [1]
```

Stack and heap

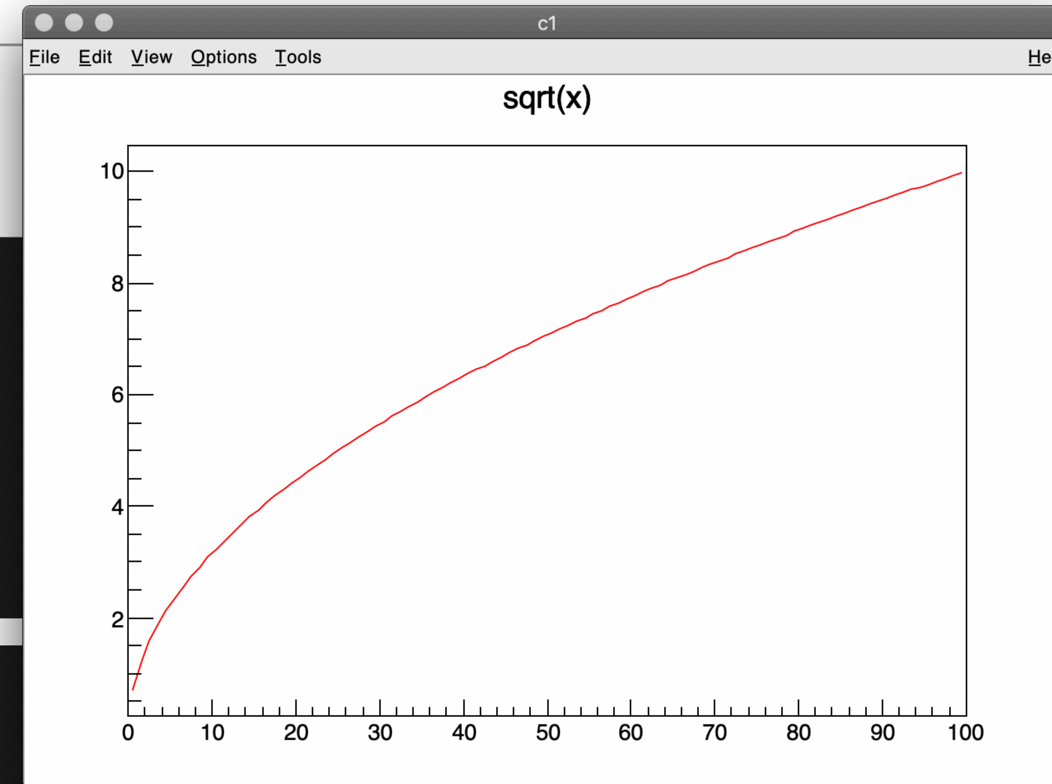
- Text segment: code to be executed
- Initialised data segment: initialised global variable
- Uninitialised data segment: contains uninitialised global variables
- The stack: contains the frames, collections of all data associated with one subprogram call (one function)
- The heap: dynamic memory, requested with “`new`”



Stack and heap

- Let’s try with pointers

```
void func(){

 TF1* f_sqrt = new TF1("f","sqrt(x)",0,100);
 cout << f_sqrt->Eval(9) << endl;
 f_sqrt->Draw();

}
```




```
root [0]
Processing func.C...
3
Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
root [1]
```

- Without the pointer, the function `func()` is in the stack, and its scope ends after closing the last “`}`”. The program, made just by this function, ends and all variables inside the function are lost.

- “`new`” puts the object on the heap, escapes scope and the object survives.