C/C++
ROOT

Matteo Duranti, Valerio Vagelli

matteo.duranti@infn.it

(cfr. https://www.tutorialspoint.com/cprogramming/c_file_io.htm
https://www.tutorialspoint.com/c_standard_library/c_function_feof.htm
http://www.cplusplus.com/doc/tutorial/files/
https://www.tutorialspoint.com/cprogramming/c_preprocessors.htm
http://www.cplusplus.com/articles/2LywvCM9/
https://root.cern.ch/root/html534
http://webhome.phy.duke.edu/~raw22/public/root.tutorial/basic_root_20100701.pdf)

C++

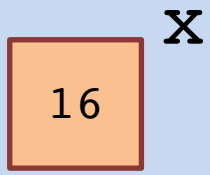
- Programming language
  - Human “friendly” instructions (top level) that are translated to “machine” instructions by the compiler
  - Strict rules
  - Object Oriented
    - Structures interacting among themselves through methods

C++ Variables

- A ***variable*** is a named location in the memory
  - The name defines uniquely the variable in the scope { ... }
  - After the definition, a memory block is assigned. The compiler knows the bytes to allocate and which operations are legal by the type of the variable
  - After the initialization, the variable gets a starting value.

    ⓘ Try to always initialize variables! Many runtime errors are generated by ill-initialized variables

```
#include <iostream>   //# defines a preprocessor instruction
using namespace std;  //:: namespace resolution operator

int main() {    // { defines a scope
  int x;          //variable with type int is defined
  x=7+9;          //variable x is initialized to some value
  double d = 5.6; //variable d with type double is defined and initialized
  cout<<x<<" "<<d<<endl;
  cout<<"Size of x: "<<size_of(x)<<endl;
  cout<<"Size of d: "<<size_of(d)<<endl;
  return 0;       //return value of the function main
}
```




```
16 5.6
4
8
```

Algorithm Flux

- The flux of information can be controlled by several construct

- **IF construct**
  - decides which scope to solve, controlled by a boolean (true/false) value

```
if( this_expression == true )
  { do_this(); }
else
  { do_that(); }
```

- Shortuct: if/else assignment in a line

```
int value=0;
if( this_expression == true )
  { value = this(); }
else
  { value = that(); }
```

```
int value = this_expression ? this() : that();
```

Algorithm Flux

- The flux of information can be controlled by several construct
- **FOR construct**
  - Iterates the instructions in the scope for a fixed amount of times

```
for( int i=0; i<100; i++)
  { do_this();
    do_that();
  }
```

  - A FOR loop can be solved (exit the loop and continue with the code) with a `break` call

```
for( int i=0; i<100; i++)
  { bool I_am_bored = check_if_bored();
    if( I_am_bored ) break;
  }
```

  - A `continue` call allows to skip to the next iteration without solving the complete scope

```
for( int i=0; i<100; i++)
  { bool I_like_this = check_if_good (i);
    if( !I_like_this ) { continue; }
    else { do_stuff_on_this(); }
  }
```

Algorithm Flux

- The flux of information can be controlled by several construct
- **WHILE construct**
  - Iterates the instructions in the scope while the heading expression is true

```
while( head_expr==true )
{
  do_this();
  if( time_to_stop() ) head_expr=false;
}
```

  - `break` calls and `continue` calls work as for the FOR loop case

Arrays

- An ***array*** is a group of elements with the same type indexed by the same variable

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

int main() {
  int xx[5] = {1,-3,1,0,4};
  int yy[5];
  for(int i=0; i<5; i++)
  {  yy[i] = xx[i]-xx[0]; }

  float mm[4][7];
  for(int ii=0; ii<4; ii++)
    for(int jj=0; jj<7; jj++)
    {  mm[ii][jj] = some_value(ii,jj); }

  return 0;
}
```

- An array is NOT a dynamic structure. If you want an array with a dynamic size, you should use a `std::vector`

  `http://www.cplusplus.com/reference/vector/vector/`

Functions

- A **function** is a segment of code (i.e. a set of instructions) that perform a single task
- Functions can be called during the execution of the programs.
- Functions may return a single value, a complex object, or nothing.
- Use functions! Make your code **modular**. It helps when things don't work and you have to debug your code.

```
type fun_name (type par1, type par2, … type parn){
//instructions here;
}

int magnitude( float number ){
  if( number<=0 ) return 0;
  else return (int)log10(number);
}

void even( int number ){
  printf("%d is %s\n", number, number%2==0 ? "even" : "odd" );
  return;
}

int main() {
  int x=102;
  even(x);
  printf("magnitude is %d\n", magnitude(x) );
```

```
102 is even
magnitude is 2
```

Functions

- A **function** is a segment of code (i.e. a set of instructions) that perform a single task
- Functions can be called during the execution of the programs.
- Functions may return a single value, a complex object, or nothing.
- Use functions! Make your code **modular**. It helps when things don't work and you have to debug your code.

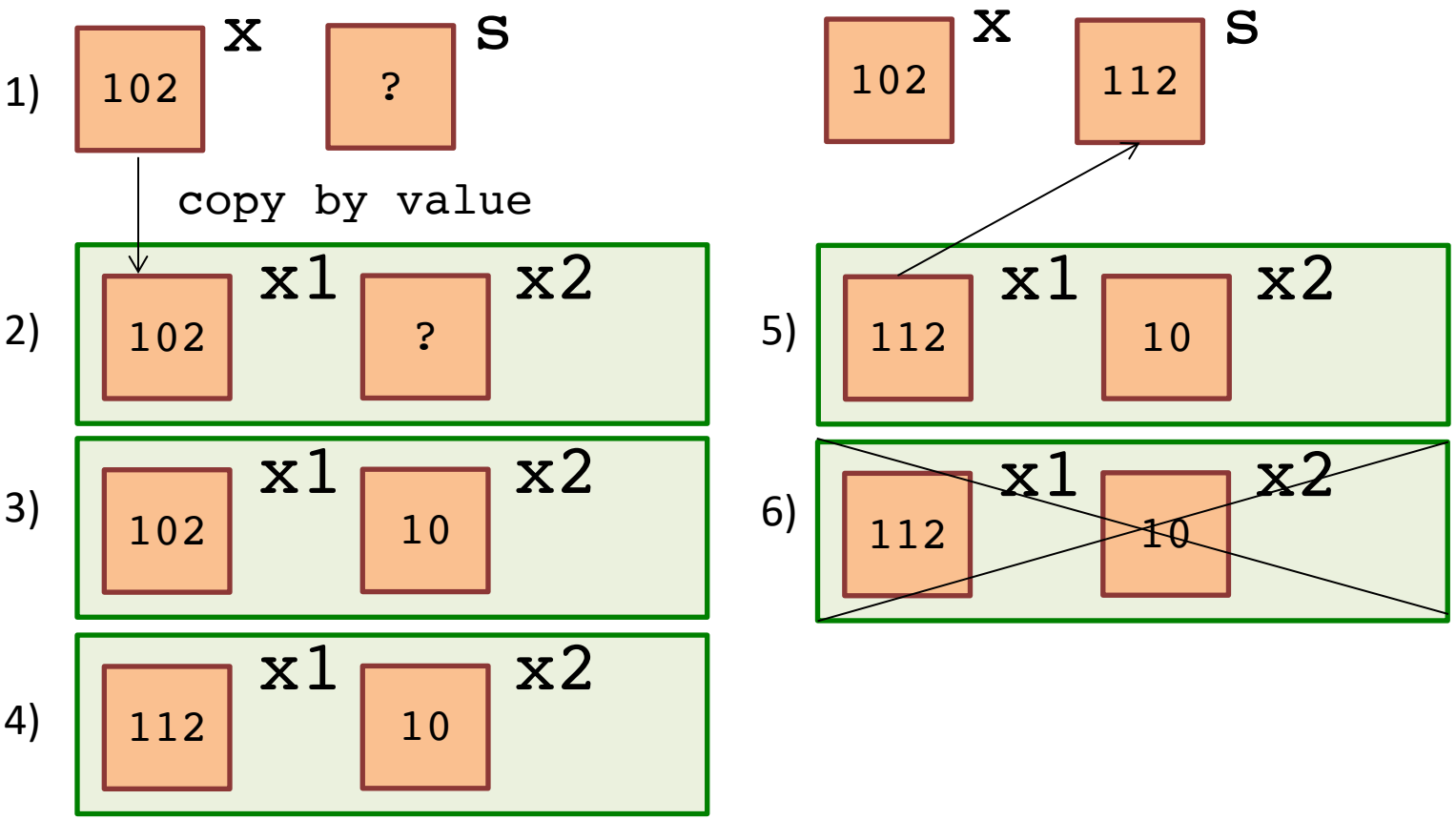
```
int sum( int x1, int x2=4 ){
  x1 = x1+x2;
  return x1;
}

int main() {
  int x=102;
  int s = sum( x, 10 );
  printf("x:%d  s:%d\n", x, s);

  int y=8;
  printf("y:%d  s:%d\n", x, sum( sum(y,2) ) );
}
```

```
x:102 s:112
y:8 s:14
```

Functions

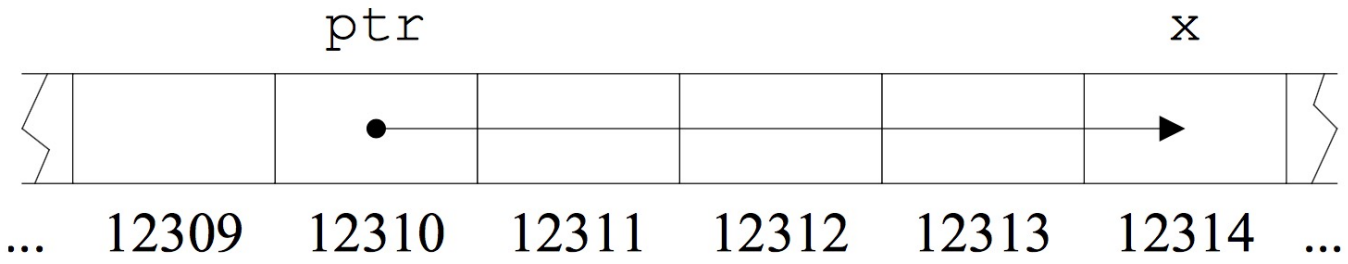
```
int sum( int x1, int x2=4 ){
  x1 = x1+x2;
  return x1;
}
```

```
s = sum(x,10);
```



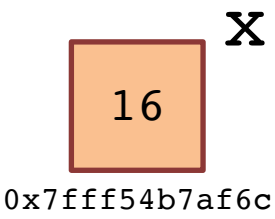
C++ Pointers

- For a C++ program, the memory of a computer is like a succession of memory cells, each one byte in size, and each with a unique address.
  - Data representations larger than one byte occupy memory cells that have consecutive address
- Each cell can be easily located in the memory by means of its unique address
- When a variable is declared, the memory needed to store its value is assigned a specific location in memory (its memory address)
- it may be useful for a program to be able to obtain the address of a variable during runtime in order to access data cells that are at a certain position relative to it.
- **Pointers** are variables storing integers (usually memory addresses of other variables)
- Arrays are pointers!



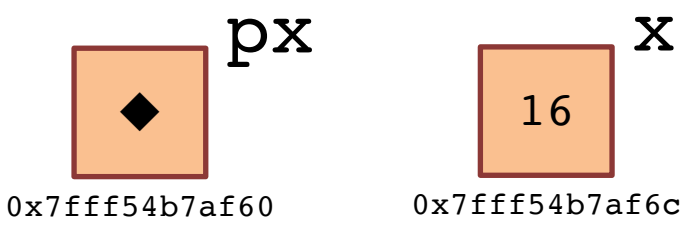
C++ Pointers

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

int main() {    // {
  int x = 16;
  int *px = NULL;    //pointer to integer is defined and initialized
  px = &x;           //value of px is set to the address "&" of x
  int **ppx = &(px); //pointer to pointer
  cout<<x<<" "<<px<<" "<<ppx<<endl;
  cout<<&x<<" "<<&px<<" "<<&ppx<<endl;
  cout<<*px<<" "<<*ppx<<" "<<**ppx<<endl;
  *px = 25;
  cout<<x<<" "<<*px<<" "<<**ppx<<endl;
  return 0;       //return value of the function main
}
```



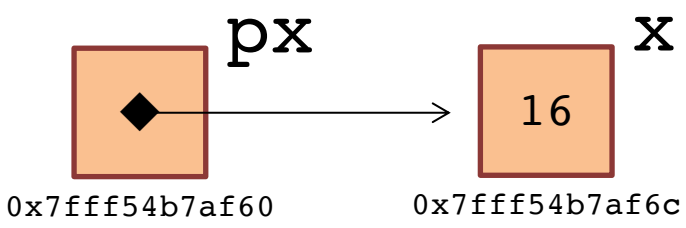
C++ Pointers

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

int main() {    // {
  int x = 16;
  int* px = NULL;    //pointer to integer is defined and initialized
  px = &x;           //value of px is set to the address "&" of x
  int** ppx = &(px); //pointer to pointer
  cout<<x<<" "<<px<<" "<<ppx<<endl;
  cout<<&x<<" "<<&px<<" "<<&ppx<<endl;
  cout<<*px<<" "<<*ppx<<" "<<**ppx<<endl;
  *px = 25;
  cout<<x<<" "<<*px<<" "<<**ppx<<endl;
  return 0;       //return value of the function main
}
```



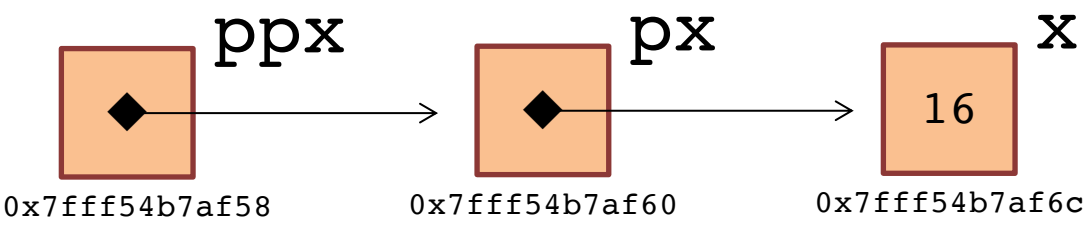
C++ Pointers

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

int main() {    // {
  int x = 16;
  int* px = NULL;    //pointer to integer is defined and initialized
  px = &x;           //value of px is set to the address "&" of x
  int** ppx = &(px); //pointer to pointer
  cout<<x<<" "<<px<<" "<<ppx<<endl;
  cout<<&x<<" "<<&px<<" "<<&ppx<<endl;
  cout<<*px<<" "<<*ppx<<" "<<**ppx<<endl;
  *px = 25;
  cout<<x<<" "<<*px<<" "<<**ppx<<endl;
  return 0;       //return value of the function main
}
```



C++ Pointers

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

int main() {    // {
  int x = 16;
  int* px = NULL;    //pointer to integer is defined and initialized
  px = &x;           //value of px is set to the address "&" of x
  int** ppx = &(px); //pointer to pointer
  cout<<x<<" "<<px<<" "<<ppx<<endl;
  cout<<&x<<" "<<&px<<" "<<&ppx<<endl;
  cout<<*px<<" "<<*ppx<<" "<<**ppx<<endl;
  *px = 25;
  cout<<x<<" "<<*px<<" "<<**ppx<<endl;
  return 0;       //return value of the function main
}
```



C++ Pointers

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

int main() {    // {
  int x = 16;
  int* px = NULL;    //pointer to integer is defined and initialized
  px = &x;           //value of px is set to the address "&" of x
  int** ppx = &(px); //pointer to pointer
  cout<<x<<" "<<px<<" "<<ppx<<endl;
  cout<<&x<<" "<<&px<<" "<<&ppx<<endl;
  cout<<*px<<" "<<*ppx<<" "<<**ppx<<endl;
  *px = 25;
  cout<<x<<" "<<*px<<" "<<**ppx<<endl;
  return 0;       //return value of the function main
}
```

```
16 0x7fff54b7af6c 0x7fff54b7af60
0x7fff54b7af6c 0x7fff54b7af60 0x7fff54b7af58
16 0x7fff54b7af6c 16
25 25 25
```

C++ Pointers & Functions

- C++ functions copy arguments by value
- C++ functions are independent scopes
  - ⓘ Use pointers or references to modify arguments with functions

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

void increase(int x)     { x++; };        //pass by value
void p_increase(int* px) { (*px)++; };    //pass by pointer
void r_increase(int& x)  { x++; };        //pass by reference

int main(){
  int i=0;
  increase(i); cout<<i<<endl;
  int* pi = &i;
  p_increase(pi); cout<<i<<endl;
  int j=0;
  r_increase(j); cout<<j<<endl;
  return 0;
}
```


```
0
1
1
```

File writing/reading (C)

Opening Files

You can use the **fopen( )** function to create a new file or to open an existing file. This call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. The prototype of this function call is as follows –

```
FILE *fopen( const char * filename, const char * mode );
```

Here, **filename** is a string literal, which you will use to name your file, and access **mode** can have one of the following values –

Mode	Description
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for both reading and writing.
w+	Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.
a+	Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

If you are going to handle binary files, then you will use following access modes instead of the above mentioned ones –

```
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"
```

File writing/reading (C)

Closing a File

To close a file, use the fclose( ) function. The prototype of this function is –

```
int fclose( FILE *fp );
```

The **fclose(-)** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file **stdio.h**.

There are various functions provided by C standard library to read and write a file, character by character, or in the form of a fixed length string.

File writing/reading (C)

Writing a File

Following is the simplest function to write individual characters to a stream –

```
int fputc( int c, FILE *fp );
```

The function **fputc()** writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise **EOF** if there is an error. You can use the following functions to write a null-terminated string to a stream –

```
int fputs( const char *s, FILE *fp );
```

The function **fputs()** writes the string **s** to the output stream referenced by fp. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error. You can use **int fprintf(FILE *fp,const char *format, ...)** function as well to write a string into a file. Try the following example.

Make sure you have **/tmp** directory available. If it is not, then before proceeding, you must create this directory on your machine.

```
#include <stdio.h>

main() {
   FILE *fp;

   fp = fopen("/tmp/test.txt", "w+");
   fprintf(fp, "This is testing for fprintf...\n");
   fputs("This is testing for fputs...\n", fp);
   fclose(fp);
}
```

When the above code is compiled and executed, it creates a new file **test.txt** in /tmp directory and writes two lines using two different functions. Let us read this file in the next section.

File writing/reading (C)

Reading a File

Given below is the simplest function to read a single character from a file –

```
int fgetc( FILE * fp );
```

The **fgetc()** function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error, it returns **EOF**. The following function allows to read a string from a stream –

```
char *fgets( char *buf, int n, FILE *fp );
```

The functions **fgets()** reads up to n-1 characters from the input stream referenced by fp. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including the new line character. You can also use **int fscanf(FILE *fp, const char *format, ...)** function to read strings from a file, but it stops reading after encountering the first space character.

File writing/reading (C)

```c
#include <stdio.h>

main() {

   FILE *fp;
   char buff[255];

   fp = fopen("/tmp/test.txt", "r");
   fscanf(fp, "%s", buff);
   printf("1 : %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("2: %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("3: %s\n", buff );
   fclose(fp);

}
```

When the above code is compiled and executed, it reads the file created in the previous section and produces the following result −

```
1 : This
2: is testing for fprintf...


3: This is testing for fputs...
```

Let's see a little more in detail about what happened here. First, **fscanf()** read just **This** because after that, it encountered a space, second call is for **fgets()** which reads the remaining line till it encountered end of line. Finally, the last call **fgets()** reads the second line completely.

File writing/reading (C)

```c
#include <stdio.h>

int main () {
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   if(fp == NULL) {
      perror("Error in opening file");
      return(-1);
   }

   while(1) {
      c = fgetc(fp);
      if( feof(fp) ) {
         break ;
      }
      printf("%c", c);
   }
   fclose(fp);

   return(0);
}
```

File writing/reading (C)

Binary I/O Functions

There are two functions, that can be used for binary input and output –

```
size_t fread(void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *a_file);
```

Both of these functions should be used to read or write blocks of memories - usually arrays or structures.

File writing/reading (C++)

Input/output with files

C++ provides the following classes to perform output and input of characters to/from files:

- **ofstream:** Stream class to write on files
- **ifstream:** Stream class to read from files
- **fstream:** Stream class to both read and write from/to files.

These classes are derived directly or indirectly from the classes `istream` and `ostream`. We have already used objects whose types were these classes: `cin` is an object of class `istream` and `cout` is an object of class `ostream`. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use `cin` and `cout`, with the only difference that we have to associate these streams with physical files. Let's see an example:

```
1  // basic file operations
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main () {
7    ofstream myfile;
8    myfile.open ("example.txt");
9    myfile << "Writing this to a file.\n";
10   myfile.close();
11   return 0;
12 }
```

```
[file example.txt]
Writing this to a file.
```

This code creates a file called `example.txt` and inserts a sentence into it in the same way we are used to do with `cout`, but using the file stream `myfile` instead.

But let's go step by step:

C++ Classes

- C++ is an object oriented language that allows the interaction in the code between different modules
  - The basic data structure is the Object
  - The procedures used to handle, update and retrieve the Object information are called methods
- In C++, Objects are implemented in the form of Classes
- If not specified otherwise, many instances of classes can be defined

C++ Classes

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

class Rectangle
{
     public:
     double base; double height;          //members of the class

     double getArea(void);                //methods of the class
     void setBaseHeight(double,double);   //methods of the class
};

double Rectangle::getArea(void)
{  return base * height; }

void Rectangle::setBaseHeight( double b, double h )
{ base=b; height=h; }

int main( )
{
   Rectangle rec;
   rec.setBaseHeight(2, 5.5);  // the operator "." is used to access methods
   double area = rec.getArea();
   cout<<"Area "<<area<<endl;
   return 0;
}
```

```
Area 11
```

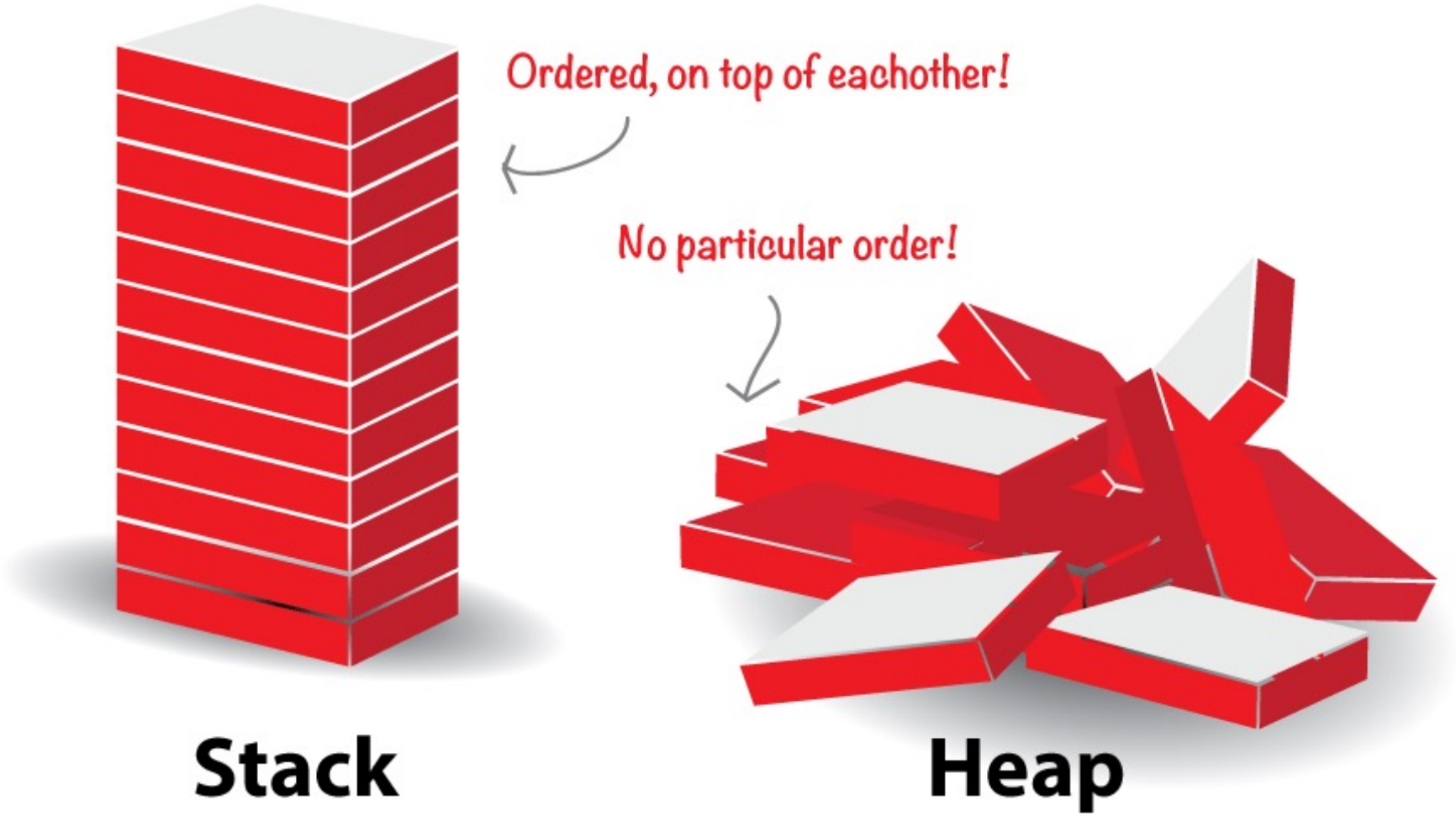
C++ Classes

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

class Rectangle
{
     public:
     Rectangle();  //constructor
     double base; double height;
     double getArea(void);
     void setBaseHeight(double,double);
     private:
     ~Rectangle();  //destructor
};

Rectangle::Rectangle(){
     printf("Creating-Rectangle\n");
     base=1; height=1;
     return;
}

Rectangle::~Rectangle(){
     printf("Destroying-Rectangle\n");
     return;
}
```

```
int main( )
{
   Rectangle rec;
   double area = rec.getArea();
   cout<<"Area "<<area<<endl;
   rec.setBaseHeight(2, 4);
   area = rec.getArea();
   cout<<"Area "<<area<<endl;
   return 0;
}
```

```
Creating-Rectangle
Area 1
Area 8
Destroying-Rectangle
```

Memory



Memory

	stack	**heap**
access speed	fast	slow
memory management	CPU - automatic	manual allocation and deallocation
efficient use of memory	yes	no - may become fragmented, or leak
variable scope	local	global
size	limited (OS dependent)	no limit (or rather the limit of physical memory)
variable resizing	no	yes
need pointers?	no	yes

Memory Management

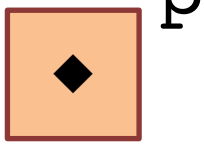
- C++ allocates memory when a variable is declared in a scope
- The memory remains allocated inside the scope
- The memory is freed and again available when the scope is resolved

```
int main() {
    int* p = NULL;
    if( true )
        {
        int x = 10;
        p = &x;
        }
    /*some code here….*/
    cout<<*p<<endl;
    return 0;
}
```

Memory Management

- C++ allocates memory when a variable is declared in a scope
- The memory remains allocated inside the scope
- The memory is freed and again available when the scope is resolved

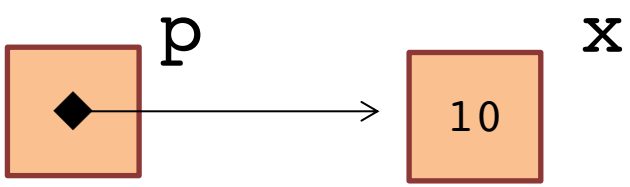
```
int main() {
     int* p = NULL;
     if( true )
          {
          int x = 10;
          p = &x;
          }
     /*some code here….*/
     cout<<*p<<endl;
     return 0;
}
```



Memory Management

- C++ allocates memory when a variable is declared in a scope
- The memory remains allocated inside the scope
- The memory is freed and again available when the scope is resolved

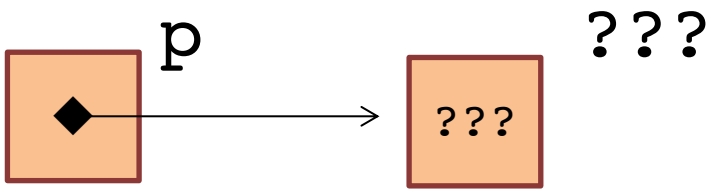
```
int main() {
     int* p = NULL;
     if( true )
          {
          int x = 10;
          p = &x;
          }
     /*some code here….*/
     cout<<*p<<endl;
     return 0;
}
```



Memory Management

- C++ allocates memory when a variable is declared in a scope
- The memory remains allocated inside the scope
- The memory is freed and again available when the scope is resolved

```
int main() {
     int* p = NULL;
     if( true )
          {
          int x = 10;
          p = &x;
          }
     /*some code here….*/
     cout<<*p<<endl;
     return 0;
}
```



Memory Management

- The **new** operator can be used to allocate memory that will remain allocated until the user manually frees it
- The **delete** operator de-allocates the memory that has been previously allocated
  - ⓘ For each new operation, a delete operation is needed (when the variable is no more used) in order to not waste memory

```
int* ptoint(int x){
  int* px = new int;
  *px = x;
  return px;
}

int main(){
  int* pi=NULL;
  for(int i=0; i<3; i++)
    {
      pi = ptoint(i);
      cout<<*pi<<endl;
      delete pi;
    }
  return 0;
}
```

```
0
1
2
```

Memory Management

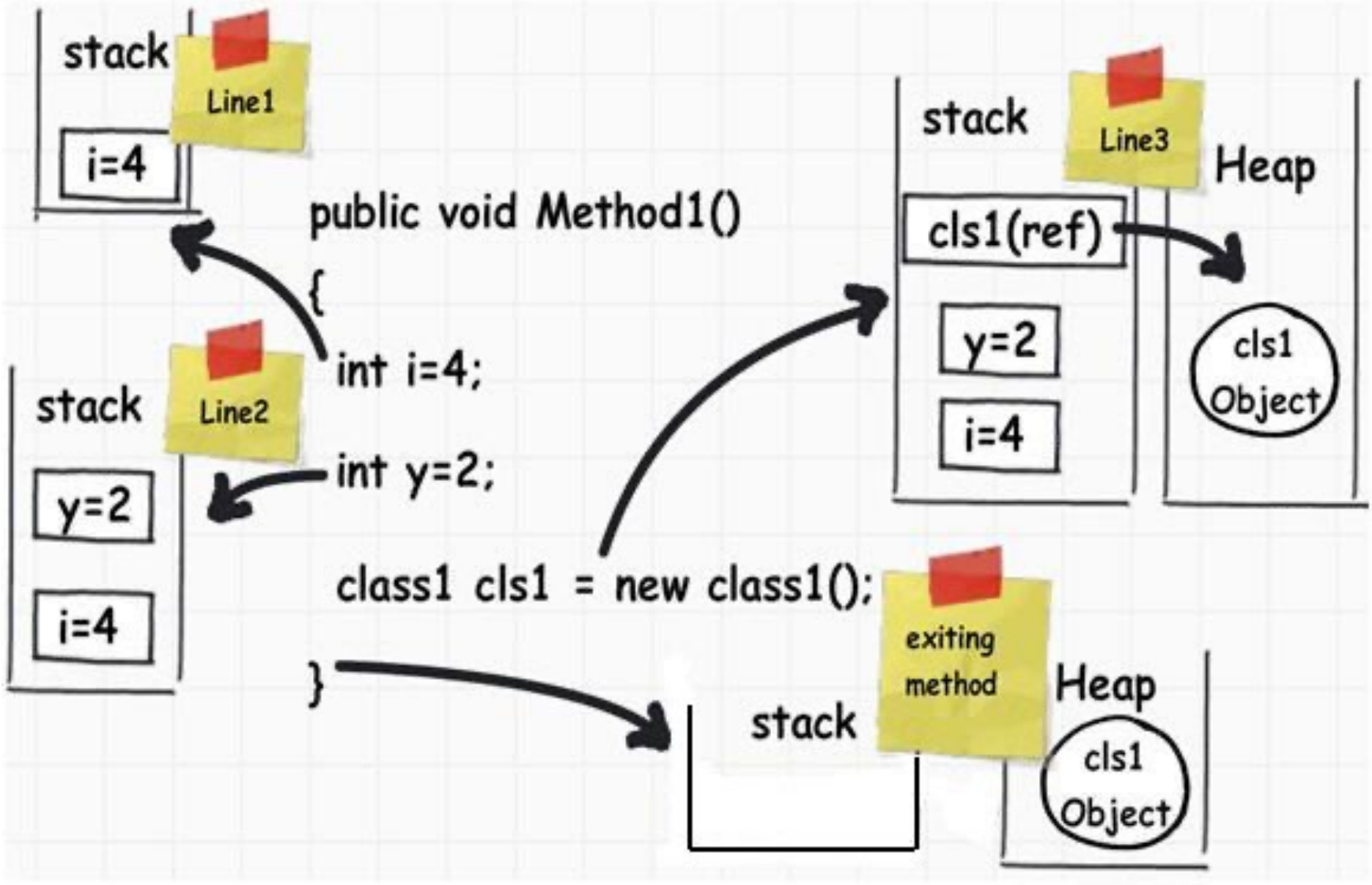
- The **new** operator can be used to allocate memory that will remain allocated until the user manually frees it
- The **delete** operator de-allocates the memory that has been previously allocated
  - ⓘ For each new operation, a delete operation is needed (when the variable is no more used) in order to not waste memory

```
int main()
{
  Rectangle *rec = new Rectangle();
  (*rec).setBaseHeight(2, 4);
  double area = (*rec). getArea();
  cout<<"Area "<<area1<<endl;
  rec->setBaseHeight(4, 5);
  area = rec->getArea();
  cout<<"Area "<<area<<endl;
  delete rec;
  return 0;
}
```

```
Creating-Rectangle
Area 8
Area 20
Destroying-Rectangle
```

- The → operator can be used as shortcut for **(*point_to_class).**

Memory


stack
Line1
i=4
public void Method1()
{
int i=4;
int y=2;
class1 cls1 = new class1();
}
stack
Line2
y=2
i=4
stack
Line3
Heap
cls1(ref)
y=2
i=4
cls1
Object
exiting
method
stack
Heap
cls1
Object

C++ Classes

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

class Rectangle{
     public:
     double base; double height;          //members of the class

     double getArea(void);                //methods of the class
     void setBaseHeight(double,double);   //methods of the class
};

double Rectangle::getArea(void){
     return base*height;
}

void Rectangle::setBaseHeight(double b, double h){
     base=b;
     height=h;
}




int main(){
   Rectangle rec;
   rec.setBaseHeight(2, 5.5);  // the operator "." is used to access methods
   double area = rec.getArea();
   cout<<"Area "<<area<<endl;
   return 0;
}
```

```
Area 11
```

C++ Classes

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

class Rectangle{
     public:
     double base; double height;          //members of the class

     double getArea(void);                //methods of the class
     void setBaseHeight(double,double);   //methods of the class
};

double Rectangle::getArea(void){
     return base*height;
}

void Rectangle::setBaseHeight(double b, double h){
     base=b;
     height=h;
}




int main(){
   Rectangle* prec = new Rectangle();
   prec->setBaseHeight(2, 5.5);  // the operator "->" is used to access methods
   double area = prec->getArea();
   cout<<"Area "<<area<<endl;
   return 0;
}
```

```
Area 11
```

C++ Classes

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

class Rectangle{
     public:
      Rectangle();  //constructor
      double base;
      double height;
      double getArea(void);
      void setBaseHeight(double,double);
      ~Rectangle();  //destructor
};

Rectangle::Rectangle(){
      printf("Creating-Rectangle\n");
      base=1; height=1;
      return;
}

Rectangle::~Rectangle(){
      printf("Destroying-Rectangle\n");
      return;
}



int main(){
   Rectangle rec;
   double area = rec.getArea();
   cout<<"Area "<<area<<endl;
   rec.setBaseHeight(2, 4);
   area = rec.getArea();
   cout<<"Area "<<area<<endl;
   return 0;
}
```

```
Creating-Rectangle
Area 1



Area 8

Destroying-Rectangle
```

C++ Classes

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

class Rectangle{
     public:
      Rectangle();  //constructor
      double base;
      double height;
      double getArea(void);
      void setBaseHeight(double,double);
      ~Rectangle();  //destructor
};

Rectangle::Rectangle(){
      printf("Creating-Rectangle\n");
      base=1; height=1;
      return;
}

Rectangle::~Rectangle(){
      printf("Destroying-Rectangle\n");
      return;
}



int main(){
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<"Area "<<area<<endl;
   prec->setBaseHeight(2, 4);
   area = prec->getArea();
   cout<<"Area "<<area<<endl;
   return 0;
}
```

```
Creating-Rectangle
Area 1

                ?

Area 8

Destroying-Rectangle
```

C++ Classes

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

class Rectangle{
     public:
      Rectangle();  //constructor
      double base;
      double height;
      double getArea(void);
      void setBaseHeight(double,double);
      ~Rectangle();  //destructor
};

Rectangle::Rectangle(){
      printf("Creating-Rectangle\n");
      base=1; height=1;
      return;
}

Rectangle::~Rectangle(){
      printf("Destroying-Rectangle\n");
      return;
}



int main(){
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<"Area "<<area<<endl;
   prec->setBaseHeight(2, 4);
   area = prec->getArea();
   cout<<"Area "<<area<<endl;
   return 0;
}
```

```
Creating-Rectangle
Area 1



Area 8
```

C++ Classes

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

class Rectangle{
     public:
      Rectangle();  //constructor
      double base;
      double height;
      double getArea(void);
      void setBaseHeight(double,double);
      ~Rectangle();  //destructor
};

Rectangle::Rectangle(){
      printf("Creating-Rectangle\n");
      base=1; height=1;
      return;
}

Rectangle::~Rectangle(){
      printf("Destroying-Rectangle\n");
      return;
}


int main(){
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<"Area "<<area<<endl;
   prec->setBaseHeight(2, 4);
   area = prec->getArea();
   cout<<"Area "<<area<<endl;
   delete prec;
   return 0;
}
```

```
Creating-Rectangle
Area 1



Area 8
Destroying-Rectangle
```

C++ Classes

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

class Rectangle{
     public:
      Rectangle();  //constructor
      double base;
      double height;
      double getArea(void);
      void setBaseHeight(double,double);
      ~Rectangle();  //destructor
};

Rectangle::Rectangle(){
      printf("Creating-Rectangle\n");
      base=1; height=1;
      return;
}

Rectangle::~Rectangle(){
      printf("Destroying-Rectangle\n");
      return;
}


int main(){
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<"Area "<<area<<endl;
   prec->base=2; prec->height=4;
   area = prec->getArea();
   cout<<"Area "<<area<<endl;
   delete prec;
   return 0;
}
```

```
Creating-Rectangle
Area 1



Area 8
Destroying-Rectangle
```

C++ Classes

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

class Rectangle{
     public:
      double getArea(void);
      void setBaseHeight(double,double);
     private:
      double base;
      double height;
};
```

```
int main(){
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<"Area "<<area<<endl;
   prec->base=2; prec->height=4;
   area = prec->getArea();
   cout<<"Area "<<area<<endl;
   delete prec;
   return 0;
}
```

Neither compiles…

C++ Classes

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

class Rectangle{
     public:
      double getArea(void);
      void setBaseHeight(double,double);
      double base;
      double height;
};
```

```
int main(){
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<“Area “<<area<<endl;
   prec->base=2; prec->height=4;
   area = prec->getArea();
   cout<<“Area “<<area<<endl;
   if (prec->base=3) {
       printf(“Base is 3\n”);
   }
   cout<<“Area “<<area<<endl;
   delete prec;
   return 0;
}
```

```
Area 1


Area 8

Base is 3

Area 12
```

!

C++ Classes

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

class Rectangle{
     public:
      double getArea(void);
      void setBaseHeight(double,double);
      double getBase();
      double getHeight();
     private:
      double base;
      double height;
};

double Rectangle::getBase(){ return base; }

double Rectangle::getHeight(){ return height; }



int main(){
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<"Area "<<area<<endl;
   prec->base=2; prec->height=4;
   area = prec->getArea();
   cout<<"Area "<<area<<endl;
   if (prec->getBase()=3) {
       printf("Base is 3\n");
   }
   cout<<"Area "<<area<<endl;
   delete prec;
   return 0;
}
```

```
Area 1


Area 8

Base is 3

Area 8
```

! (the 'if' is anyhow logically wrong…)

C++ Classes

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

class Rectangle{
     public:
      double getArea(void);
      void setBaseHeight(double,double);
      inline double getBase(){ return base; };
      inline double getHeight() { return height; };
     private:
      double base;
      double height;
};
```

```
int main(){
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<"Area "<<area<<endl;
   prec->base=2; prec->height=4;
   area = prec->getArea();
   cout<<"Area "<<area<<endl;
   if (prec->getBase()=3) {
       printf("Base is 3\n");
   }
   cout<<"Area "<<area<<endl;
   delete prec;
   return 0;
}
```

This (*inline*) tells the compiler to substitute the function call with all the instructions it contains:

✔ avoids to the program execution to jump here and there on the stack (from the reached point to the place where the function binary part is stored) and to continue its normal sequential execution

✖ the memory (stack) print of the executable is, in general, larger (a function called several times is repeated several times in memory).. It must be used only when the "function is small".

C++ Classes

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

class Rectangle{
     public:
      double getArea(void);
      void setBaseHeight(double,double);
      static void whatAmI();
      double base;
      double height;
};

void Rectangle::whatAmI(){
     printf("I am a Rectangle!\n");
}
```

```
int main(){
   Rectangle::whatAmI();
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<"Area "<<area<<endl;
   prec->base=2; prec->height=4;
   area = prec->getArea();
   cout<<"Area "<<area<<endl;
   prec->whatAmI();
   delete prec;
   return 0;
}
```

```
I am a Rectangle!


Area 1


Area 8
I am a Rectangle!
```

C++ Classes

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

class Rectangle{
     public:
      double getArea(void);
      void setBaseHeight(double,double);
      double getDiffArea(Rectangle* prec2);
     private:
      double base;
      double height;
};

double Rectangle::getDiffArea(Rectangle* prec2){
      double area  = getArea();
      double area2 = prec2->getArea();
      return area-area2;
}
```

What is better to pass as parameter to the function, for the second rectangle:

- pointer to it
- it
- reference to it

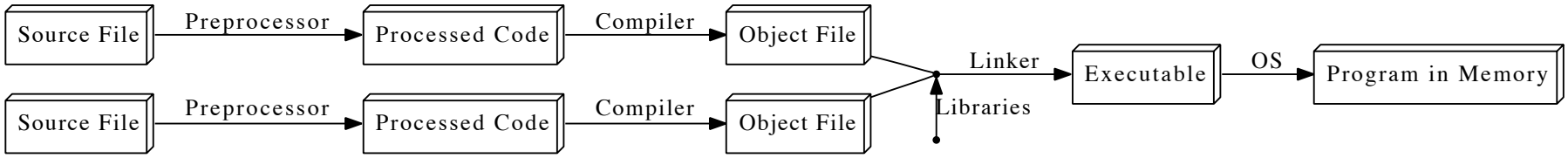
and why?

```
int main(){
   Rectangle* prec = new Rectangle();
   double area = prec->getArea();
   cout<<"Area "<<area<<endl;
   Rectangle* prec2 = new Rectangle();
   prec2->setBaseHeight(2, 4);
   area = prec2->getArea();
   cout<<"Area "<<area<<endl;
   area = prec->getDiffArea(prec2);
   cout<<"Area "<<area<<endl;
   return 0;
}
```

```
Area 1


Area 8

Area -7
```

Compiler

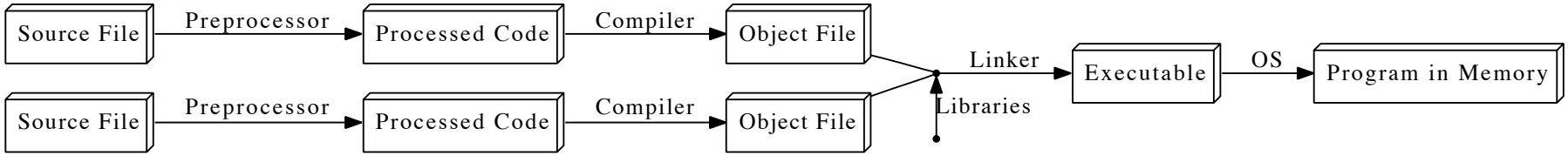
- The compilation procedure translates the source file(s) high-level instructions to low-level machine instructions



- The Compiler program build object files, that contain the instructions translated from every single source files
  - Object files may contain gaps in the program, that depends on the level of interaction with other pieces of code
- The Linker program "fills the gaps", parsing together the object files with eventually external pieces of precompiled code (libraries)
- The executable is produced and can be run
- The instruction for compilation can be complicated → a dedicated language (Makefile) is often used to (try to) make our lives easier

Compiler

- The compilation procedure translates the source file(s) high-level instructions to low-level machine instructions



During the Compilation we must pass the path where to search for the header files, for example `esterna.h`, installed into `/usr/path_esterna/include`:

```
#include "esterna.h"

int main(){

  int var = func_in_esterna(5.3);
  return 0;
}
```

we need to pass the path (`/usr/path_esterna/include`) where to search for `esterna.h`:

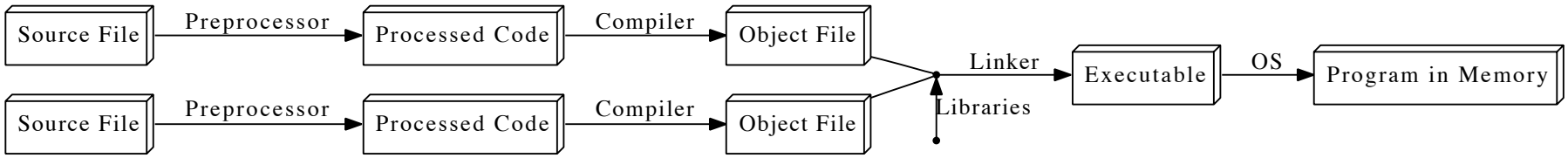
```
vvagelli@Firefly~/test $ g++ -I/usr/path_esterna/include -c test.C –o test.o
```

Having the header files, the compiler can check if the syntax (return and input types) of the external functions used, is correct, so check their interface.

So far the implementation is not know and, moreover, no compiled binary is provided

Compiler

- The compilation procedure translates the source file(s) high-level instructions to low-level machine instructions



During the Linking we must pass the path where to search for the compiled libraries (*.so, *.a or *.dylib), for example `libesterna.so`, installed into `/usr/path_esterna/lib`:

```
#include "esterna.h"

int main(){

  int var = func_in_esterna(5.3);
  return 0;
}
```

we need to pass the path (`/usr/path_esterna/lib`) where to search for `libesterna.so` and which library to link, `libesterna.so` (in the same dir we can have more libraries):

```
vvagelli@Firefly~/test $ g++ test.o -L/usr/path_esterna/lib –lesterna –o test
```

The symbols (i.e. functions) required (by `test.o`) are taken from `libesterna.so` and linked into the executable, `test`.

The `–l` option doesn't want the leading `lib`, nor the extension (`.so`): `libgsl.so` → `-lgsl`

Compiler

- The compilation of a simple standalone programs looks like

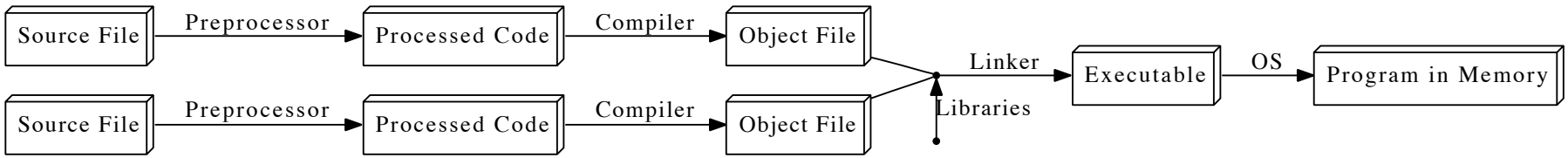
```
vvagelli@Firefly~/test $ g++ test.C –o test
vvagelli@Firefly~/test $ ls -altrh
-rw-r--r--   1 vvagelli  watchmen   1.7K Oct  9 13:03 test.C
-rwxr-xr-x   1 vvagelli  watchmen   11K  Oct  9 13:04 test
vvagelli@Firefly~/test $./test
→ the programs runs ☺
```

- When the projects is more complex, the single line to type may look like

```
vvagelli@Firefly~/test $ g++ -g -O -Wall –pedantic test.C -o exe/test -
D_DEBUG_ -I/Users/vvagelli/root/root5.34/include  -
L/Users/vvagelli/root/root5.34/lib -lCore -lCint -lRIO -lNet -lHist -lGraf -
lGraf3d -lGpad -lTree -lRint -lPostscript -lMatrix -lPhysics -lMathCore -
lThread -lpthread -Wl,-rpath,/Users/vvagelli/root/root5.34/lib -
stdlib=libc++ -lm –ldl
```

- Things become “easier” (according to some guys) using the `Makefile` programming language
- `Makefile` automatize the compilation and linking processes according to the recent changes in the code
- The `Makefile` code has to be written in the Makefile file, that has to live in the main folder of your C++ project

Compiler (step by step)




```
vvagelli@Firefly~/test $ man gcc
…
OPTIONS
   Stage Selection Options
       -E  Run the preprocessor stage.

       -fsyntax-only
           Run the preprocessor, parser and type checking stages.

       -S  Run the previous stages as well as LLVM generation and
           optimization stages and target-specific code generation,
           producing an assembly file.

       -c  Run all of the above, plus the assembler, generating a target
".o" object file.

       no stage selection option
           If no stage selection option is specified, all stages above are
           run, and the linker is run to combine the results into an
               executable or shared library.
```

Compilation (step by step)

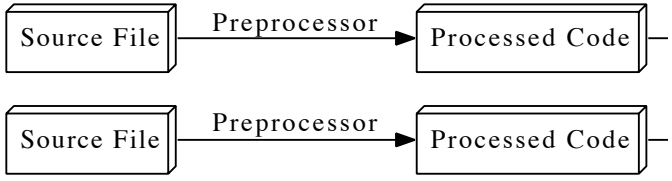
- Let's do a simple "Hello world!" program. Let's start with a single source file, *program.C:*

```
#include <stdio.h>

void print();

int main()
{
  print();
  return 0;
}

void print(){
  printf("Hello, World!\n");
  return;
}
```

- To preprocess, compile and link, we can issue:

```
vvagelli@Firefly~/test $ g++ program.C -o program
```

Compiler (step by step)




```
vvagelli@Firefly~/test $ man gcc
…
OPTIONS
   Stage Selection Options
       -E  Run the preprocessor stage.
```

- by running ‘gcc –E’ one could produce the preprocessed version of the source file (not so useful)

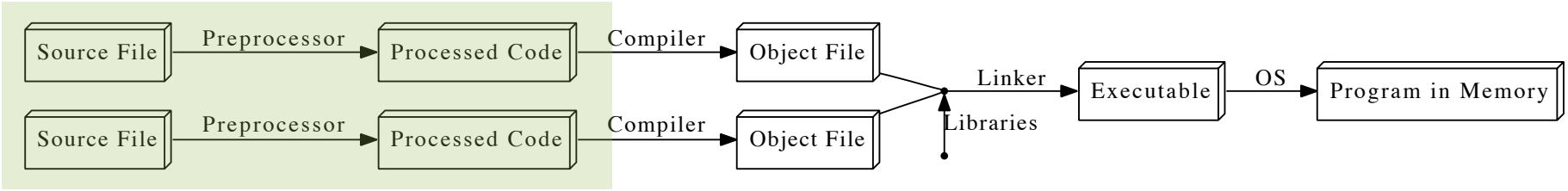
```
vvagelli@Firefly~/test $ g++ program.C -E -o program.i
```

- the ‘program.i’ file will look like:

```
# 1 "program.C"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
…
# 230 "/usr/include/stdio.h" 3 4
extern "C" {
void clearerr(FILE *);
int fclose(FILE *);
int feof(FILE *);
…
```

Preprocessor directives

- Steps from source code to machine level instructions



- The steps are performed before the program starts to run
  - Some languages follow (more or less) the same procedure BUT during the execution process. This slows down the program execution.
    - This is one of the reasons why C++ code runs far faster than code in many more recent languages.
- The Preprocessor modifies the source code according to user directives

```
#include "external_header.h" //includes external definitions
#define CONSTANT 10 //the string CONSTANT is replaced everywhere by 10

int main(){
#ifdef _DEBUG_
  cout<<"Do This"<<endl;
#else
  cout<<"Do That"<<endl;
#endif
  return 0;
}
```

Preprocessor directives

```
vvagelli@Firefly~/test $ g++ -g -O -Wall test.C -o exe/test -D_DEBUG_ -
I/Users/vvagelli/root/root5.34/include  -L/Users/vvagelli/root/root5.34/lib
-lCore -lCint -lRIO -lNet -lHist -lGraf -lGraf3d -lGpad -lTree -lRint -
lPostscript -lMatrix -lPhysics -lMathCore -lThread -lpthread -Wl,-
rpath,/Users/vvagelli/root/root5.34/lib -stdlib=libc++ -lm –ldl
```

```
#include “external_header.h” //includes external definitions
#define CONSTANT 10 //the string CONSTANT is replaced everywhere by 10

int main(){
#ifdef _DEBUG_
  cout<<“Do This”<<endl;
#else
  cout<<“Do That”<<endl;
#endif
  return 0;
}
```

Preprocessor directives

```
vvagelli@Firefly~/test $ g++ -g -O -Wall test.C -o exe/test -D_DEBUG_ -
I/Users/vvagelli/root/root5.34/include  -L/Users/vvagelli/root/root5.34/lib
-lCore -lCint -lRIO -lNet -lHist -lGraf -lGraf3d -lGpad -lTree -lRint -
lPostscript -lMatrix -lPhysics -lMathCore -lThread -lpthread -Wl,-
rpath,/Users/vvagelli/root/root5.34/lib -stdlib=libc++ -lm –ldl
```

```
#include “external_header.h” //includes external definitions
#define CONSTANT 10 //the string CONSTANT is replaced everywhere by 10

int array_sum[CONSTANT];

int main(){
  int sum=0;
  for (int ii=0; ii<CONSTANT; ii++) {
    sum+=ii;
    array_sum[ii]=sum;
#ifdef _DEBUG_
    printf(“ii=%d) sum=%d\n”, ii, sum);
#endif
  }
  return 0;
}
```

Direttive preprocessore

The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives –

Directive	**Description**
#define	Substitutes a preprocessor macro.
#include	Inserts a particular header from another file.
#undef	Undefines a preprocessor macro.
#ifdef	Returns true if this macro is defined.
#ifndef	Returns true if this macro is not defined.
#if	Tests if a compile time condition is true.
#else	The alternative for #if.
#elif	#else and #if in one statement.
#endif	Ends preprocessor conditional.
#error	Prints error message on stderr.
#pragma	Issues special commands to the compiler, using a standardized method.

Direttive preprocessore

Preprocessors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use *#define* for constants to increase readability.

```
#include <stdio.h>
#include "myheader.h"
```

These directives tell the CPP to get stdio.h from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file.

```
#undef  FILE_SIZE
#define FILE_SIZE 42
```

It tells the CPP to undefine existing FILE_SIZE and define it as 42.

```
#ifndef MESSAGE
   #define MESSAGE "You wish!"
#endif
```

It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

```
#ifdef DEBUG
   /* Your debugging statements here */
#endif
```

It tells the CPP to process the statements enclosed if DEBUG is defined. This is useful if you pass the *-DDEBUG* flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

Direttive preprocessore

Predefined Macros

ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

Macro	Description
__DATE__	The current date as a character literal in "MMM DD YYYY" format.
__TIME__	The current time as a character literal in "HH:MM:SS" format.
__FILE__	This contains the current filename as a string literal.
__LINE__	This contains the current line number as a decimal constant.
__STDC__	Defined as 1 when the compiler complies with the ANSI standard.

Let's try the following example –

```
#include <stdio.h>

main() {

   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );
   printf("ANSI :%d\n", __STDC__ );

}
```

When the above code in a file **test.c** is compiled and executed, it produces the following result –

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

Direttive preprocessore

Preprocessor Operators

The C preprocessor offers the following operators to help create macros −

The Macro Continuation (\) Operator

A macro is normally confined to a single line. The macro continuation operator (\) is used to continue a macro that is too long for a single line. For example −

```
#define  message_for(a, b)  \
   printf(#a " and " #b ": We love you!\n")
```

The Stringize (#) Operator

The stringize or number-sign operator ('#'), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro having a specified argument or parameter list. For example −

```
#include <stdio.h>

#define  message_for(a, b)  \
   printf(#a " and " #b ": We love you!\n")

int main(void) {
   message_for(Carole, Debra);
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Carole and Debra: We love you!
```

Direttive preprocessore

The Token Pasting (##) Operator

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token. For example −

```
#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void) {
   int token34 = 40;
   tokenpaster(34);
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
token34 = 40
```

It happened so because this example results in the following actual output from the preprocessor −

```
printf ("token34 = %d", token34);
```

This example shows the concatenation of token##n into token34 and here we have used both **stringize** and **token-pasting**.

Direttive preprocessore

The Defined() Operator

The preprocessor **defined** operator is used in constant expressions to determine if an identifier is defined using #define. If the specified identifier is defined, the value is true (non-zero). If the symbol is not defined, the value is false (zero). The defined operator is specified as follows –

```
#include <stdio.h>

#if !defined (MESSAGE)
   #define MESSAGE "You wish!"
#endif

int main(void) {
   printf("Here is the message: %s\n", MESSAGE);
   return 0;
}
```

Try it

When the above code is compiled and executed, it produces the following result –

```
Here is the message: You wish!
```

Direttive preprocessore

Parameterized Macros

One of the powerful functions of the CPP is the ability to simulate functions using parameterized macros. For example, we might have some code to square a number as follows −

```
int square(int x) {
   return x * x;
}
```

We can rewrite above the code using a macro as follows −

```
#define square(x) ((x) * (x))
```

Macros with arguments must be defined using the **#define** directive before they can be used. The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between the macro name and open parenthesis. For example −

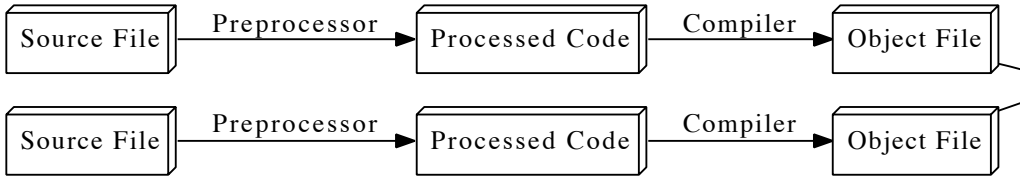
```
#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void) {
   printf("Max between 20 and 10 is %d\n", MAX(10, 20));
   return 0;
}
```

Try it

When the above code is compiled and executed, it produces the following result −

```
Max between 20 and 10 is 20
```

Compiler (step by step)

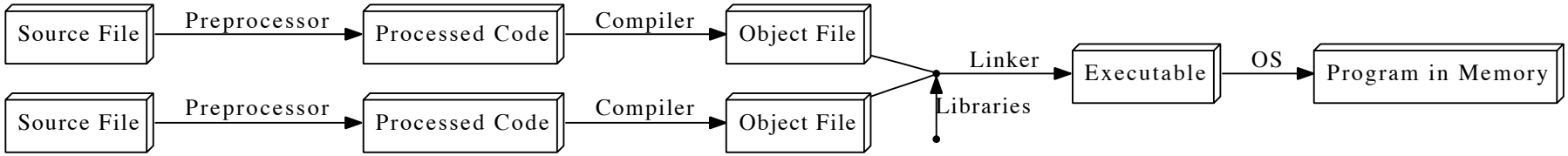



```
vvagelli@Firefly~/test $ man gcc
…
OPTIONS
   Stage Selection Options
       -c  Run all of the above, plus the assembler, generating a target
".o" object file.
```

- by running ‘gcc –c’ we can compile the source code, *without* linking (and so producing and executable [or a library])

```
vvagelli@Firefly~/test $ g++ -c program.C -o program.o
```

Compiler (step by step)




```
vvagelli@Firefly~/test $ man gcc
…
OPTIONS
   Stage Selection Options
    no stage selection option
           If no stage selection option is specified, all stages above are
           run, and the linker is run to combine the results into an
                executable or shared library.
```

- so after the object(s) creation we can finally link into an executable:

```
vvagelli@Firefly~/test $ g++ program.o -o program
```

- and so, summing the two:

```
vvagelli@Firefly~/test $ g++ -c program.C -o program.o
vvagelli@Firefly~/test $ g++ program.o -o program
```

- and this is equivalent to:

```
vvagelli@Firefly~/test $ g++ program.C -o program
```

Compiler (step by step)

- this is not only useful, but also needed, when compiling a large ‘project’ made of several source files:

```
vvagelli@Firefly~/test $ ls
program.C
program_func.C
program_func.h
```

```
#include “program_func.h”

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

```
#include <stdio.h>
#include “program_func.h”

void print(){
  printf("Hello, World!\n");
  return;
}
```

```
void print();
```

Compiler (step by step)

- this is not only useful, but often also needed, when compiling a large ‘project’ made of several source files:

```
vvagelli@Firefly~/test $ ls
program.C
program_func.C
program_func.h
```

- to compile the ‘project’, one should compile all the pieces and then link them together, in an executable:

```
vvagelli@Firefly~/test $ g++ -c program_func.C -o program_func.o
vvagelli@Firefly~/test $ g++ -c program.C -o program.o
vvagelli@Firefly~/test $ g++ program.o program_func.o -o program
```

- and this is equivalent to:

```
vvagelli@Firefly~/test $ g++ program_func.C program.C -o program
```

Debug

Two types of bugs in the code

- Syntax errors, identified by the compiler. Easy to fix
- Run-time errors: the program crashes during runtime. Typically this is due to a wrong management of the memory (wrong assignment, access to ill-defined memory blocks etc….)

Read carefully the program dump, sometimes helps to find the problem or in which function the problem happens

Poor man approach: isolate the incriminated lines of code the hard way (using a set of printf, for example)

Programs are available to investigate these problems. In particular (typically installed by default in UNIX environments)

- **gdb (GNU debugger)**, useful to set breakpoints in the code and investigate the value of variables in the code during runtime

  https://www.gnu.org/software/gdb/
- **valgrind**, specialized in memory management and code profiling
  - http://valgrind.org
- **Tools (part of XCode, in MacOsX)**, both the tasks above and also more

Compiler (`Makefile`)

- All the *rules* to compile a program/project can be written in a "script" (including some logic) that automatically execute the whole 'flow' of commands that need to be run

```
CXX           := g++
CXXFLAGS      := -g –O -Wall -pedantic #-g compile with debug flags, -O
optimizes the compilation (whatever that means), -Wall enables all warnings
-pedantic is pedantic
EXT_LIBS      := $(shell root-config --libs) #see below for expanded version
INCLUDES      := $(shell root-config –cflags)#see below for expanded version
FLAGS         := -D_DEBUG_

default: test

test: test.C
    $(CXX) $(CXXFLAGS) test.C -o exe/$@ $(FLAGS) $(INCLUDES) $(EXT_LIBS)

clean:
    rm -f exe/test
```

- The command to execute the script is *make. make* searches for a script file called `Makefile` (or `GNUMakefile`) where to find the *rules* to be executed
- We can call explicitly one of the rules: *make clean*. If we issue just *make*, the first rule is executed (in the example above *make* and *make default* are equivalent

Compiler (Makefile)

```
CXX           := g++
CXXFLAGS      := -g –O -Wall -pedantic #-g compile with debug flags, -O
optimizes the compilation (whatever that means), -Wall enables all warnings
-pedantic is pedantic
EXT_LIBS      := $(shell root-config --libs) #see below for expanded version
INCLUDES      := $(shell root-config –cflags)#see below for expanded version
FLAGS         := -D_DEBUG_

default: test

test: test.C
    $(CXX) $(CXXFLAGS) test.C -o $@ $(FLAGS) $(INCLUDES) $(EXT_LIBS)

clean:
    rm -f exe/test
```

```
vvagelli@Firefly~/test $ ls
Makefile        test.C        exe
vvagelli@Firefly~/test $ make
g++ -g -O -Wall –pedantic test.C -o exe/test -D_DEBUG_ -
I/Users/vvagelli/root/root5.34/include  -L/Users/vvagelli/root/root5.34/lib
-lCore -lCint -lRIO -lNet -lHist -lGraf -lGraf3d -lGpad -lTree -lRint -
lPostscript -lMatrix -lPhysics -lMathCore -lThread -lpthread -Wl,-
rpath,/Users/vvagelli/root/root5.34/lib -stdlib=libc++ -lm –ldl
vvagelli@Firefly~/test $ ./exe/test
```

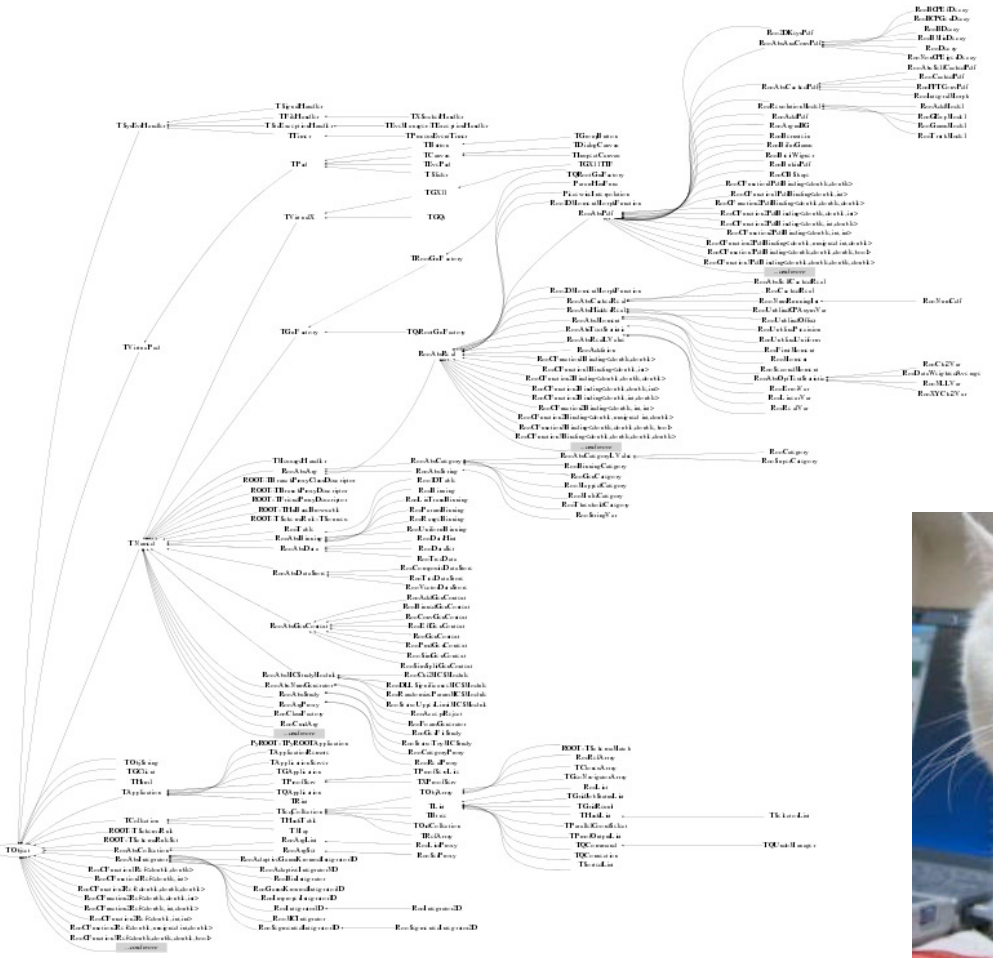
ROOT

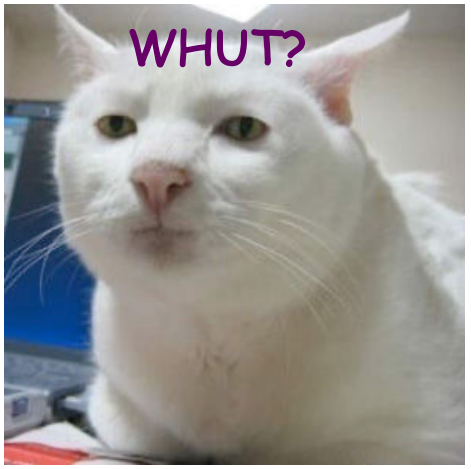
- **ROOT** is a collection of libraries that can be used for numerical (not symbolic) statistical data analysis (and more)
- Mainly developed at CERN for particle physics analysis, but greatly flexible any other field
- More than 1000 C++ classes
- Based on the Modular and Class Inheritance concepts

- These lectures will cover the details of ROOT v5.34, which is most stable and widely used version of ROOT used (mainly) by all particle physics experiments.
- ~~Recently, ROOT v6.00 has been released. While the backend of the software is different, the frontend is basically the same.~~
  Now we're at ROOT 6.24/06. However 99% of the content of the lectures is still valid

ROOT

`https://root.cern.ch/root/html534/ClassIndex.html`





`TObject:` Mother of all ROOT classes, provides default behavior and protocol for all objects in ROOT

ROOT

https://root.cern.ch/root/html534/TH1F.html

→ Which headers to be included

→ Which library to be linked

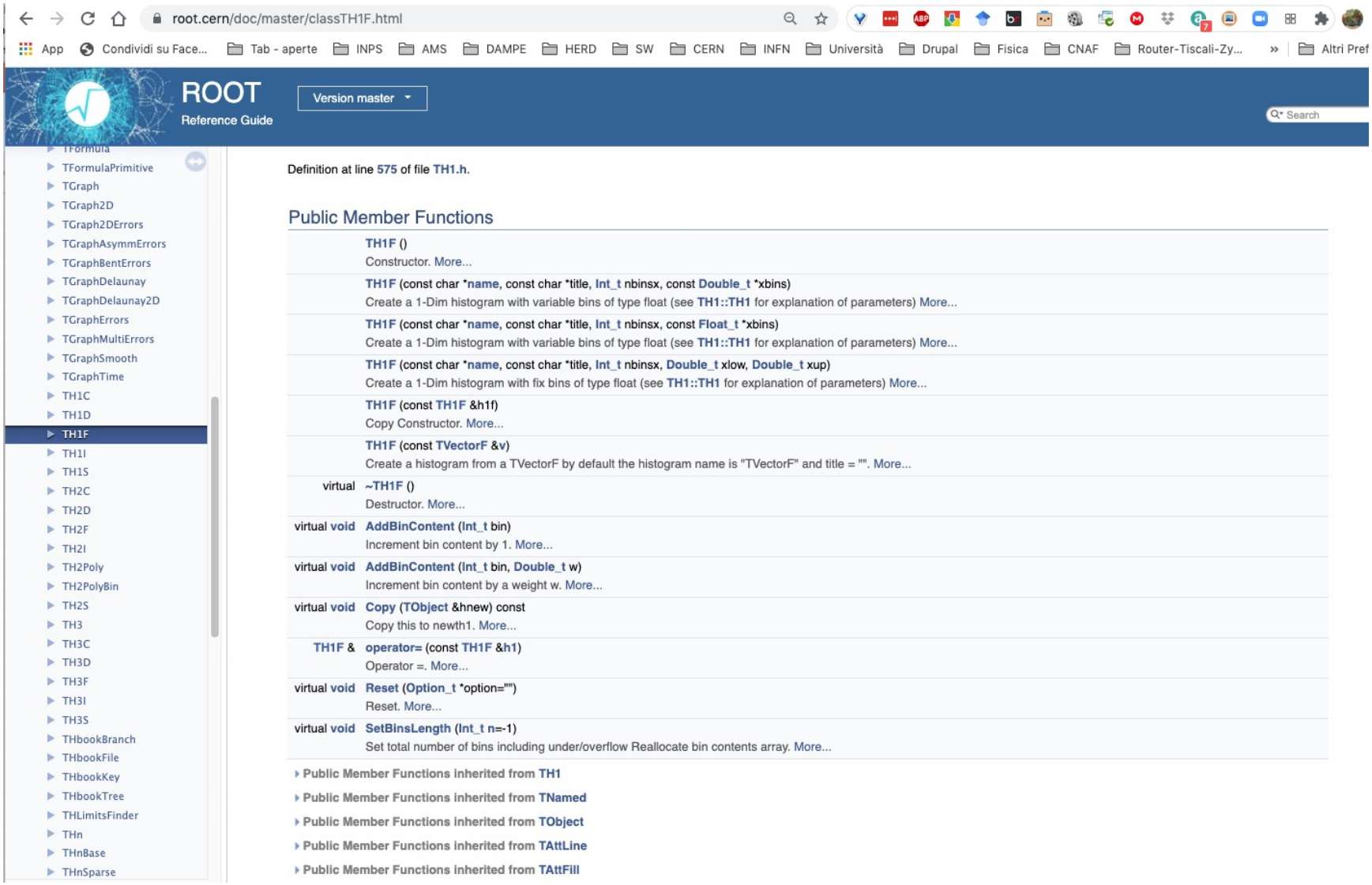
→From which parent class inherits

Class members and methods

Inheritance example: a `TH1F` object is indeed a `TH1` and a `TArrayF` object, but specialized for more specific tasks (and therefore more methods and potentials)

ROOT (v 6)

https://root.cern/doc/master/classTH1F.html

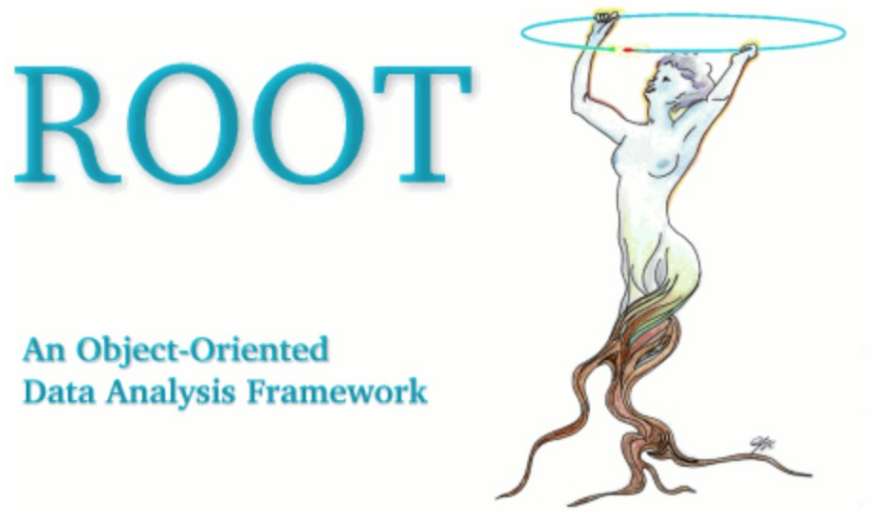


In my opinion a little bit less clear than before, but the content is essentially the same

ROOT

https://root.cern.ch/root/html534/guides/users-guide/ROOTUsersGuide.html

ROOT User's Guide



ROOT User's Guide to be read (at least the first chapters) to understand in details the principles and the basics (handling histograms, functions, fits, graphs, trees, etc….)

May 2013

ROOT Installation (ROOT 5.34)

```
vvagelli@Firefly~/sandobx $ wget https://root.cern.ch/download/root_v5.34.34.source.tar.gz
vvagelli@Firefly~/sandobx $ tar -xvf root_v5.34.34.source.tar.gz
vvagelli@Firefly~/sandobx $ cd root
vvagelli@Firefly~/sandobx/root $ less README/INSTALL
vvagelli@Firefly~/sandobx/root $ ./configure --help
vvagelli@Firefly~/sandobx/root $ ./configure macosx64 --enable-tmva --enable-roofit
vvagelli@Firefly~/sandobx/root $ make

... go take your coffee ...

vvagelli@Firefly~/sandobx/root $ source build/this_root.sh
vvagelli@Firefly~/sandobx/root $ root

... if does not work at the first try, don't panic and carry on

vvagelli@Firefly~/sandobx/root $ ls tutorials

... get yourself confident with this folder. Try some of them!
```

- To load all the settings at login, edit your $HOME/.bashrc or $HOME/.profile

```
export ROOTSYS="/users/walterwhite/root/root5.34"          #path where ROOT lives
export PATH=$ROOTSYS/bin:$PATH
export LD_LIBRARY_PATH=$ROOTSYS/lib:$LD_LIBRARY_PATH
```

ROOT Installation (ROOT 5.34)

```
vvagelli@Firefly /sandobx $ wget https://root.cern.ch/download/root_v5.34.34.source.tar.gz
vvagelli@Firefly~/sandobx $ tar -xvf root_v5.34.34.source.tar.gz
vvagelli@Firefly~/sandobx $ cd root
vvagelli@Firefly~/sandobx/root $ ls
vvagelli@Firefly~/sandobx/root $ ./configure --help
vvagelli@Firefly~/sandobx/root $ ./configure macosx64 --enable-tmva --enable-roofit
vvagelli@Firefly~/sandobx/root $ make
... go take your coffee ...

vvagelli@Firefly~/sandobx/root $ source build/this_root.sh
vvagelli@Firefly~/sandobx/root $ root

... if does not work at the first try, don't panic and carry on

vvagelli@Firefly~/sandobx/root $ ls tutorials

... get yourself confident with this folder. Try some of them!
```

The compilation part now is different: now is done via CMake

- To load all the settings at login, edit your $HOME/.bashrc or $HOME/.profile

```
export ROOTSYS="/users/walterwhite/root/root5.34"          #path where ROOT lives
export PATH=$ROOTSYS/bin:$PATH
export LD_LIBRARY_PATH=$ROOTSYS/lib:$LD_LIBRARY_PATH
```

Interactive ROOT session

- Start an interactive session and write C/C++ code

```
vvagelli@Firefly~ $ root –l
root [0] int i=2;
root [1] for(int j=1; j<5; j++) { i *= j; cout<<i<<endl; }
2
4
12
48
root [2] .q
vvagelli@Firefly~ $
```

Interactive ROOT session

- Create a ROOT macro and run it interactively

```
vvagelli@Firefly~ $ cat cool_macro.C
{
  int i=2;

  for (int j=1; j<5; j++) {
    i *= j;
    cout<<i<<endl;
  }
}
vvagelli@Firefly~ $
```

```
vvagelli@Firefly~ $ root -l cool_macro.C
root [0]
Processing cool_macro.C...
2
4
12
48
root [1] .q
vvagelli@Firefly~ $
```

Interactive ROOT session

- Create a ROOT macro, load it and run it interactively

```
vvagelli@Firefly~ $ cat hallo_world.C
void hallo_world(){
  printf("Hallo world!\n");
}

void another_function(){
  printf("blablabla\n");
}
vvagelli@Firefly~ $
```

```
vvagelli@Firefly~ $ root –l hallo_world.C
root [0]
Processing  hallo_world.C...
Hallo World
root [1] .q
vvagelli@Firefly~ $ root –l
root [0] .L  hallo_world.C        #loads the content of hallo_world.C in the memory
root [1] hallo_world()
Hallo World!
root [2] another_function()
blablabla
root [3] .q
vvagelli@Firefly~ $
```

Interactive ROOT session

- ROOT programs can be interpreted (by the CINT-v5.34 or CLING-v6.00 C++ interpreter) or compiled and run.
  `https://root.cern.ch/root/html534/guides/users-guide/CINT.html`

- The previous examples used CINT/CLING
- CINT interprets the code line-by-line and executes it "on the fly"
  - ✗ If you have a compilation problem in your macro, it tries nevertheless to execute it and, typically, fails (like accessing an ill-defined pointer)
  - ✗ in for/while loops, he reads the command to execute for every iteration
  - ✗ In general, the execution is time consuming and the outcome not reliable
  - ✓ Practical for instant checks and instant macros

Automatic Compiler of ROOT macros

- ROOT provides and automatic compiler (ACLiC) to automatically compile, link and load programs using the C++ compiler and linker

✓ Performances similar to compiled code

✓ Allows an easier handle on graphical output and “on the fly” checks on the results

✗ Does not allow a natural interaction with external libraries or modular codes

Interactive ROOT session

- Create a ROOT macro, compile and load it and run it interactively

```
vvagelli@Firefly~ $ cat hallo_world.C
void hallo_world(){
  printf("Hallo world!\n");
}

void another_function(){
  printf("blablabla\n");
}
vvagelli@Firefly~ $
```

```
vvagelli@Firefly~ $ root
root [0] .L hallo_world.C++
Info in <TMacOSXSystem::ACLiC>: /home/vvagelli/./hallo_world.C.so
root [1] hallo_world()
Hallo World
root [2] another_function()
blablabla
root [3] .q
vvagelli@Firefly~ $ root
```

Interactive ROOT session

- Create a ROOT macro, compile and load it and run it interactively

```
vvagelli@Firefly~ $ cat hallo_world2.C
#include <iostream>

void hallo_world(){
  printf("Hallo world!\n");
}

void another_function(){
  cout<<"blablabla"<<endl;
}
vvagelli@Firefly~ $
```

```
vvagelli@Firefly~ $ root
root [0] .L hallo_world2.C++
Info in <TMacOSXSystem::ACLiC>: /home/vvagelli/./hallo_world.C.so
root [1] hallo_world()
Hallo World
root [2] another_function()
blablabla
root [3] .q
vvagelli@Firefly~ $ root
```

Automatic Compiler of ROOT macros

- ROOT provides and automatic compiler (ACLiC) to automatically compile, link and load programs using the C++ compiler and linker

✓ Performances similar to compiled code

✓ Allows an easier handle on graphical output and “on the fly” checks on the results

✗ Does not allow a natural interaction with external libraries or modular codes

```
vvagelli@Firefly~ $ root
root [0] .L hallo_world.C++
Info in <TMacOSXSystem::ACLiC>: /home/vvagelli/./hallo_world.C.so
root [1] hallo_world()
Hallo World
root [2] another_function()
blablabla
root [3] .q
vvagelli@Firefly~ $ root
```

My personal poor-dumb-man suggestion

- Always compile your code, spot errors and clean warnings
- Go for the compiled solution, especially for complex codes containing many loops and interacting with external libraries
- Go for the ACLIC solution to display results and run simple standalone codes
- Go for the CLIC only solution for easy fast checks and line-by-line code

“full” compiling of ROOT code

- ROOT programs as C++ source code can be compiled linking against ROOT libraries as discussed in previous examples (for example using a standard Makefile)

```
CXX           := g++
CXXFLAGS      := -g –O -Wall -pedantic #-g compile with debug flags, -O
optimizes the compilatiom (whatever that means), -Wall enables all warnings
-pedantic is pedantic
EXT_LIBS      := $(shell root-config --libs) #see below for expanded version
INCLUDES      := $(shell root-config –cflags)#see below for expanded version
FLAGS         := -D_DEBUG_

default: test

test: test.C
    $(CXX) $(CXXFLAGS) test.C -o $@ $(FLAGS) $(INCLUDES) $(EXT_LIBS)

clean:
    rm -f exe/test
```

```
g++ -g -O -Wall –pedantic test.C -o test –D_DEBUG_ -
I/Users/vvagelli/root/root5.34/include -L/Users/vvagelli/root/root5.34/lib -
lCore -lCint -lRIO -lNet -lHist -lGraf -lGraf3d -lGpad -lTree -lRint -
lPostscript -lMatrix -lPhysics -lMathCore -lThread -lpthread -Wl,-
rpath,/Users/vvagelli/root/root5.34/lib -stdlib=libc++ -lm -ldl
```

“full” compiling of ROOT code

- ROOT programs as C++ source code can be compiled linking against ROOT libraries as discussed in previous examples (for example using a standard Makefile)

```
CXX           := g++
CXXFLAGS      := -g –O -Wall -pedantic #-g compile with debug flags, -O
optimizes the compilatiom (whatever that means), -Wall enables all warnings
-pedantic is pedantic
EXT_LIBS      := $(shell root-config --libs) #see below for expanded version
INCLUDES      := $(shell root-config –cflags)#see below for expanded version
FLAGS         := -D_DEBUG_

default: test

test: test.C
    $(CXX) $(CXXFLAGS) test.C -o $@ $(FLAGS) $(INCLUDES) $(EXT_LIBS)

clean:
    rm -f exe/test
```

- The output is a standard C++ output binary file

✓ Preferred mode for clean programming, code efficiency, and portability.

✓ Only respectable solution for heavy algorithms or analysis tasks

✗ Graphical “on the fly” output not straightforward → typically the output is saved in an output ROOT file, and then displayed and analyzed later

TObject

- All objects in ROOT (files, histograms, and more complex) inherits from the `TObject` class
- TObjects provide an interface for I/O, error handling, inspection etc….
- TObjects can be copied (`TObject::Copy()`) and cloned (`TObject:Clone()`)
- Take home message: (almost) everything in ROOT is a TObject

`https://root.cern.ch/root/html534/TObject.html`

TNamed

- Many top-level objects in ROOT that are used for data analysis inherits from the `TNamed` class
- `TNamed` objects are `TObjects` with a name and a title

`https://root.cern.ch/root/html534/TNamed.html`

- What (uniquely) defines an object:
  - CLASS: `TH1`, `TAxis`, `TEfficiency`,….
  - NAME: a unique string to identify the object (like a barcode)
  - TITLE: its nickname

ROOT Conventions

- Classes begin with `T`: `TLine, TTree,` …
- Non-class types end with `_t`: `Int_t, Char_t,` …
- Data members begin with `f: fTree, fEntries,` …
- Member functions begin with capital: `TTree::GetEntries(), TH1::Draw(),` …
- Constants begin with `k`: `kDebug, kRed,` …
- Global variables begin with `g: gEnv, gRandom,` …

Machine Independent Types

- Different machines may have different byte lengths for the same type. For example the `int` type, It may be 16 bits on some old machines and 32 bits on newer ones
- ROOT provides machine independent types
  - **Char_t** Signed Character 1 byte
  - **UChar_t** Unsigned Character 1 byte
  - **Short_t** Signed Short integer 2 bytes
  - **UShort_t** Unsigned Short integer 2 bytes
  - **Int_t** Signed integer 4 bytes
  - **UInt_t** Unsigned integer 4 bytes
  - **Long64_t** signed long integer 8 bytes
  - **ULong64_t** unsigned long integer 8 bytes
  - **Float_t** Float 4 bytes
  - **Double_t** Float 8 bytes
  - **Double32_t** Double 8 bytes in memory, written as a Float 4 bytes
  - **Bool_t** Boolean (0=false, 1=true)

Input/Output

https://root.cern.ch/root/html534/TFile.h

- ROOT objects can be saved and retrieved from memory using the `TFile` interface
- The ROOT files are similar to UNIX directory, containing objects and subdirectories
- Objects written to file via `TObject::Write(const char* name)`
- Objects can be retrieved from files via `TFile::Get(const char* name)` passing the object name. A direct cast to the object class is mandatory.

```
TFile *fout = new TFile("fout.root","recreate");
fout->cd();
h->Write( h->GetName() );
hpx->Write( "some_other_name" );
fout->Close();
```

```
TFile *fin= TFile::Open("fout.root");
fin->ls();
```

```
TFile**      fout.root
 TFile*      fout.root
 KEY: TH2F   hname;1  htitle
...
```

```
TH2F *h = (TH2F*)fin->Get("hname");
if( !h ) { cout<<"h pointing to NULL. Exit"<<endl; return 1; }
else{ h->AnyMethod(); ... }
```

Histograms

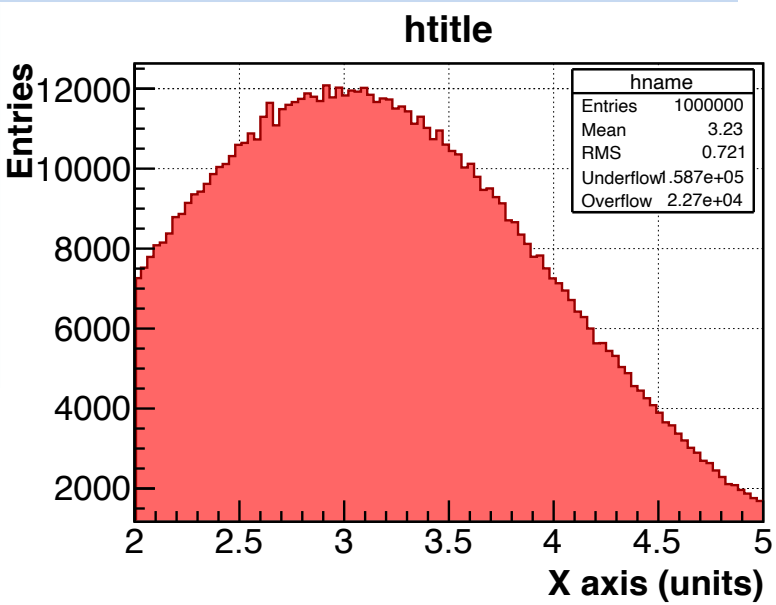
- Histograms are a powerful tool to store and represent BINNED data

```
{
  Int_t nbins=100; Double_t min=2; Double_t max=5;
  TH1F *h = new TH1F("hname","htitle",nbins,min,max);
  Int_t coolnes_level = 1000:
  CoolExperiment *myexp = new CoolExperiment(coolness_level);
  h->GetXaxis()->SetTitle("X axis (units)");
  h->GetYaxis()->SetTitle("Entries");
  h->SetLineColor(kRed+2);
  h->SetLineWidth(2);
  h->SetFillColor(kRed-7);
  for(int i=1; i<=1000000; i++)
    {
      h->Fill( myexp->GetAwesomness() );
    }

  TCanvas *c = new TCanvas("cname","ctitle");
  c->cd()->SetGrid();
  h->Draw("");
}
```

Histograms

- Histograms are a powerful tool to store and represent BINNED data

```
{
  Int_t nbins=100; Double_t min=2; Double_t max=5;
  TH1F *h = new TH1F("hname","htitle",nbins,min,max);
  Int_t coolnes_level = 1000:
  CoolExperiment *myexp = new CoolExperiment(coolness_level);
  h->GetXaxis()->SetTitle("X axis (units)");
  h->GetYaxis()->SetTitle("Entries");
  h->SetLineColor(kRed+2);
  h->SetLineWidth(2);
  h->SetFillColor(kRed-7);
  for(int i=1; i<=1000000; i++)
    {
      h->Fill( myexp->GetAwesomness() );
    }

  TCanvas *c = new TCanvas("cname","ctitle");
  c->cd()->SetGrid();
  h->Draw("");
}
```



Histograms

- Histograms are a powerful tool to store and represent BINNED data

```
{
  Int_t nbins=100; Double_t min=2; Double_t max=5;
  TH1F *h = new TH1F("hname","htitle",nbins,min,max);
  Int_t coolnes level = 1000;
  Cool
  h->G
  h->G
  h->S
  h->S
  h->S
  for(
    {

    }

  TCan
  c->c
  h->D
}
```

class TH1F: public TH1, public TArrayF

TH1F methods

TH1F : histograms with one float per channel. Maximum precision 7 digits

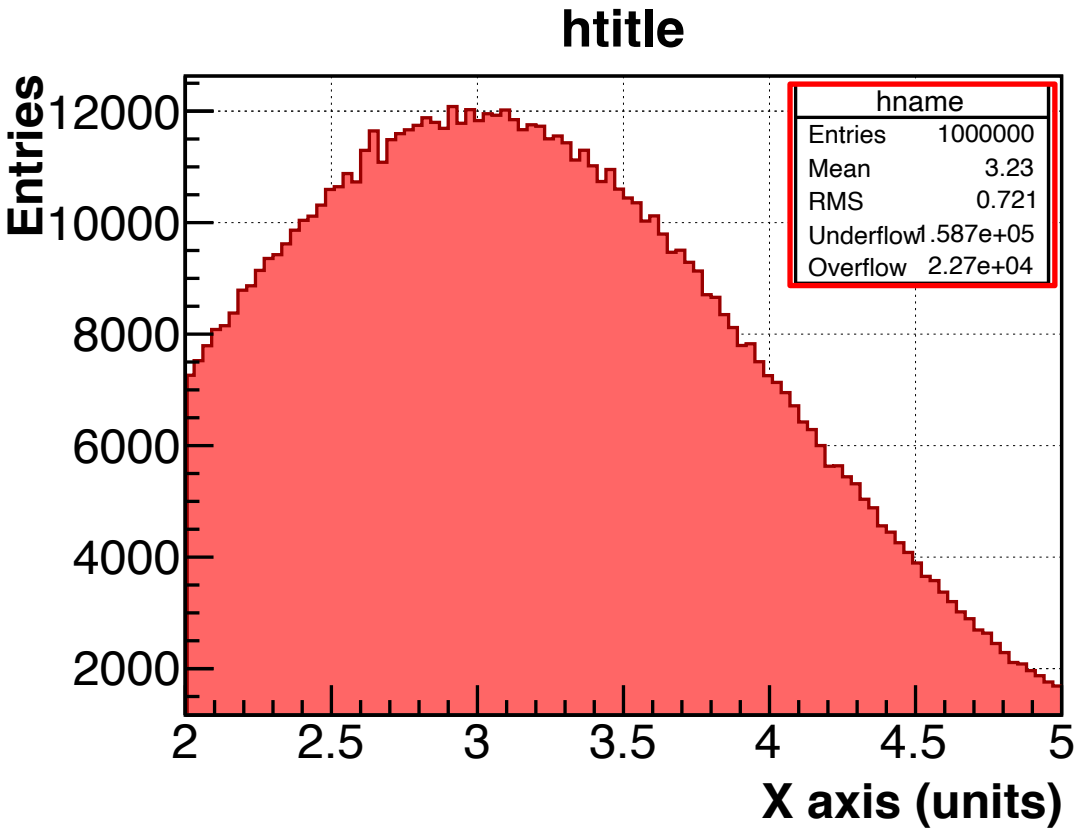
Function Members (Methods)

public:

```
                    TH1F ()
                    TH1F (const TVectorF& v)
                    TH1F (const TH1F& h1f)
                    TH1F (const char* name, const char* title, Int_t nbinsx, const Float_t* xbins)
                    TH1F (const char* name, const char* title, Int_t nbinsx, const Double_t* xbins)
                    TH1F (const char* name, const char* title, Int_t nbinsx, Double_t xlow, Double_t xup)
           virtual ~TH1F ()
              void TObject::AbstractMethod (const char* method) const
virtual Bool_t TH1::Add (const TH1* h1, Double_t c1 = 1)
virtual Bool_t TH1::Add (TF1* h1, Double_t c1 = 1, Option_t* option = "")
```

https://root.cern.ch/root/html534/TH1.html

Histograms

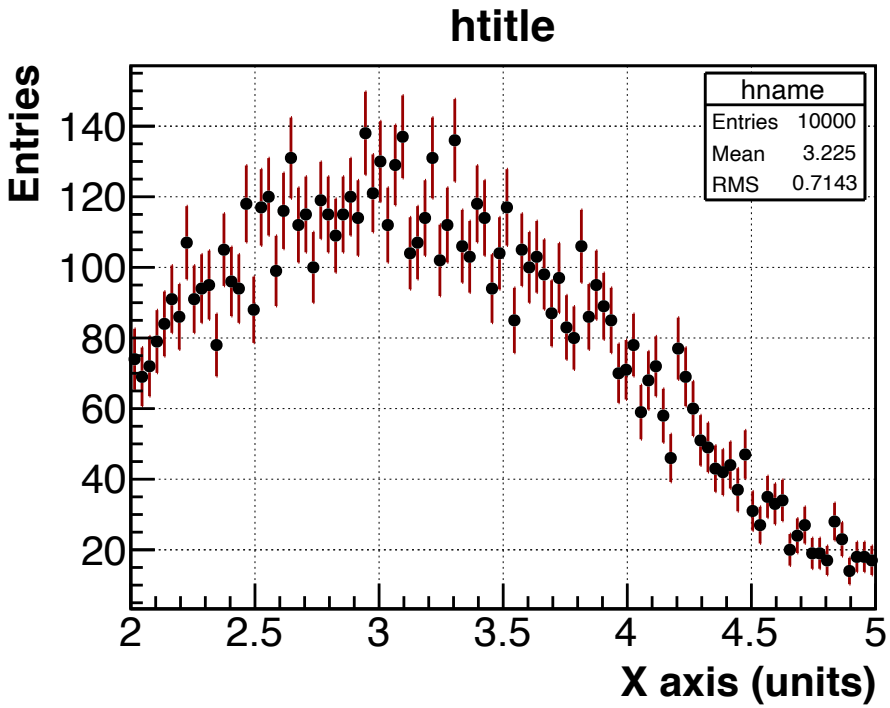


- TH1 bin convention:
  - Bin [0] : underflow
  - Bin [1] : first bin
  - Bin [N]: last bin
  - Bin [N+1] : overflow

Histograms

```
{
  ...
  h->Draw("E1");
  for(int ibin=1; ibin<=h->GetNbinsX(); ibin++)
    cout<<ibin<<" "<<h->GetBinContent(ibin)<<" "<<h->GetBinError(ibin)<<endl;

}
```

```
...
52 85 9.21954
53 105 10.247
54 100 10
55 103 10.1489
56 98 9.89949
...
```



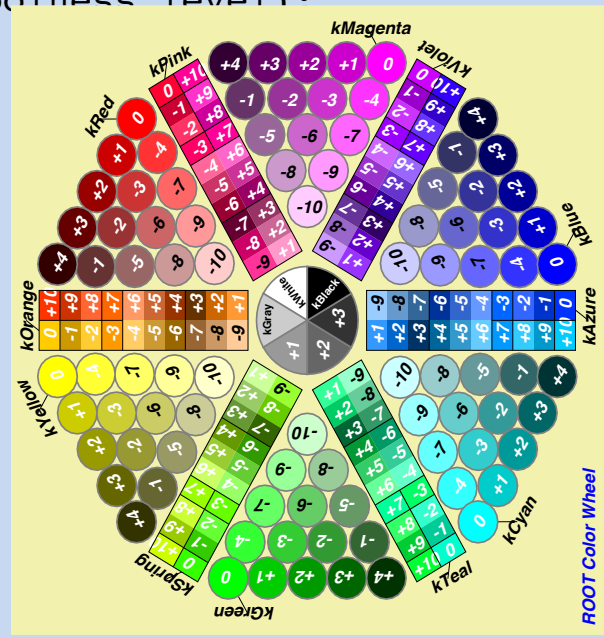
- By default, ROOT assume a poisson distribution for each bin entries (independent from each other)
  - Exercise: demonstrate this fact

https://root.cern.ch/root/html534/TH1.html

Histograms

- Histograms are a powerful tool to store a


```
{
  Int_t nbins=100; Double_t min=2; Dou
  TH1F *h = new TH1F("hname","htitle",
  Int_t coolnes_level = 1000:
  CoolExperiment *myexp = new CoolExperiment(coolness_level);
  h->GetXaxis()->SetTitle("X axis (units)");
  h->GetYaxis()->SetTitle("Entries");
  h->SetLineColor(kRed+2);
  h->SetLineWidth(2);
  h->SetFillColor(kRed-7);
  for(int i=1; i<=1000000; i++)
    {
      h->Fill( myexp->GetAwesomness() );
    }

  TCanvas *c = new TCanvas("cname","ctitle");
  c->cd()->SetGrid();
  h->Draw("");
}
```



https://root.cern.ch/root/html534/TAttFill.html

Histograms

https://root.cern.ch/root/html534/TH1.html

- Useful TH1 methods
  - `TH1::Fill(Double_t value, Double_t weight);`
  - `TH1::GetEntries();`
  - `TH1::GetXaxis(); TH1::GetYaxis();`
  - `TH1::Add(const TH1* h1, const Double_t c=1);`
  - `TH1::Divide(const TH1* h1);`
  - `TH1::GetMean(); TH1::GetRMS(); TH1::GetSkewness();`
  - `TH1::Scale(Double_t c=1);`
  - `TH1::Sumw2();`
  - `TH1::DrawNormalized();`

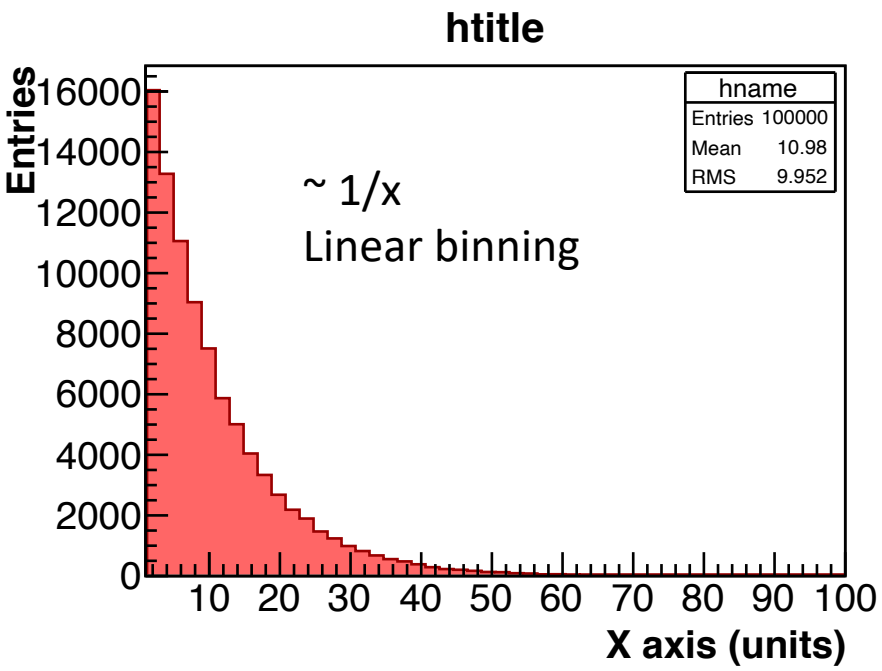
- NB: in ROOT, RMS indicates the standard deviation of the data (sqrt of II momentum around the histogram mean)

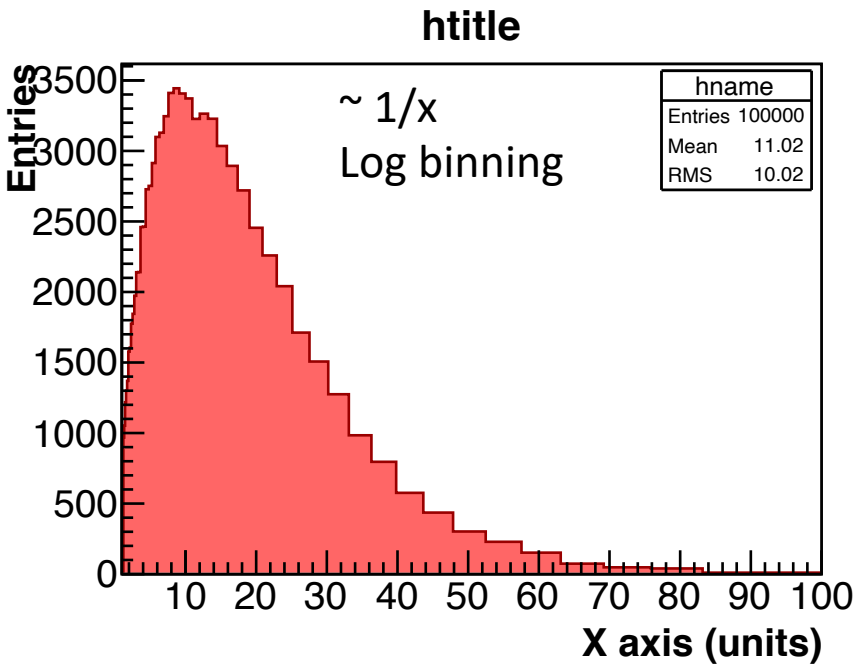
- The histogram axis can also be not uniformly spaced

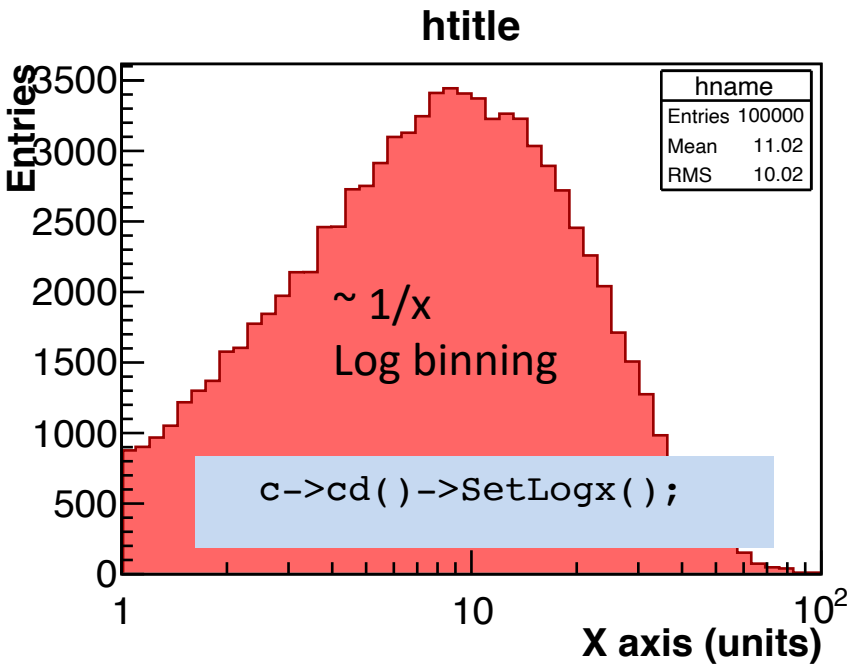
```
Double_t xmin=1; Double_t xmax=100; Int_t N=50;
Double_t *axis = new Double_t[N+1];
Double_t dlog = (TMath::Log10(xmax)-TMath::Log10(xmin))/N;
for(Int_t i=0; i<=50; i++){ axis[i] = pow( 10, log10(xmin) + i*dlog ) ; }
TH1F *h = new TH1F("hname","htitle”,N,axis);
```

htt h

- Useful TH1 meth
  - TH1::Fil
  - TH1::Get
  - TH1::Get
  - TH1::Add ;
  - TH1::Div
  - TH1::Get ness();







Parenthesis…

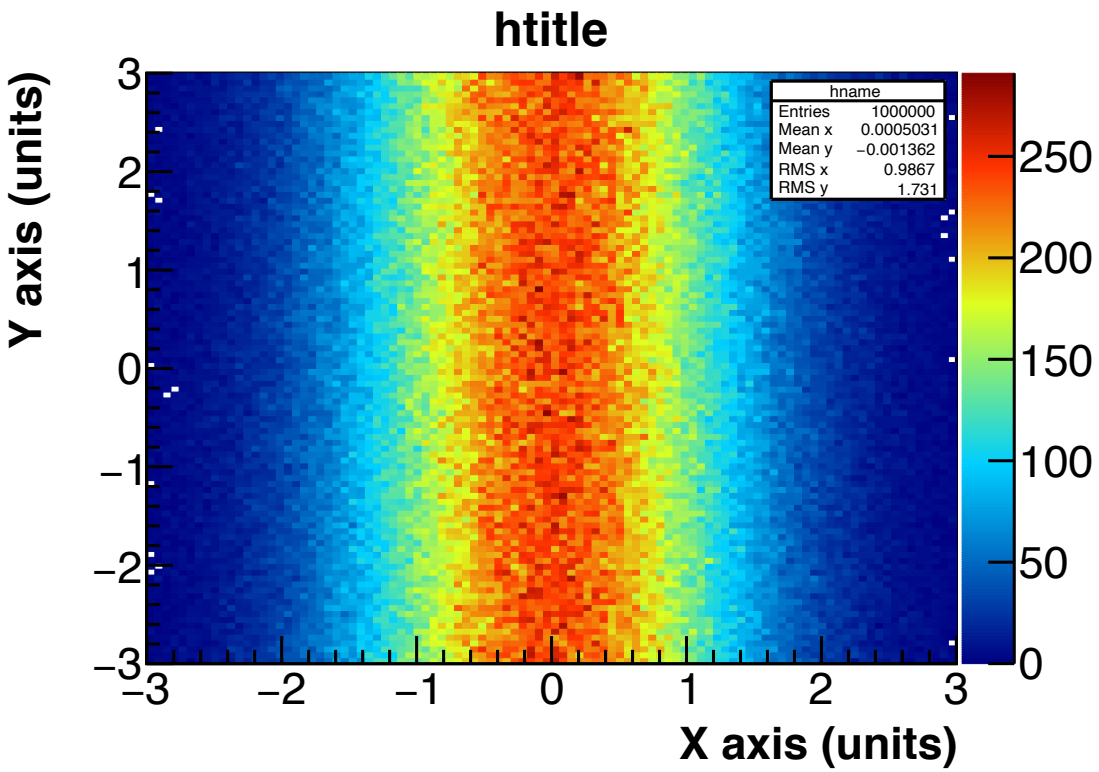
- Histograms, and all other objects in ROOT, can be declared and used in arrays/vectors

```
#include “TH1.h”
#define N 10
#define M 255

int main(){
  TH1F *h[N][M];
  for(int ii=0; ii<N; ii++)
    for(int jj=0; jj<M; jj++)
    {
      h[ii][jj] = new TH1F( Form(“h_%d_%d”,ii,jj), Form(“htitle - %d -
%d”,ii,jj), ii<50?100:200, 0, jj);
    }

  vector<TH1D*> vec;
  for(int ii=0; ii<N; ii++)
  {
    vec.push_back( h[i][0] );
  }
...
}
```

Histograms

https://root.cern.ch/root/html534/TH2.html

```
TH2F *h = new TH2F("hname","htitle;X axis (units);Y axis (units)",100,-3,3,100,-3,3);
for(int i=0; i<1000000;i++) h->Fill(gRandom->Gaus(0,1), gRandom->Uniform(-3,3));
h->Draw("COLZ”);
TH1D *px = h->ProjectionX("h_px",20,50);
TH1D *py = h->ProjectionY("h_py »,40,60);
```

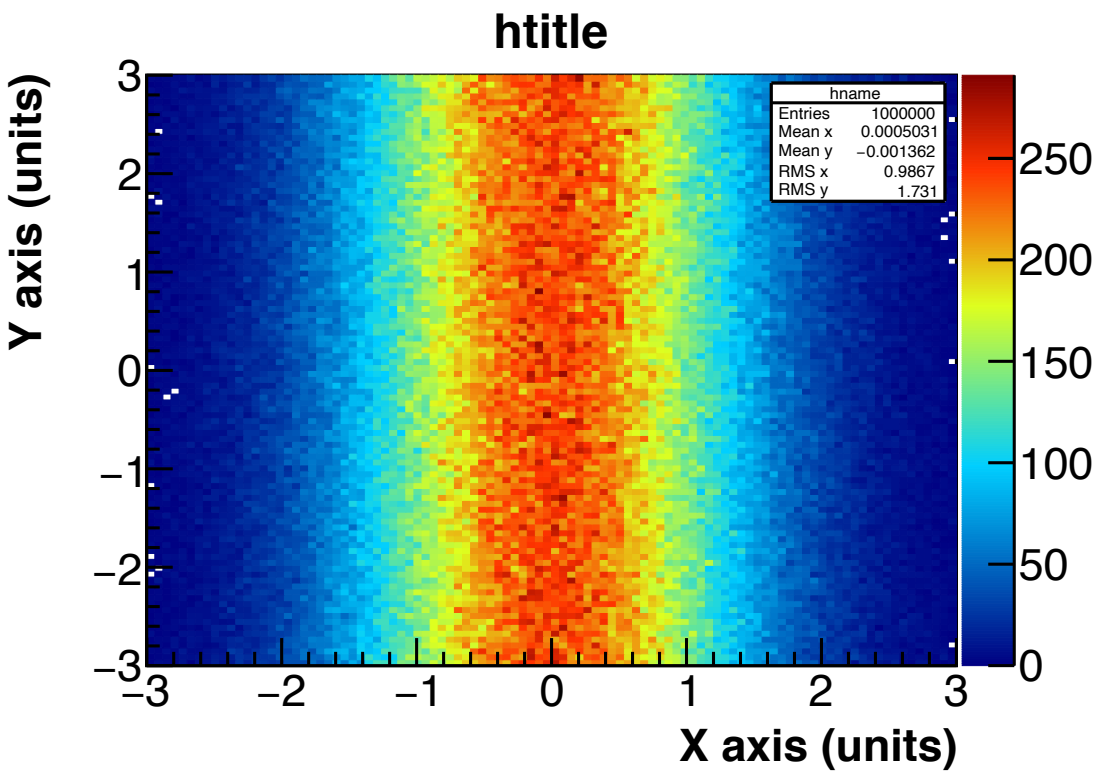


- 2D and 3D histograms managed by the `TH2` and `TH3` classes
- Inherits from `TH1`, with additional methods

Histograms

https://root.cern.ch/root/html534/TH2.html

```
TH2F *h = new TH2F("hname","htitle;X axis (units);Y axis (units)",100,-3,3,100,-3,3);
for(int i=0; i<1000000;i++) h->Fill(gRandom->Gaus(0,1), gRandom->Uniform(-3,3));
h->Draw("COLZ”);
TH1D *px = h->ProjectionX("h_px",20,50);
TH1D *py = h->ProjectionY("h_py »,40,60);
```



- 2D and 3D histograms managed by the `TH2` and `TH3` classes
- Inherits from `TH1`, with additional methods

Parenthesis: TRandom

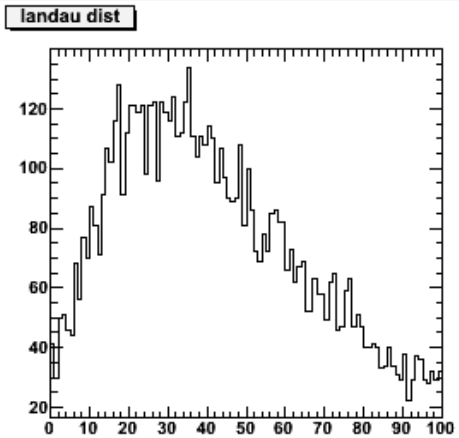
`https://root.cern.ch/doc/master/classTRandom.html`

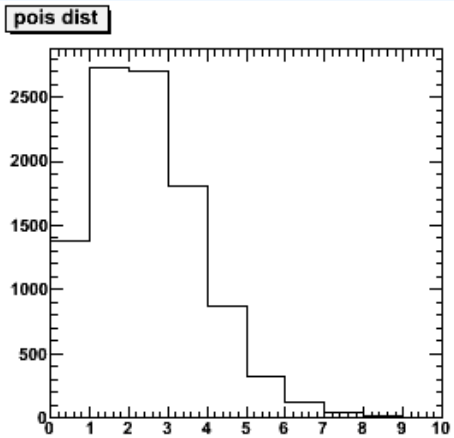
Essentially TRandom returns numbers (~ random) following a deterministic sequence:

```
TRandom3 * rand = new TRandom3();
Double_t  myNum = rand->Gaus( 30.0, 10.0 );
```

- Can be used to fill histograms:

```
TH1F * hlandau = new TH1F("l", "", 100, 0.0, 100. );
for( int i = 0 ; i < 10000 ; i++ )
   hlandau->Fill(  rand->Landau(30.0, 15 ) );
```





Can generate numbers from these distributions

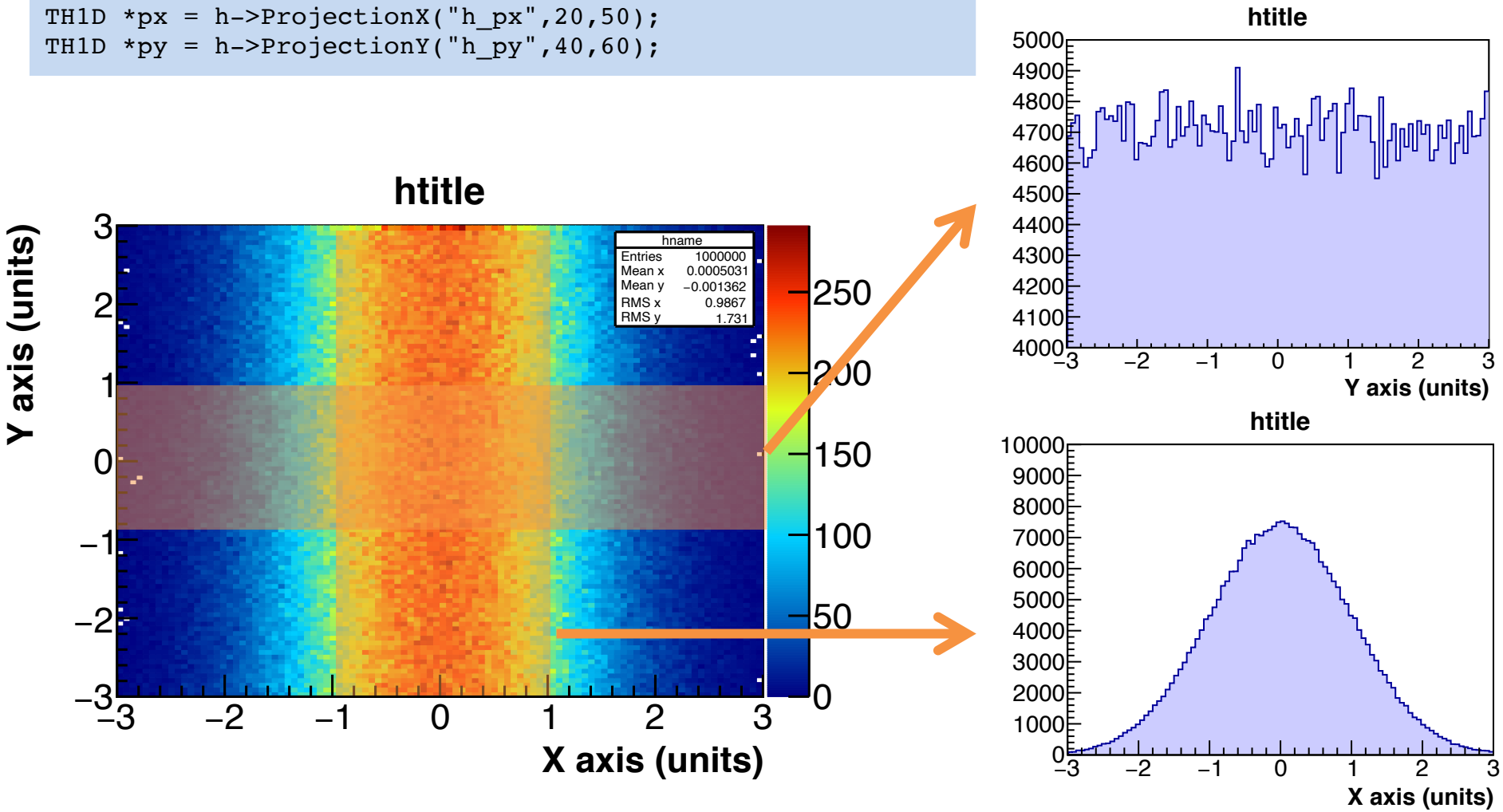
```
-Exp(tau)
-Integer(imax)
-Gaus(mean,sigma)
-Rndm()
-Uniform(x1)
-Landau(mpv,sigma)
-Poisson(mean)
-Binomial(ntot,prob)
```

The "start" of the sequence (called *seed*) is chosen by *TRandom::SetSeed(<int>)* or at the instantiat*ion new TRandom(<int>)*

Histograms

https://root.cern.ch/root/html534/TH2.html

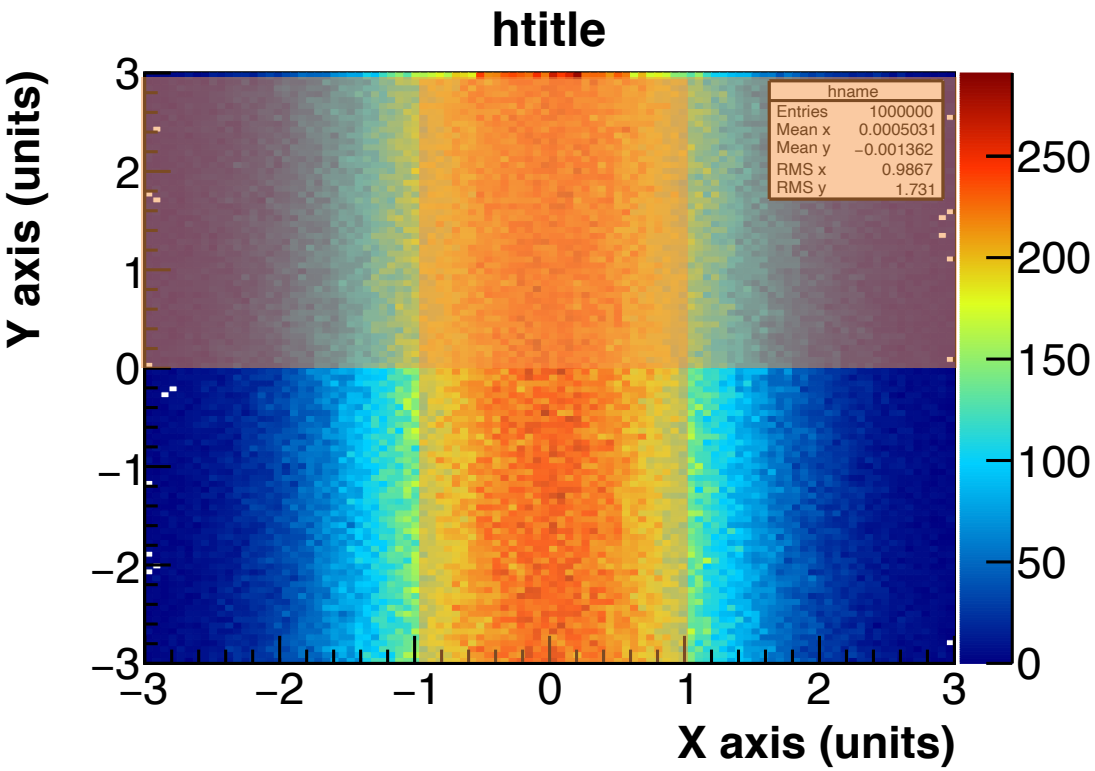
```
TH2F *h = new TH2F("hname","htitle;X axis (units);Y axis (units)",100,-3,3,100,-3,3);
for(int i=0; i<1000000;i++) h->Fill(gRandom->Gaus(0,1), gRandom->Uniform(-3,3));
h->Draw("COLZ");
TH1D *px = h->ProjectionX("h_px",20,50);
TH1D *py = h->ProjectionY("h_py",40,60);
```

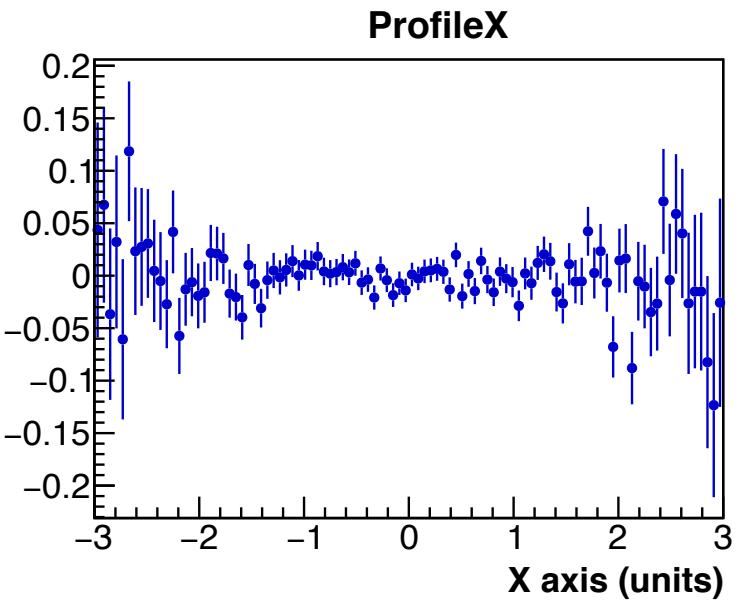


Profiles

https://root.cern.ch/root/html534/TProfile.html

```
TH2F *h = new TH2F("hname","htitle;X axis (units);Y axis (units)",100,-3,3,100,-3,3);
for(int i=0; i<1000000;i++) h->Fill(gRandom->Gaus(0,1), gRandom->Uniform(-3,3));
h->Draw("COLZ”);
TProfile *pfx = h->ProfileX("h_pfx",50,100);
pfx->Draw("");
```



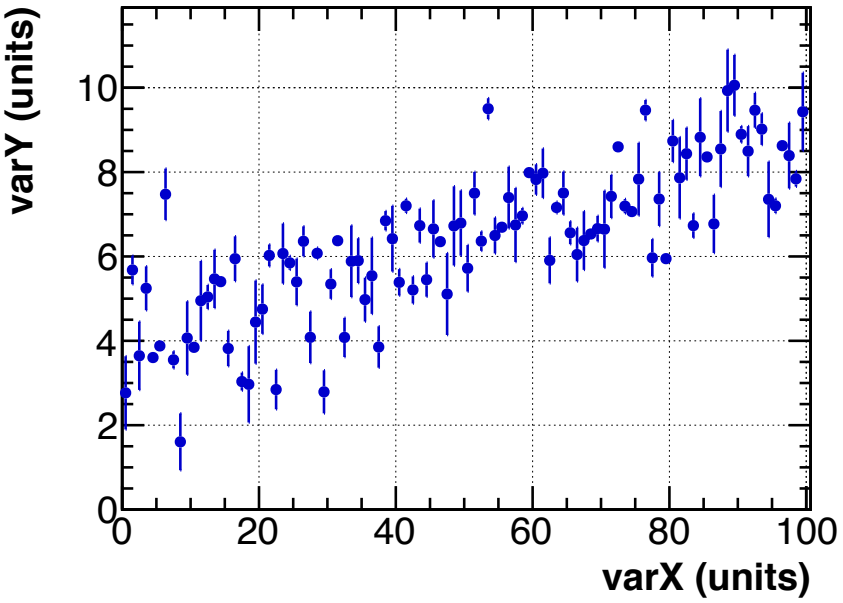


For each X bin, provides the mean and RMS of the Y var histogram

Graphs

- Graphs are the default representation for scatter plot and, in general, for data representing a relationship between values of Y and X

```
TFile *fin= TFile::Open("fin.root");
TGraphErrors *g = (TGraphErrors*)=fin->Get("gerrors");
g->Draw("AP");
for(int ip=0; ip<g->GetN(); ip++)
{cout<<g->GetX()[ip]<<" "<<g->GetY()[ip]<<" "<<
 g->GetErrorX(ip)<<" "<<g->GetErrorY(ip)<<endl; }
```




```
...
26.5 6.35923 0 0.354434
27.5 4.08275 0 0.609833
28.5 6.07419 0 0.134274
...
```

- NB: differently from histograms, Graphs point counting start from "0"

Graphs

- ROOT provides many classes to handle graphs
  - `TGraph` (no errors)
  - `TGraphErrors` (simmetric errors)
  - `TGraphAsymmErrors` (asimmetric errors)
  - `TGraphBentErrors` (titlted errors)
  - …..

- Use graphs to analyze and fit when searching for a Y(X) relation

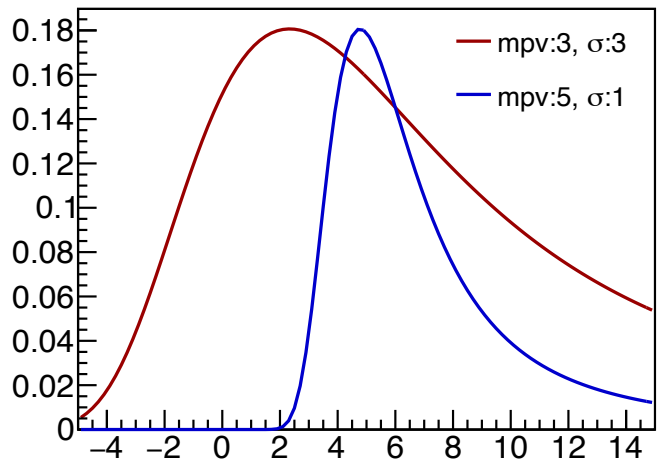
- Some useful methods
  - `TGraph::Print()` dumps the values stored in the X and Y arrays
  - `TGraph::Eval(Double_t x)` extrapolates the graph points into the `x` value (using a spline with desired order) and gives the y value at that x
  - `TGraphAsymmErrors::Divide( TH1* pass, TH1* total, Option_t* opt = "cp" )` divides two histograms (where pass is a subset of total) and computes efficiencies. Why asymmetric errors? (see theory)

Functions

https://root.cern.ch/root/html534/TF1.html

- ROOT provides a complete interface to mathematical functions

```
TF1 *flandau = new TF1("flandau","TMath::Landau(x,[0],[1])",-5,15);
flandau->SetParName(0,"MPV");
flandau->SetParName(1,"sigma");
flandau->SetLineColor(kRed+2);
flandau->SetParameters(3,3);
TCanvas *clandau = new TCanvas("clandau","clandau");
clandau->cd();
flandau->Draw("");
TF1 *flandau2 = (TF1*)flandau->Clone("flandau2");
flandau2->SetLineColor(kBlue-1);
flandau2->SetParameters(5,1);
flandau2->Draw("same");
TLegend *legend = new TLegend(0.6,0.7,0.84,0.89);
legend->AddEntry(flandau,"mpv:3, #sigma:3","L");
legend->AddEntry(flandau2,"mpv:5, #sigma:1","L");
legend->Draw("same");
```

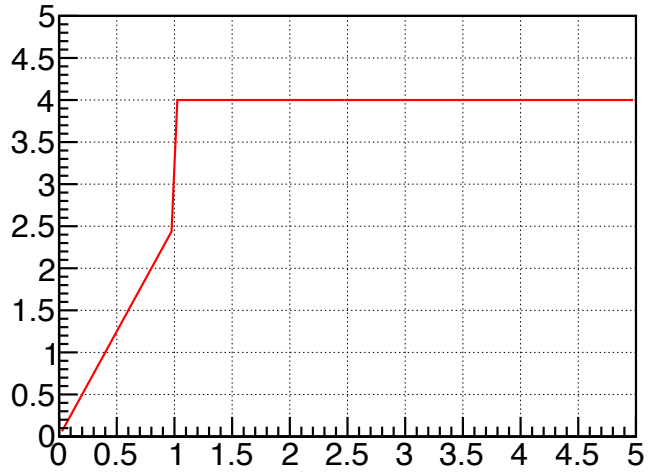


Functions

https://root.cern.ch/root/html534/TF1.html

- ROOT allows the possibility to use user-defined functions

```
Double_t stepf( Double_t *_xx, Double_t *par){
  Double_t x = _xx[0];
  return (x>par[0]) ? par[1] : x*par[2];
}

int main(){
  Int_t npar=3;
  TF1 *func = new TF1("func",stepf,0,5,npar);
  func->SetParameters(1,4,2.5);
  TCanvas *cfunc = new TCanvas("cfunc","cfunc");
  cfunc->cd()->SetGrid();
  func->Draw("");
  return 0;
}
```



Functions

`https://root.cern.ch/root/html534/TF1.html`

- Some useful methods
  - `TF1::Derivative(Double_t x)`
  - `TF1::Integral(Double_t xmin, Double_t xmax)`
  - `TF1::Eval(Double_t x)`
  - `TF1::GetRandom(Double_t xmin, Double_t xmax)`

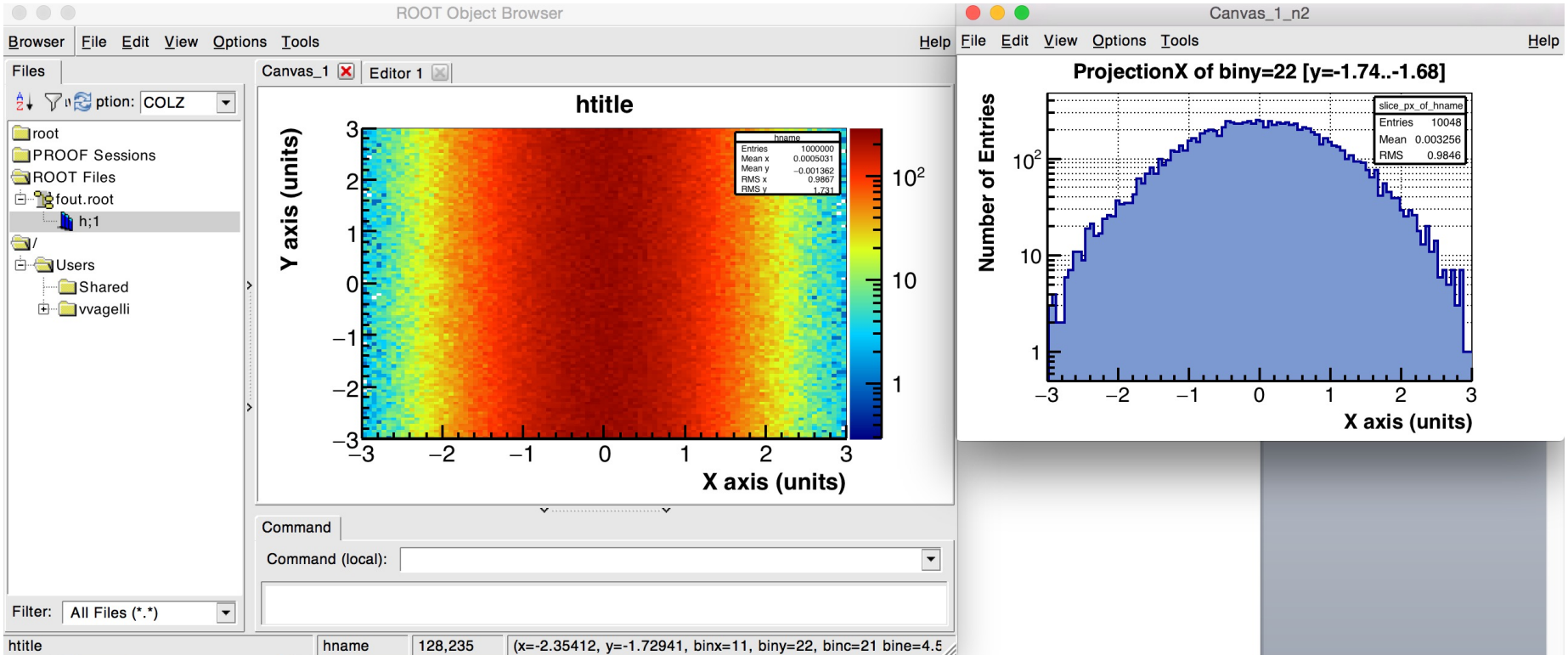
- The TMath class provides the basic mathematical functions. More complex tools are available in the MathCore and MathMore libraries (via `ROOT::Math` namespace), like `ROOT::Math::riemann_zeta(Double_t x)`

- Histograms and Graphs con be fitted with TF1 (this will be covered in a dedicated lecture)

GUI

- ROOT allows to browse files and edit plots using a Graphical User Interface
- Many actions (but not all) can be also performed via GUI. This gives an immediate impact of the action, but only small actions can be performed

```
vvagelli@Firefly~ $ root –l fout.root
root [0]
Attaching file fout.root as _file0...
root [1] new TBrowser
```



Trees

`https://root.cern.ch/root/html534/TTree.html`

- The `TTree` is the core class for (unbinned) data storage and analysis
- You can imagine a `TTree` as a smart database, where the data you collect are stored and later retrieved for analysis
- The concept is designed to store data in blocks of “data acquisition”
- The tree is the holder of your measurement
- The tree has entries, each one represents one set of measurement (time,temperature, pressure,... / velocity,mass,energy,charge,…)

Trees

https://root.cern.ch/root/html534/TTree.html

```
TFile *fin= TFile::Open("ntuples.root");
TTree *t = (TTree*)fin->Get("tree");
t->Print();
```

```
******************************************************************************
*Tree    :tree      : tree                                                   *
*Entries :   748657 : Total =      1093810927 bytes  File  Size =  610660092 *
*        :          : Tree compression factor =   1.79                       *
******************************************************************************
*Br    0 :Run       : Run/i                                                  *
*Entries :   748657 : Total  Size=    3020474 bytes  File Size  =     361472 *
*Baskets :      284 : Basket Size=      13824 bytes  Compression=   8.34     *
*.............................................................................*
*Br    1 :RunTag    : RunTag/i                                               *
*Entries :   748657 : Total  Size=    3021338 bytes  File Size  =      47037 *
*Baskets :      284 : Basket Size=      13824 bytes  Compression=  64.11     *
*.............................................................................*
*Br    2 :Event     : Event/i                                                *
*Entries :   748657 : Total  Size=    3021050 bytes  File Size  =    2390942 *
*Baskets :      284 : Basket Size=      13824 bytes  Compression=   1.26     *
*.............................................................................*
```

https://root.cern.ch/root/html534/TTree.html#TTree:Branch@3

Trees

https://root.cern.ch/root/html534/TTree.html

```
cout<<t->GetEntries()<<endl;
```

```
748657
```

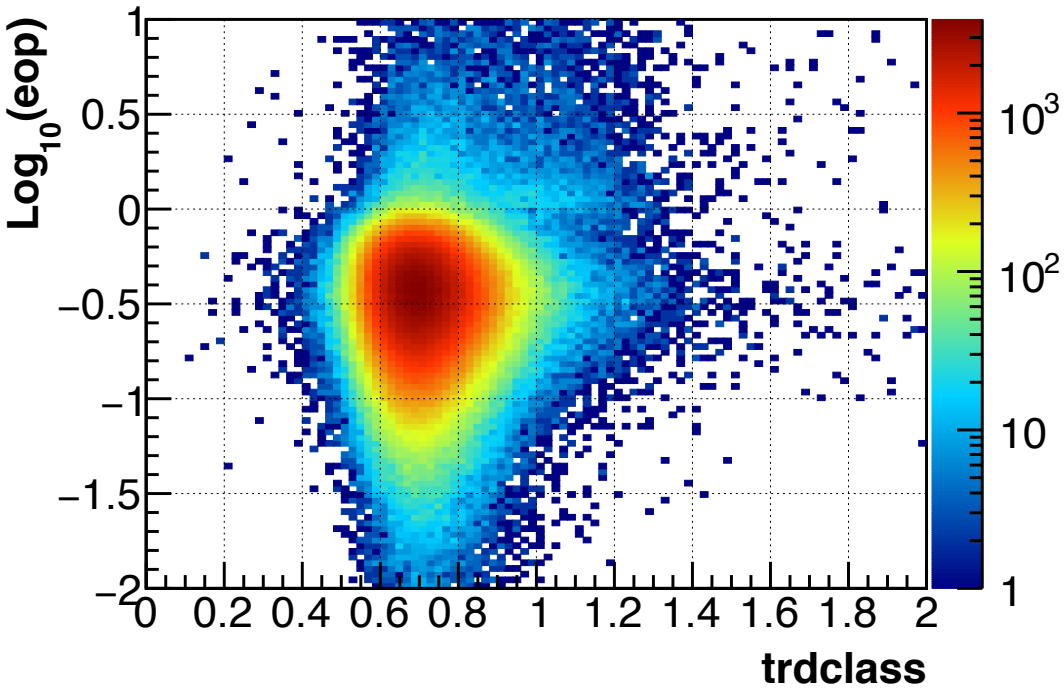
```
t->Show(5); //show the content of entry #5
```

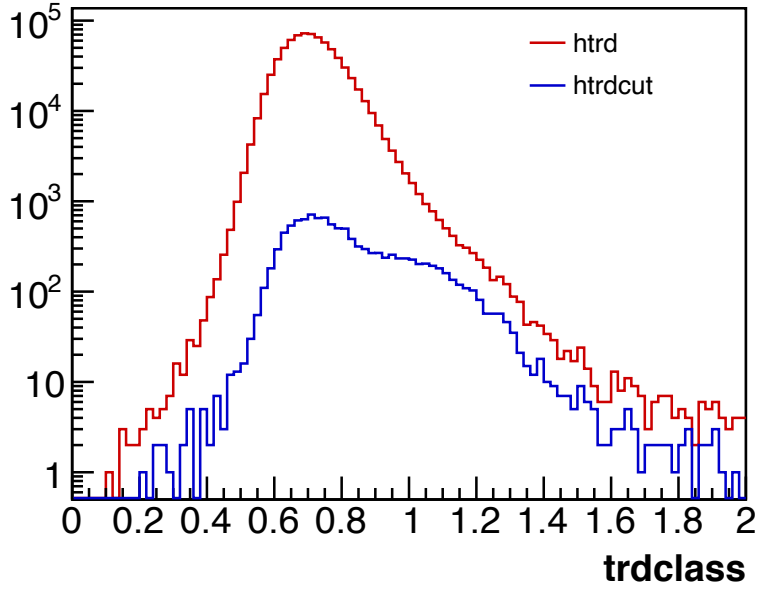
```
======> EVENT:5
 Run             = 1305859009
 RunTag          = 61440
 Event           = 67450
 Time            = 1305859236
 trkQlay         = 0,
                   0, 1.00414, 1.07635, 0.964231, 1.06249,
                   1.13795, 1.17315, 1.291
```

Trees

https://root.cern.ch/root/html534/TTree.html#TTree:Draw@2

```
TH1F *htrd = new TH1F("htrd",";trdclass",100,0,2);
tree->Draw("trdclass>>htrd");
TH2F *heoptrd = new TH2F("heoptrd",";trdclass;Log_{10}(eop)",100,0,2,100,-2,1);
tree->Draw("log10(eop):trdclass>>heoptrd","","COLZ");
TH1F *htrdcut = new TH1F("htrdcut",";trdclass",100,0,2);
tree->Draw("trdclass>>htrdcut","eop>1");
```





Trees

- The `TTree::Draw()` method is useful for dirty & quick checks
- A complete analysis of `TTree` data is usually done in an analysis macro
- The following code is equivalent to the previous example

```
{
  ...
  Float_t trdclass; //the type has to be the same as in the TTree branch
  Float_t eop;
  Double_t darray[20]; //for arrays, also its size must coincide
  //activate the TTree Branches
  tree->SetBranchAddress("trdclass", &trdclass);
  tree->SetBranchAddress("eop",      &eop);
  tree->SetBranchAddress("darray",   array);  // or &array[0]. Clear why?
  ...
  //Loop on TTree entries
  for(Int_t ientry=0; ientry<(Int_t)tree->GetEntries; ientry++)
  {
    tree->GetEntry(ientry); //variables are updated with entry stored values
    //Fill the histograms
    htrd->Fill(trdclass);
    htrdeop->Fill(trd, log10(eop));
    if( eop>1 ){ htrdcut->Fill(trdclass);
  }
...
}
```

Trees

- `TTrees` can be created using user data and stored in a `TFile` for future analyses.

```
{
  Float_t fvar;
  UShort_t usvar;
  Bool_t barray[10];

  TTree *fout = new TFile("fout.root","recreate");
  TTree *tout = new TTree("tname","ttitle");
  tout->Branch("fvar",   &fvar,   "fvar/F");
  tout->Branch("usvar",  &usvar,  "usvar/s");
  tout->Branch("barray", barray,  "barray[10]/0");

  for(Int_t imeasure=0; imeasure<Nmeasures; imeasure++)
  {
    /* Fill the variables with meaningful numbers */
    fvar = detector->GetContinuousValue();
    usvar = detector->GetDiscreteCounts();
    barray = detector->GetStatusArray();
    /* Save this entry in the tree */
    tout->Fill();
  }
  fout->cd();
  tout->Write();
  fout->Close();
}
```

Trees

- `TTree::ReadFile(const char *filename)` can be created used to create a tree using number saved previously in a text file

`stupidfile.txt`

```
aaa/I:bbb/F:ccc/C
0 3.4 who
999 -11.2 watches
-3 0.01 the
619 3.14 watchmen
```

```
{
  TFile *fout = new TFile("fout","recreate");
  fout->cd();
  TTree *tree = new TTree("tree");
  tree->ReadFile("stupidfile.txt")
  tree->Write();
  fout->Close();
}
```

Trees

```
#include "TFile.h"
#include "TTree.h"
#include <iostream>
#include <fstream>
using namespace std;

int filltree(const char* infilename, const char* outfilename){

TFile *fout = new TFile( outfilename,"recreate"); fout->cd();
TTree *tree = new TTree("treename","treetitle");
int aaa; float bbb; string ccc;     //declare tree variables
tree->Branch("aaa",&aaa,"aaa/I");
tree->Branch("bbb",&bbb,"bbb/F");
tree->Branch("ccc",&ccc,"ccc/C");

std::ifstream infile( infilename ); //open file to read
infile>>ccc; //skip first line
while( !infile.eof()){              //read the file
infile >> aaa >> bbb >> ccc;        //store the file content into variables
printf("%d\t%f\t%s\n",aaa,bbb,ccc.c_str());
tree->Fill(); //store the values into the tree. One entry is saved
}

tree->Write(); //Write the tree to file
fout->Close(); //close and saves the output file
return 0;
}
```

Trees

```
vvagelli@Firefly~ $ root outfile.root
Lybraries loaded
Style set
root [0]
Attaching file outfile.root as _file0...
root [1] treename->GetEntries()
(const Long64_t)5
root [2] treename->Show(0)
======> EVENT:0
 aaa               = 0
 bbb               = 3.4
 ccc               = who
root [3] treename->Show(2)
======> EVENT:2
 aaa               = -3
 bbb               = 0.01
 ccc               = the
```

TChain

https://root.cern.ch/root/html534/TChain.html

- TTRees saved in different files, but the analysis has to be run on the whole dataset
- The TChain class can be used to chain the files, and it can be treated as a unique TTree spanning all the files

```
{
  TChain *chain = new TChain("tree");
  chain->Add("/some/dir/*.root");
  chain->GetListOfFiles()->Print();
```

```
Collection name='TObjArray', class='TObjArray', size=100
 OBJ: TChainElement   tree/some/dir/file1.root
 OBJ: TChainElement   tree/some/dir/file1.root
 OBJ: TChainElement   tree/some/dir/file2.root
```

```
  Float_t fvar; //the type has to be the same as in the TTree branch
  chain->SetBranchAddress("fvar", &fvar);
  for(Int_t ientry=0; ientry<(Int_t)chain->GetEntries; ientry++)
  {
    chain->GetEntry(ientry); //variables are updated with entry stored values
    //Fill the histograms
    hfvar->Fill(TMath::ACos(fvar));
  }
}
```

Algebra and Physics Tools

- Many tools provided by ROOT to solve algebra and physics problems.
- A selection of useful classes:
  - `TString` to handle character strings
  - `TVector` and `TMatrix`, to handle and solve numerical linear algebra problems
  - `TLorentzVector`, to describe Lorentz transformations
  - `TRandom`, to generate random numbers
  - `TSpectrum`, to analyze and process spectra
  - `TEfficiency`, to calculate efficiencies and their uncertainties
  - `TSpline`, for non-parametric extrapolations
  - `TMVA` (Toolkit for Multivariate Analysis) classes, for multivariate analysis of big data samples
  - `RooFit` classes, for advanced data fitting