

## C++ programming

### Lecturers:

- **Eric Chabert**
- **Eric Conte**

### Program:

- 6 hours of lectures (& tutorials)
- 4 computing sessions (3h each) with an introduction to the use of



# Goals of that course

The program has been adjusted to your level

It is the first version so it will clearly not be perfect ...

.. **but** we can interact so don't hesitate to stop me & fill free to ask questions

## Goals (within the limitation of 6 hours)

```
while( !is_understood(what_I_said)){
    cout<<explanation<<endl;
    explanation+=adjustment;
}
```

- (Re)inforce your knowledge & **understanding** of the basis
- Give you examples of applications
- Highlight “not well known enough” features of C++
- Give you guidance for your current & future developments
- Discuss more advanced functionalities

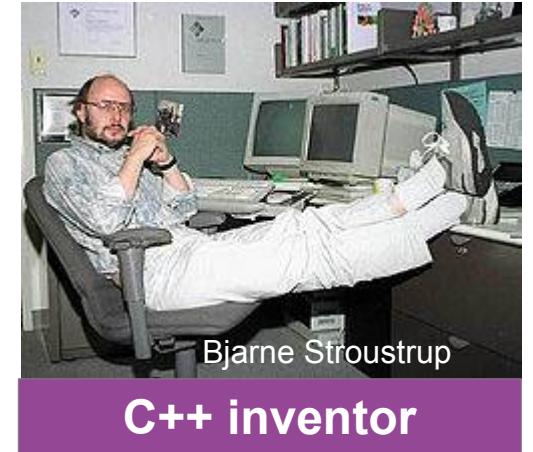
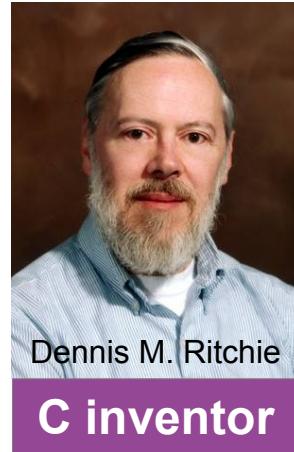
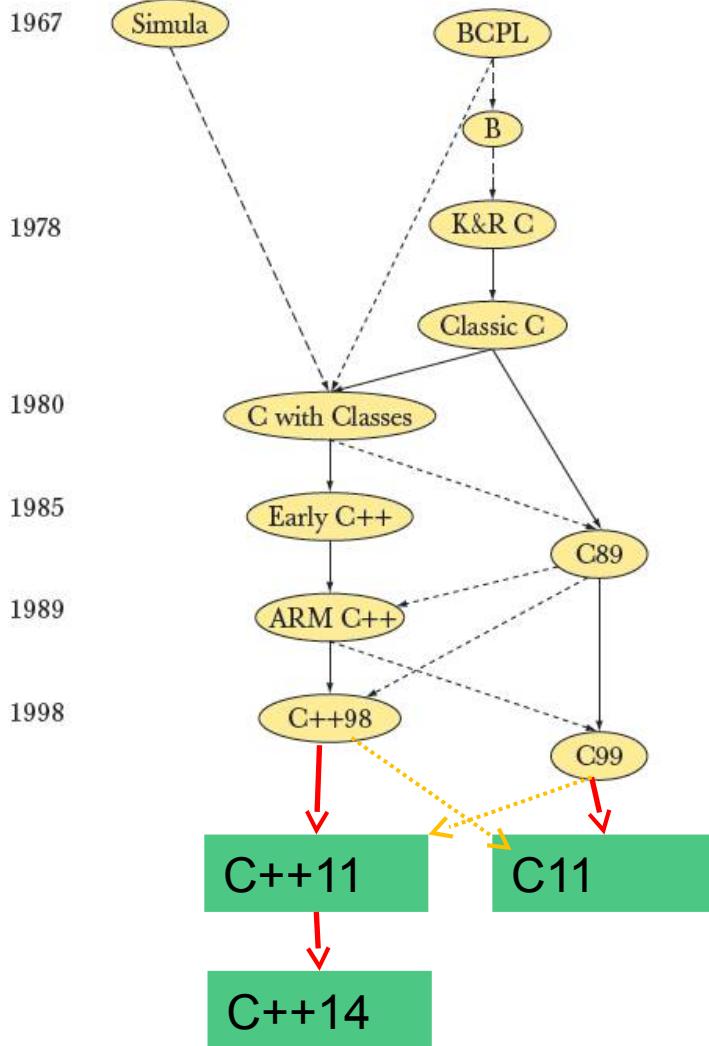
- ✓ *Everything will not be covered*
- ✓ *No formal lectures on ROOT or GEANT4 here*
- ✓ *It is not an advanced lecture and will not become an C++ expert*

You're following a **beginner/intermediate condensed** lecture

# Why C++ ?

- We are looking for **scientific application** (use of numerical methods,...) and we want program to run “*fast*”
  - It cannot be an interpreted language (ex: python), but a ***compiled one***
- We have to deal with a **complex environment** and to perform well advanced tasks
  - It must be an ***oriented object*** language (ex: java, ...)
- We need a language for which *tools* already exists
  - It must have libraries (standard or not)
    - **C++ is the (*one*) answer !**
- Most of HEP collaborations use C++ for their software developments
- C++ is precisely defined by an ISO standard and is available on all OS
- *Programming concepts that you will learn using C++ can be used in other languages*

# C++: a bit of history



- Both C & C++ were “born” in the Bell Labs
- C++ *almost* embed the C
- Many tasks could be done in a C/style or C++/style
- Some C/style “procedures” are be to proscribed
- C++ has a long history and is still in development
- C++ is less “modern” than java (91), python(93), C#(2000) ...
- We will **here** discuss about **C++98** (not **C++11**)

# Writing a program requires many steps

- **Preparatory work**

- ✚ Modelisation of the problem
  - ✚ Identification of the algorithms or tools to be used (does appropriate libraries exist ?)
  - ✚ Defining the specifications
  - ✚ Project management: task division/sharing ...

- **Writing the code**

- ✚ This is not the most time consuming tasks ....

- **Compilation**

- ✚ From simple one to more complex (use of Makefile)
  - ✚ Debugging (could be time consuming)

- **Test**

- ✚ Test of every part & functionality of the program
  - ✚ Verification of the code protection (Crash can happen during runtime. Unexpected behavior ...)

- **Optimization [optional]**

- ✚ Could be done with respect to different quantity: cpu time, memory usage, desired precision, ...

- **Utilization**

- ✚ Private/Restrictive/Public usage ? ... feedback to come ...

# What is the language made of ?

- Types (*bool, int, float, char,...*)
- Expression and statements
- Selection (if/else, switch,...)
- Iteration (while,for,...)
- Functions (“*intrinsic*” or *user-defined*)
  - Accessible via libraries
- Containers (vector, map,...)
  - Accessible via libraries

With those ingredients, you can do a lot of things ....



## V65XX Controller

Control Software for VME Power Supply Modules



- Console Program
- Supports Linux and Windows
- full control of a single VME High Voltage V65XX Board

...

# "Hello world" example

Header files  
Preprocessor directive

Only one "main"  
function per executable  
Return an **integer** that can be used  
by the system

main.cpp

```
#include <iostream>

int main() {
    float x; //declaration
    int i=3; //declaration and affectation
    std::cout << "Hello world" << std::endl;
    std::cout << "i=" << i << std::endl;
    return 0;
}
```

Type  
Variable declaration  
Variable affectation

Input/Output

Terminal

Namespace

```
echabert@sbgat603:~$ g++ -o main.exe main.cpp
echabert@sbgat603:~$ ./main.exe
Hello world
i=3
```

It is already a "rich" example !!

# Variable and types

## Several types could be accessible

- Build-in types:  
Ex: bool, int, float, double, char
- Standard library types:  
Ex: complex, string, ...
- Specific libraries:  
Ex: (Root) Float\_t, TString, TH1F
- User defined types:  
your own classes



**On the machine, everything is only bits filled with 0/1**

**“Type” is what interprets bits and give them meaning !**

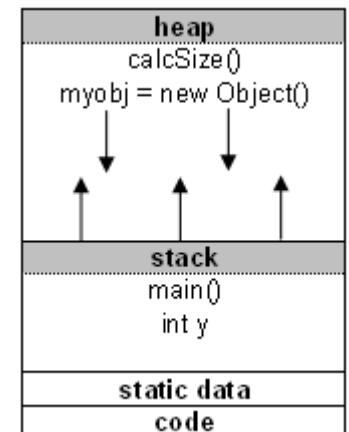
# build-in type

Type representation (number of bits used) depends on the platform

Ex: on my computer (iCore7, 64 bits)

type	content	size	range
bool	True (0) ou False (1)	8 bits	
short	Signed integer	16 bits	[-32768,32767]
int	Signed integer	32 bits	[-2147483648,2147483647]
long	Signed integer	64 bits	[-9223372036854775808,92233720368547]
float	floating-point	32 bits	de $1.4E^{-45}$ à $3.4E^{+38}$
double	floating-point	64 bits	de $4.9E^{-324}$ à $1.8E^{+308}$
char	ASCII char	8 bits	[0,255]

- Sign uses 1 bit – “unsigned” type have double possible value
- Once you “declare” a variable of a given type, you allocate memory
- Build-in type goes on the **stack**
  - fast access
  - available during the whole existence of the program



# Standard library types

- **String**
  - ↳ “Extension” of character chains
  - ↳ Discussed later in the course
- **complex<Scalar>**
  - ↳ `complex<double>`
  - ↳ `complex<float>`
  - ↳ .. it's an example of “template class”

**Headers:**

```
#include <string>
#include <complex>
```

# Types defined in other libraries



- `int_least8_t`
- `int_least16_t`
- `int_least32_t`
- `uint_least8_t`
- `uint_least16_t`
- `uint_least32_t`
- ...



- `Int_t`
- `UInt_t`
- `Double_t`
- `Double32_t` ....

**Headers:**  
`#include <boost/cstdint.hpp>`

Will ensure the number of bits used on the machine (portable)

**Headers:**  
`#include <Rtypes.h>`

(basic) types can be (re)defined by specific library

# Comments & documentation

- To comment the end of a line: //

*Instruction; // Here starts the comment*

- To comment a block of lines: /\* block \*/

```
Instructions;  
/* The following lines are inactive  
for(int i=0;i<10;i++){  
    i = i*10;  
}  
*/
```

- Comments are **really useful**

- Comment what variables represents (*names are not always sufficient*)

```
TVector3 fP; // 3 vector component  
Double_t fE; // time or energy of (x,y,z,t) or (px,py,pz,e)
```

- Comment functional block

- Ex: reading input, computing a sum, writing an output, ...

- Comment the program, the functions (.h), the classes (.h)

- Explain the goals, the input, the output, the main algo ...

**Commenting is not a lost of time.**

**It will be useful for *you* already few weeks after coding  
but also for your *co-developers* or future *users* of your code !!!**

**Tools for documentation formatting exists, ex: doxygen**

# Usage of variables

- **Declaration**

- **required**
- Precise the type of the variable

```
int i;
```

- **Initialization**

- **Strongly recommended**
- Can lead to unexpected behavior otherwise
- Declaration & Initialization can be done at once

```
i=0;  
int j=10;
```

```
Float x = 103.4;  
Float y = 1.034e2; //e ou E
```

- **Affectation**

```
i=j; //affectation  
i = i*2+1;  
i=j**2+3*j;
```

```
int i=23;  
short b = (short) i; //C-like  
short b = i; //C-like  
Short b = static_cast<short>(i); //C++ like  
i= (int) 10.6; // will be truncated  
Float f = 10.6;  
i = static_cast<float>(f); //C++ like
```

- **Operations**

- **Conversion**

- Implicit (explicit)
- Truncated numbers
- Other features ...

```
float a = 3.2;  
int i = 1/a;
```

# Declaration and initialization (II)

```
int a;
int a = 7;

bool b = true; // other literal: false

char c = 'c'; //could be also special characters: ., $, ...

// 3 example to declare and initialize a float
float f1 = 1234.567;
float f2 = 1.234567E3; // scientific notation - could be e or E
float f3 = 1234.567F; // f or F specify that it is a float

string s0;
string s1 = "Hello, world";
string s2 = "1.2";

complex<double> z(1.0,2.0);
```

## Declaration:

- introduce a **name** into a **scope**
- specify a **type** for named object
- sometimes it includes an **initialization**
- a name must always be declared before being used (compilation error otherwise)

## Initialization:

- Syntax depends on the type (see examples above)

# Variables: operations

- Arithmetic operations
- Affectation
- Comparison operations
- Boolean operations
- Pre and post in(de)crement

Arithmetic operation		Arithmetic/Affectation	
+	addition	+=	add
-	soustraction	-=	subtract
*	multiplication	*=	multiply
/	division	/=	divide
%	modulo	%=	modulo
-(unaire)	opposed		

In/De crement	
i++	post-increment
i--	post-decrement
++i	pre-increment
--i	pre-decrement

Comparison operators	
==	equality
!=	difference
> ; >=	Greater than(or equal)
< ; <=	Lower than (or equal)

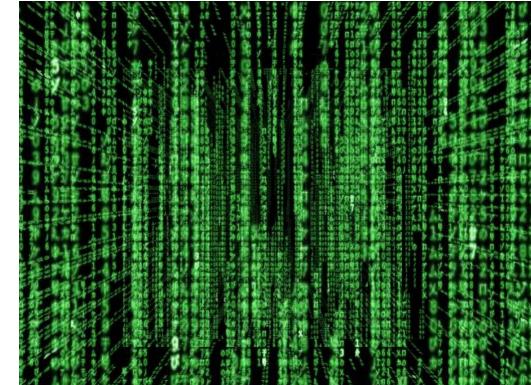
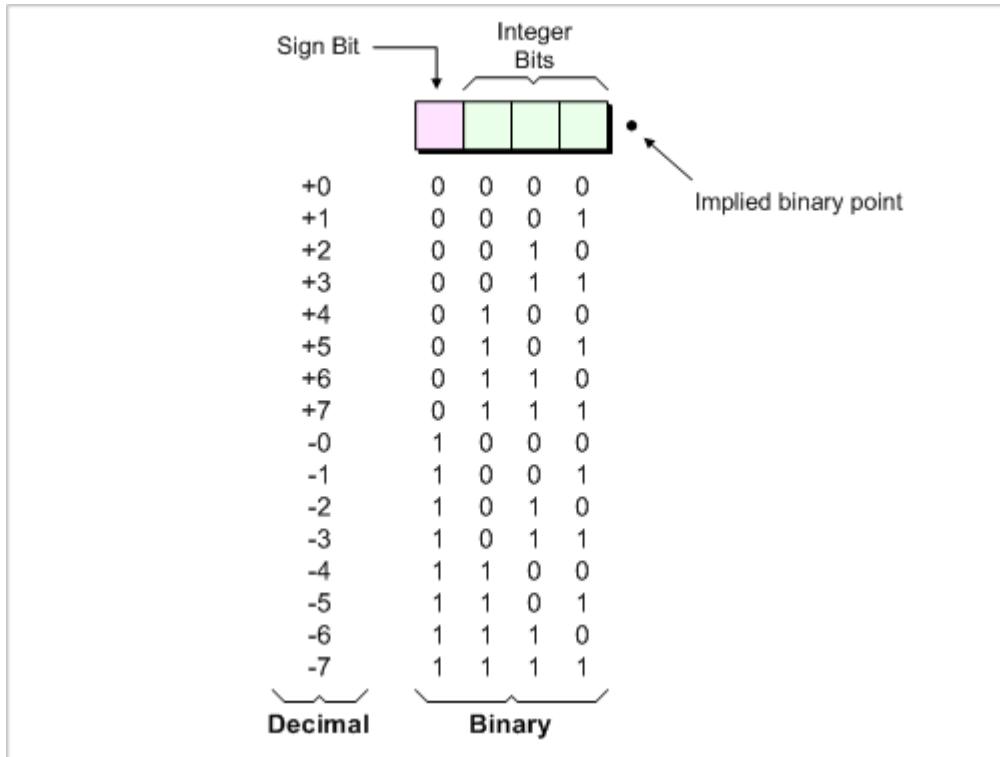
4 equivalent incrementation:

```
a=a+1;  
a+=1;  
a++;  
++a;
```

“Concise operators” are generally better to use

```
a+=c    ↔  a=a+c  
a*=scale  ↔  a=a*scale
```

# Integer representation



**int** - max value =  $2^{31} - 1$   
**unsigned int** - max value =  $2^{32} - 1$

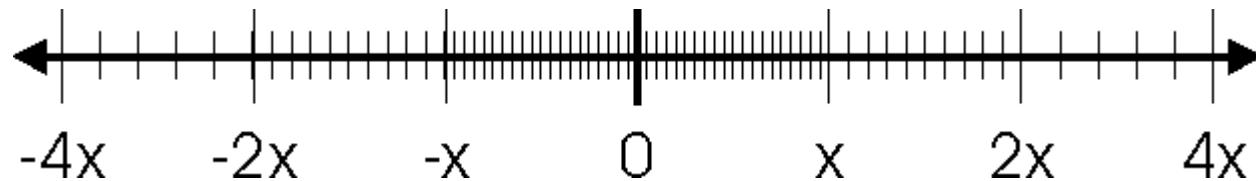
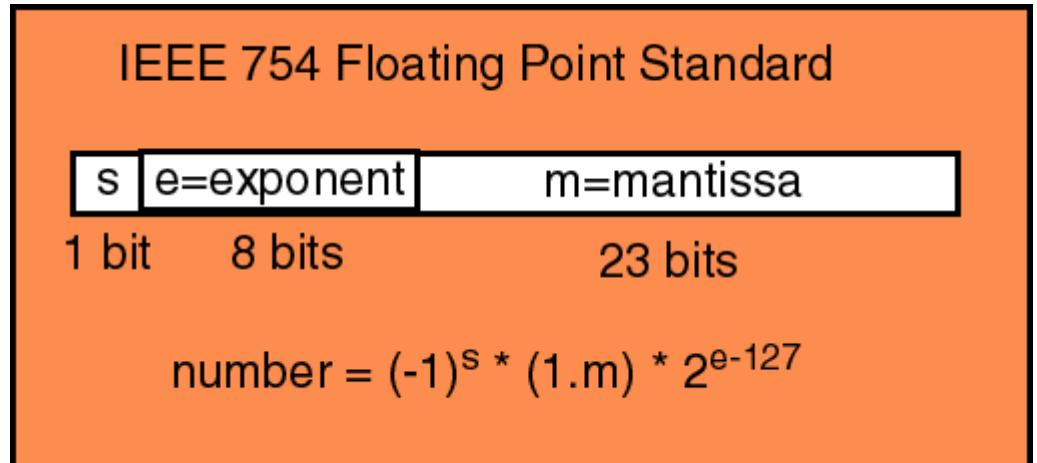
```
int a = 4; // coded ...000100
a=a<<3; // coded ...100000
cout<<"a="<<a<<endl;
a=32
```

**Bitwise operators**  
*not only applicable to integers*

&	AND
	OR
^	XOR (exclusive or)
~	NOT – inversion of the bit
<<	Left shift
>>	Right shift

# Floating Point representation

- Real (float & double) are actually mainly represented using floating point representation following the norm IEEE-754.
- Representation:  $(-1)^s \times m \times b^{e-E}$ 
  - S: sign
  - M: mantissa
  - B: base
  - E: exponent
- Reals are obviously discretely represented on computers
- Absolute precision evolves with the value of the variable



# Expressions

## Boolean type expression

<b>Equality operators</b>	<code>==</code>	equal
	<code>!=</code>	Not equal
<b>Logical operators</b>	<code>&amp;&amp;</code>	and
	<code>  </code>	or
	<code>!</code>	not
<b>Relation operators</b>	<code>&lt;</code>	Less than
	<code>&lt;=</code>	Less than or equal
	<code>&gt;</code>	Greater than
	<code>&gt;=</code>	Greater than or equal

```
bool debug_mode = false;
cout<<"Do you want to debug ?"<<endl;
cin>>debug_mode;

for(int i=0;i<1000;i++)
    for(int j=0;j<1000;j++)
        for(int k=0;k<1000;k++){
            result+=i*j+k; // some stupid formula
            if( debug_mode && ( (i>=j && j==k) || (i+j>100) ) ) {
                cout<<"Some stupide message !"<<endl;
            }
        }
```

All those expressions can be combined also with the help of ()

# Precision & numerical uncertainty

- Representation
  - The value you could want to represent might not be represented (*approximation*)

```
Float a = 1;  
Float b = 3;  
Float c = a/b  
cout<<"c =" <<c <<endl;  
c=3.3333333333333315e-01
```

- Truncation
  - The result of a computation involving two well defined represented numbers can lead to a truncated number

- “Reduced” variable (close to 1)

- This is equivalent to performing a change of variable with dimensionless variable
- Subtraction of two variables having big difference will lead to a high uncertainty

- Expressions being analytically equal will not necessarily give the same numerical results

- First step before implementing a formula is to choose the the **better** expression (lowest uncert.)

```
float a = 1.23456789;  
float b = 9.87654321;  
  
float c = (a*a-b*b)/(a-b);  
float d = a+b;  
cout.precision(20);  
cout<<"c = " <<c <<endl;  
cout<<"d = " <<d <<endl;
```

$(a^2-b^2)/(a-b) \neq (a+b)$  ?

c = 11.111111640930175781  
d = 11.111110687255859375

# Precision & numerical uncertainty

## Precautions & tests:

- **Division by zero:**
  - ★ will lead to a crash – test the denominator first
- **Division of integer**
  - ★ Ex: float a = 1/3; // a = 0 !! - at least numerator or denominator should be a float
- **Equality test of reals**
  - ★ It might be better to test a small difference  $\epsilon$  between the two variables (truncature pb)

## Stl offers tools to perform test on numbers

- **Isinf** // test for infinite
- **isnan** // NAN = Not A Number
  - // all combinations of bits doesn't represent a number (float/double)
- ...

# Simple examples

```
float x = 1.0/333;  
float sum = 0;  
for(int i=0;i<333;++i)  
    sum+=x;  
cout<<sum<<endl;
```

0.999999

```
short y = 40000;  
int i = 1000000;  
cout<<y<<" "<<i*i<<endl;
```

-25536 -727379968

Floating-point numbers are approximation of real numbers

- Can lead to **numerical errors** (quantification?)

Integer types represent integer up to a certain limit

- **Overflow problem**

Integer and real numbers are infinite while the number of bits to represent is definitively finite !  
Remember this while applying numerical methods

```
float f0 = 5.7;  
int j = f0;  
cout<<f0<<" "<<j<<endl;
```

5.7 5

Implicit conversion float to integer  
→ **truncature problem**

Lost of precision for large integer values  
→ **troncature problem**

```
float f1 = 0;  
long k=123456789123456789;  
f1=k;  
cout.precision(12);  
cout<<f1<<" "<<k<<endl;
```

1.23456790519e+17 123456789123456789

# Type-safety violation

```
int a = 42928;
char c = a;
int b = c;
cout<<a<<" "<<b<<" "<<c<<endl;
```

42928 -80 ↴

```
char c = 'a';
int i = c;
cout<<"char: "<<c<<" integer: "<<i<<endl;
```

char: a integer: 97

C++ doesn't prevent you from trying to put a large value into a small variable (though a compiler may warn)  
→ **Implicit narrowing**

In memory, everything is just bits: 01100001  
Type is what gives meaning to bits:  
01100001 is the char 'a'  
01100001 is the integer 97

```
int x;           // gets a "random" initial value
char c;          // gets a "random" initial value
double d;        // gets a "random" initial value
                // not every bit pattern is valid floating-point value
double dd = d;  // potential error: some implementation
                // can't copy invalid floating-point value
cout<<" x:"<<x<<" c:"<<c<<" d:"<<d<<endl;
```

x:4196320 c: d:6.95315e-310

- **Always initialize your variable !!**
- **Valid exception:** input variable

# STL: Standard Template Library

- C++ offers a very useful library than can be used: STL
- It offers solutions in various aspects:
  - ✚ Defining containers
  - ✚ Providing algorithms
  - ✚ Input/Output
  - ✚ More details later in the course
- Most of the “tools” (variables, functions, classes,...) are defined in the **namespace std**
- To give access to those functionalities, one need to include file
  - ✚ Ex: #include <iostream>
- To use it, one need to specify the namespace
  - ✚ Ex: std::cout
  - ✚ Or using **namespace std**; and then **cout** (no need to precise the namespace)

# Numerics: standard functions

There are already a lot of “tools” in the STL that can help you in your implementation (and tests)

Headers:  
#include <cmath>

## Classification macro / functions

**fpclassify** Classify floating-point value (macro/function )

**isfinite** Is finite value (macro )

**isinf** Is infinity (macro/function )

**isnan** Is Not-A-Number (macro/function )

**isnormal** Is normal (macro/function )

**signbit** Sign bit (macro/function )

## Minimum, maximum, difference functions

**fdim** Positive difference (function )

**fmax** Maximum value (function )

**fmin** Minimum value (function )

## Other functions

**fabs** Compute absolute value (function )

**abs** Compute absolute value (function )

**fma** C++ Multiply-add (function )

## Rounding and remainder functions

**ceil** Round up value (function )

**floor** Round down value (function )

**fmod** Compute remainder of division (function )

**trunc** C++ Truncate value (function )

**round** C++ Round to nearest (function )

**lround** C++ Round to nearest and cast to long integer (function )

**llround** C++ Round to nearest and cast to long long integer (function )

**rint** C++ Round to integral value (function )

**lrint** C++ Round and cast to long integer (function )

**llrint** C++ Round and cast to long long integer (function )

**nearbyint** C++ Round to nearby integral value (function )

**remainder** C++ Compute remainder (IEC 60559) (function )

**remquo** C++ Compute remainder and quotient (function )

## Floating-point manipulation functions

**copysign** Copy sign (function )

**NAN** Not-A-Number (constant )

**nextafter** Next representable value (function )

**nexttoward** Next representable value toward precise value (function )

From <http://www.cplusplus.com/reference/cmath/>

# Mathematical libraries: standard functions

## Trigonometric functions

<a href="#">cos</a>	Compute cosine ( <a href="#">function</a> )
<a href="#">sin</a>	Compute sine ( <a href="#">function</a> )
<a href="#">tan</a>	Compute tangent ( <a href="#">function</a> )
<a href="#">acos</a>	Compute arc cosine ( <a href="#">function</a> )
<a href="#">asin</a>	Compute arc sine ( <a href="#">function</a> )
<a href="#">atan</a>	Compute arc tangent ( <a href="#">function</a> )
<a href="#">atan2</a>	Compute arc tangent with two parameters ( <a href="#">function</a> )

## Hyperbolic functions

<a href="#">cosh</a>	Compute hyperbolic cosine ( <a href="#">function</a> )
<a href="#">sinh</a>	Compute hyperbolic sine ( <a href="#">function</a> )
<a href="#">tanh</a>	Compute hyperbolic tangent ( <a href="#">function</a> )
<a href="#">acosh</a> C++11	Compute arc hyperbolic cosine ( <a href="#">function</a> )
<a href="#">asinh</a> C++11	Compute arc hyperbolic sine ( <a href="#">function</a> )
<a href="#">atanh</a> C++11	Compute arc hyperbolic tangent ( <a href="#">function</a> )

You've certainly already used some of them.  
Other are less well known but might be useful for you in a future project  
...

**Headers:**  
`#include <cmath>`

## Exponential and logarithmic functions

<a href="#">exp</a>	Compute exponential function ( <a href="#">function</a> )
<a href="#">frexp</a>	Get significand and exponent ( <a href="#">function</a> )
<a href="#">ldexp</a>	Generate value from significand and exponent ( <a href="#">function</a> )
<a href="#">log</a>	Compute natural logarithm ( <a href="#">function</a> )
<a href="#">log10</a>	Compute common logarithm ( <a href="#">function</a> )
<a href="#">modf</a>	Break into fractional and integral parts ( <a href="#">function</a> )
<a href="#">exp2</a> C++11	Compute binary exponential function ( <a href="#">function</a> )
<a href="#">expm1</a> C++11	Compute exponential minus one ( <a href="#">function</a> )
<a href="#">ilogb</a> C++11	Integer binary logarithm ( <a href="#">function</a> )
<a href="#">log1p</a> C++11	Compute logarithm plus one ( <a href="#">function</a> )
<a href="#">log2</a> C++11	Compute binary logarithm ( <a href="#">function</a> )
<a href="#">logb</a> C++11	Compute floating-point base logarithm ( <a href="#">function</a> )
<a href="#">scalbn</a> C++11	Scale significand using floating-point base exponent ( <a href="#">function</a> )
<a href="#">scalbln</a> C++11	Scale significand using floating-point base exponent (long) ( <a href="#">function</a> )

## Power functions

<a href="#">pow</a>	Raise to power ( <a href="#">function</a> )
<a href="#">sqrt</a>	Compute square root ( <a href="#">function</a> )
<a href="#">cbrt</a> C++11	Compute cubic root ( <a href="#">function</a> )
<a href="#">hypot</a> C++11	Compute hypotenuse ( <a href="#">function</a> )

# Do not reinvent the wheel !

*A lot of things are already available in the stl*

## C library:

- <cassert> (assert.h)
- <cctype> (ctype.h)
- <cerrno> (errno.h)
- <cfenv> (fenv.h)
- <cfloat> (float.h)
- <cinttypes> (inttypes.h)
- <ciso646> (iso646.h)
- <climits> (limits.h)
- <clocale> (locale.h)
- <cmath> (math.h)
- <csetjmp> (setjmp.h)
- <csignal> (signal.h)
- <cstdarg> (stdarg.h)
- <cstdbool> (stdbool.h)
- <cstddef> (stddef.h)
- <cstdint> (stdint.h)
- <cstdio> (stdio.h)
- <cstdlib> (stdlib.h)
- <cstring> (string.h)
- <ctgmath> (tgmath.h)
- <ctime> (time.h)
- <cuchar> (uchar.h)
- <cwchar> (wchar.h)
- <cwctype> (wctype.h)

## Comparison macro / functions

<b>isgreater</b>	Is greater (macro )
<b>isgreaterequal</b>	Is greater or equal (macro )
<b>isless</b>	Is less (macro )
<b>islessequal</b>	Is less or equal (macro )
<b>islessgreater</b>	Is less or greater (macro )
<b>isunordered</b>	Is unordered (macro )



- Containers:**
  - <array>
  - <deque>
  - <forward\_list>
  - <list>
  - <map>
  - <queue>
  - <set>
  - <stack>
  - <unordered\_map>
  - <unordered\_set>
  - <vector>
- Input/Output:**
  - <fstream>
  - <iomanip>
  - <ios>
  - <iosfwd>
  - <iostream>
  - <istream>
  - <ostream>
  - <sstream>
  - <streambuf>

## Multi-threading:

## Other:

- <algorithm>
- <bitset>
- <chrono>
- <codecvt>
- <complex>
- <exception>
- <functional>
- <initializer\_list>
- <iterator>
- <limits>
- <locale>
- <memory>
- <new>
- <numeric>
- <random>
- <ratio>
- <regex>
- <stdexcept>
- <string>
- <system\_error>
- <tuple>
- <typeindex>
- <typeinfo>
- <type\_traits>
- <utility>
- <valarray>

# A simple example

Let's consider the problem of looking to the smallest element of a std::vector

```
void f(const vector<int>& vc)
{
    // pedestrian (and has a bug):
    int smallest1 = v[0];
    for (int i = 1; i < vc.size(); ++i) if (v[i] < smallest1) smallest1 = v[i];

    // better:
    int smallest2 = numeric_limits<int>::max();
    for (int i = 0; i < vc.size(); ++i) if (v[i] < smallest2) smallest2 = v[i];

    // or use standard library:
    vector<int>::iterator p = min_element(vc.begin() ,vc.end());
    // and check for p==vc.end()
}
```

#include <vector>

#include <limits>

#include <algorithm>

A lot of “common problems” have been treated and implemented by more experimented C++ developer than you:

Why won't we use their tools ?

Once you have a project, first check on the existing tools (lib) if a solution have been already developed.

If yes, it will let you know time to concentrate on the specificity of your current project and also time to analyze your results !

# Const variables

It is not a good idea to have “magic numbers”, “hardcoded values”.

When reviewing your codes, you should change them (better to be done at first implementation)

## Many possibilities:

- The value is a *parameter*:

- user can change it (cin, file, ...)

```
Int nof_channels = 0;  
cout<<"Enter the number of channel:"<<endl;  
cin>>nof_channels;
```

- The value is redefining by a macro alias:

```
#define NOF_CHANNELS 12
```

- The value can be a constant !

```
const int nof_channels = 12;
```

- Initialization should come with definition
  - The value is protected and could not be changed later on the program
  - Attempts to change the value will lead to compilation error

It is useful to define as **const** variables many kind of variables:

- mathematical/physical constants:  $\pi, G, \epsilon_0$
  - constants variables of your software: *number of channels*, ...

It helps for the **meaning**: 12 doesn't mean anything while nof\_channels does !

It avoid numerical **problems**: having dependent on the number of digits:  $3.14!=3.14159265$

# Static variable

- **Static variables** keep their values and are not destroyed even after they go out of scope
  - ↳ Can be useful for incrementation by example

```
int GenerateID()
{
    static int s_nID = 0;
    return s_nID++;
}

int main()
{
    //cout<<"s_nID = "<<s_nID<<endl; //lead to an error:
    //‘s_nID’ was not declared in this scope
    std::cout << GenerateID() << std::endl;
    //cout<<"s_nID = "<<s_nID<<endl; //lead to an error here too
    std::cout << GenerateID() << std::endl;
    std::cout << GenerateID() << std::endl;
    return 0;
}
```

0  
1  
2

# Coding rules: name of variables

- C++ Rules:

- starts with a letter
- *only* contains letters,digits, underscores
- cannot use *keywords* names (if, int, ...)

- Recommendations

- Choose meaningful names
  - Avoid confusing abbreviations and acronyms
  - Use conventions (i,j,k as loop indexes by example)
- Avoid overly long names
  - the\_number\_of\_channels // too long
  - nof\_channels // shorter and meaningful
  - Nofc; // what does it mean ??
- You could define your own rules (or the own of your team)
  - Use of capital letters, underscore
  - Examples
    - ROOT class names starts with a “T” (ex:**TGraph**)
    - variable with a “f” (ex: **fEntries**)
    - Accessors starts with “Get” (ex: histo->**GetXaxis()** )

**Are forbidden:**

- x
- 12x
- Time.Acquisition@CERN
- My Variable
- ...

# Instructions

## Selection (if/else)

```
if( test){  
    Instructions1;  
}  
elseif(test2){ // optional  
    Instructions2;  
}  
else{  
    Instructions3;  
} // brackets not needed if  
// there is only one line
```

Ex: if(a>b) max=a;  
 else max=b;

## Condensed syntax

```
test ? Inst1: Inst2 ;
```

Ex: (a>b) ? max=a: max=b;

## Loop: for

```
for(initialize,condition,increment){  
    instructions;  
}  
Initialize: ex: int i=0;  
Condition: ex: i<10  
Increment: ex: i++ (i=0 at it. 1)  
           ++i (i=1 at it. 1)
```

**For** is used when the number of iterations is well defined

(ex: summation of all elements of an array/vector)

## Loop: while

```
while(condition){  
    Instructions;  
}
```

**While** is mandatory when the number of iterations is not known before running time  
(ex: minimization problem)

```
do{  
    Instructions;  
}  
while(condition)
```

**Ensure that instructions are run at least once**

# Control commands

- Break
  - Allow to stop a loop
- Continue
  - Allow to bypass a section of code
  - Used in loops to go directly to next iteration
- Return
  - Ends a function (**Ex:** main)
  - Can be followed by a variable

```
int max = 10000;
int sum = 0;
for(int i=0;i<100;i++){
    if( (i*i)%3==0 ) continue; // does not sum if i^2 is a multiple of 3
    sum+=i*i;
    if(sum>=max) break; // stops is sum is greater than max
}
return 0 ; //ends the main function

//useless code
for(int i=0;i<10;i++)
}
```

# Precedence and associativity

- Operator precedence determines which operator will be performed first in a group of operators with different precedences
  - Ex:  $5+3*2$  computed as  $5+(3*2) = 11$  and not  $(5+3)*2 = 16$
- The operator associativity rules define the order in which adjacent operators with the same precedence level are evaluated
  - $8-3-2$  computed as  $(8-3)-2=3$  and not  $8-(3-2)=7$

Max priority ↑

Operator Name	Associativity	Operators
Primary scope resolution	left to right	::
Primary	left to right	() [] . -> dynamic_cast typeid
Unary	right to left	++ -- + - ! ~ & * (type_name) sizeof new delete
C++ Pointer to Member	left to right	.->*
Multiplicative	left to right	* / %
Additive	left to right	+ -
Bitwise Shift	left to right	<< >>
Relational	left to right	< > <= >=
Equality	left to right	== !=
Bitwise AND	left to right	&
Bitwise Exclusive OR	left to right	^
Bitwise Inclusive OR	left to right	
Logical AND	left to right	&&
Logical OR	left to right	
Conditional	right to left	? :
Assignment	right to left	= += -= *= /= <<= >>= %= &= ^=  =
Comma	left to right	,

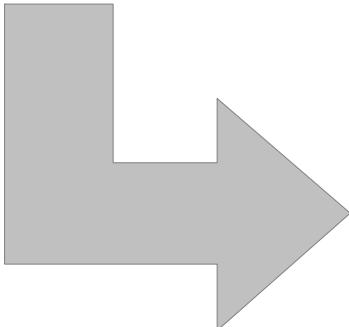
- To ensure that your calculus will be performed as you expected, you can always add **parentheses**.
- Nevertheless it is better to not “overload” your code with unnecessary ()

From <http://n.ethz.ch/~werdemic/download/week3/C%20Precedence.html>

# Input/Output

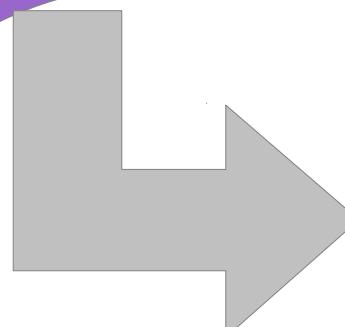
## INPUT:

- Keyboard (**default**)
- Files
- Data base
- Other input device
- Other programs
- Other part of a program



## Code

Make some **computation** partially based on the input (if any) and produce an output !



## OUTPUT:

- Screen (**default**)
- Files
- Data base
- Other input device
- Other programs
- Other part of a program

# Input and Output

<b>cin</b>	Standard input stream ( <a href="#">object</a> )
<b>cout</b>	Standard output stream ( <a href="#">object</a> )
<b>cerr</b>	Standard output stream for errors ( <a href="#">object</a> )
<b>clog</b>	Standard output stream for logging ( <a href="#">object</a> )

```
./prog.exe > log.stdout      # redirect only cout streams  
./prog.exe 1> log.stdout     # idem  
  
. ./prog.exe 2>log.stderr    # redirect cerr streams  
. ./prog.exe 2>/dev/null      # avoid having cerr streams on screen or in a file  
. ./prog.exe > log.txt 2>&1   # redirect cout & cerr streams  
. ./prog.exe &> log.txt       # idem
```

<b>ifstream</b>	Input file stream class ( <a href="#">class</a> )
<b>ofstream</b>	Output file stream ( <a href="#">class</a> )

# I/O and types

```
int a = 0;
char c = 'a';
float f = 0.0;
string s;

cin>>a;
cin>>c;
cin>>f;
cin>>s;

cout<<"int: "<<a<<" float: "<<f<<" char: "<<c<<" string: "<<s<<endl;
```

Separator can be a space or a new line

Bad input can lead to errors and stop the program

- Ex: Enter a character for an integer or a float

It can also lead to unexpected behaviour

- One **should** protect the code for this !

**Correct behaviour:**

**Input:** 1 3.4 a toto

**Output:** int: 1 float: 3.4 char: a string:  
toto

**Input:** 1 3.4 1 3.4

**Output:** int: 1 float: 3.4 char: 1 string: 3.4

**“Undesired” behaviour:**

**Input:** 1 3.4 abc toto

**Output:** int: 1 float: 3.4 char: a string:

**Input:** 1.2 a toto

**Output:** int: 1 float: 0.2 char: a string:  
toto

# I/O and types

```
int a = 0;
char c = 'a';
float f = 0.0;
string s;

cin>>a;
cin>>c;
cin>>f;
cin>>s;

cout<<"int: "<<a<<" float: "<<f<<" char: "<<c<<" string: "<<s<<endl;
```

Separator can be a space or a new line

Bad input can lead to errors and stop the program

- Ex: Enter a character for an integer or a float

It can also lead to unexpected behaviour

- One **should** protect the code for this !

**Correct behaviour:**

**Input:** 1 3.4 a toto

**Output:** int: 1 float: 3.4 char: a string:  
toto

**Input:** 1 3.4 1 3.4

**Output:** int: 1 float: 3.4 char: 1 string: 3.4

**“Undesired” behaviour:**

**Input:** 1 3.4 abc toto

**Output:** int: 1 float: 3.4 char: a string:

**Input:** 1.2 a toto

**Output:** int: 1 float: 0.2 char: a string:  
toto

# I/O types

“**cout**” can redirect all built-in types and some std library types (string, complex,...)

```
int a = 12;
float f = 12.345;
complex<double> d(1.23,4.56);
char c = 'a';
string s = "my string with whatever I want: @ 3.14 ;-) ...";

cout<<"a: "<<a<<endl;
cout<<"f: "<<f<<endl;
cout<<"d: "<<d<<endl;
cout<<"c: "<<c<<endl;
cout<<"s: "<<s<<endl;
```

```
a: 12
f: 12.345
d: (1.23,4.56)
c: a
s: my string with whatever I want: @ 3.14 ;-) ...
```

<< operator can also be **overloaded** to any user-defined type !

- You can define the desired precision
- Precision of the value and printing it are different things.

```
#include <iomanip>

double pi =3.14159;
cout << std::setprecision(5) << f << endl;
cout << std::setprecision(9) << f << endl;
cout << std::fixed;
cout << std::setprecision(5) << f << endl;
cout << std::setprecision(9) << f << endl;
```

```
12.345
12.3450003
12.34500
12.345000267
```

# I/O types

## List of special characters:

```
cout<<"\a \v\t bcdefghi \v\t c \n";
cout<<"1\t 2\v3\v\b4"<<endl;

//question - answer
cout<<"what's your name ? \f Chabert"<<endl;

//in a loop ...
for(int i=0;i<1E12;i++){
    // overprint a message on the latest line (here the iterator)
    if(i%1000==0) cout<<i<<"\r";
    pow(3.13,5); // some stupid calculation
}
```

### Control characters:

- \a = alert (bell)
- \b = backspace
- \t = horizontal tab
- \n = newline (or line feed)
- \v = vertical tab
- \f = form feed
- \r = carriage return

### Punctuation characters:

- \" = quotation mark (backslash not required for "")
- \' = apostrophe (backslash not required for '')
- \? = question mark (used to avoid trigraphs)
- \\ = backslash

```
a
bcd...hi
c
1
2
3
4
what's your name ?
Chabert
41284000
```

# Go further: *Fancy cout* !

```
cout << "\033[1;5;102;31mbold red text\033[0m\n";
cout << " normal printing \n";
cout << "\033[105;34mblue text\033[0m\n";
cout << "\033[4m underline text\033[0m\n";
```

bold red text  
normal printing  
blue text  
underline text

**Bold:** 1  
Underline: 4  
Text color: 30+x  
Background color: 100+x

Color table [10]							
Intensity	0	1	2	3	4	5	6
Normal	Black	Red	Green	Yellow	Blue	Magenta	Cyan
Bright	Black	Red	Green	Yellow	Blue	Magenta	Cyan

Although these features might be amazing, there are not deployed/available on all systems

This illustrates an other problem, having a **portable** code :

- Support on most **compilers**
- Support on most **O.S.**

This is not always a request for physicists but it depends on the case.

# Files: input/output

- Input file: ifstream

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
    string line;
    ifstream myfile ("example.txt");
    if (myfile.is_open())
    {
        while ( getline (myfile,line) )
        {
            cout << line << '\n';
        }
        myfile.close();
    }

    else cout << "Unable to open file";

    return 0;
}
```

## #include <ifstream>

### Reading can be performed:

- Per line: **getline()**
- Per character(s): **get()**
- Ignore characters: **ignore()**
- Read buffer: **read()**, **readsome()**
- Depending on a format: **operator>>**

### Check state flag:

- **eof()**: check the end of file
- **good()**: state of stream is good
- **bad()**: true if a reading or writing operation fails
- **fail()**: true if bad() and if a format error happens

Many more possible options. Check documentation !

# Files: input/output

- Output file: ofstream

#include <fstream>

## Opening modes:

ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file.
ios::trunc	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

```
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
    ofstream myfile ("example.txt");
    if (myfile.is_open())
    {
        myfile << "This is a line.\n";
        myfile << "This is another line.\n";
        myfile.close();
    }
    else cout << "Unable to open file";
    return 0;
}
```

```
[file example.txt]
This is a line.
This is another line.
```

# Functions

Functions represent/implement computations/algorithms

- Return type (int, void )
  - Return **one** variable at maximum
  - **Void** means don't return a value
  - Type can be an user-defined class
- Name
- “Arguments” or “parameters”
  - (last) parameters can have default value
- Body
  - (in the definition)

```
Hello
square of 5 = 25
square of 6 = 36
sum(2-9) = 44
sum(from 2 to max) = 5049
default sum = 5050
```

```
void PrintMessage(string message){
    cout<<message<<endl;
}

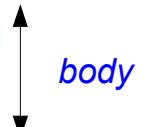
//int square(int a); // just the declaration

int square(int a){ // declaration and implementation
    return a*a;
}

int sum(int min=1, int max=100){
    int sum=0;
    for(int i=min;i<=max;i++) sum+=i;
    return sum;
}

int main(){
    PrintMessage(string("Hello"));
    int a = 5;
    int aa = square(a);
    cout<<"square of "<<a<<" = "<<aa<<endl;
    cout<<"square of 6 = "<<square(6)<<endl;
    cout<<"sum(2-9) = "<<sum(2,9)<<endl;
    cout<<"sum(from 2 to max) = "<<sum(2)<<endl;
    cout<<"default sum = "<<sum()<<endl;

    return 0 ;
}
```



Possibility to declare many functions with the same name in the same scope if they have different arguments (number,type)

# Function: call by value, ref, ...

```
//functions that return the square of a+1

//Call by value: a copy of a will be made -
//'a' can be modified inside the function without effect outside
int square_ap1_val(int a) { ++a; return a*a; }

//Call by reference
//if 'a' is modified inside the function it will have consequence afterward
int square_ap1_ref(int& a) { ++a; return a*a; }

//Call by const reference
//It ensure that the function does not have the right to modified the value
int square_ap1_cref(const int& a) {
    // ++a ; return a; // this is forbidden: compilation error
    int b = a+1;
    return b*b; //not the most relevant implementation here
}

//Call by pointer
int square_ap1_point(int* a) { ++(*a); return (*a)*(*a); }
//int* const a: would have ensure have the pointer could not change
//const int* a: would have ensure that the pointed value could'nt change
//          but not always possible

//Call by pointer
//Here the pointed value of a will be modified to be (a+1)^2
void square_ap1_vpoint(int* a) {++(*a); (*a)*=(*a); }
```

call-by-value:	res = 4 val = 1
call-by-reference:	res = 4 val = 2
call-by-const-reference:	res = 9 val = 2
call-by-pointer:	res = 9 val = 3
void function - call-by-pointer:	val = 16

```
int main(){
    int value = 1;
    int result = 0;
    result = square_ap1_val(value);
    cout<<"call-by-value: \t\t\tres = "<<result<<" val = "<<value<<endl;
    result = square_ap1_ref(value);
    cout<<"call-by-reference: \t\t\tres = "<<result<<" val = "<<value<<endl;
    result = square_ap1_cref(value);
    cout<<"call-by-const-reference: \t\t\tres = "<<result<<" val = "<<value<<endl;
    result = square_ap1_point(&value);
    cout<<"call-by-pointer: \t\t\tres = "<<result<<" val = "<<value<<endl;
    square_ap1_vpoint(&value);
    cout<<"void function - call-by-pointer: \t\t\tval = "<<value<<endl;

    return 0 ;
}
```

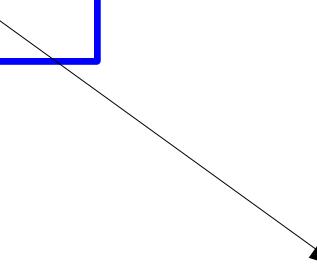
# Functions: guidance for arguments

- Use call-by-value for small objects only
- Use call-by const-reference for large objects
- Return a result rather than modify an object through a reference argument
- Use call-by reference only when you have to
  - Ex: case of multiples outputs
- Be careful with the use of pointers
  - Take care of deletion
  - Modification of the pointer
  - Modification of the pointed value

# int main(int argc,char\*\* argv)

- It might be convenient to “transmit” information to the program from the command line
- It avoid to recompile the code to change its execution
- It might avoid to read configuration file

```
int main(int argc, char** argv){  
  
    cout<<"There are "<<argc<<" arguments !"<<endl;  
    for(int i=0;i<argc;i++)  
        cout<<"\t"<<argv[i]<<endl;  
  
    return 0 ;  
}
```



```
There are 5 arguments !  
- ./a.out  
- toto  
- 2  
- @1  
- myfile
```

# int main(int argc,char\*\* argv)

```
void help(){
    cout<<"usage: prog.exe [options] filename"<<endl;
    cout<<"list of options:"<<endl;
    cout<<" -d: debug"<<endl;
    cout<<" -o outfile: write outputs in a file"<<endl;
}

bool ReadArguments(int argc, char** argv, bool debug, string ifilename, string ofilename){
    for(int i=1;i<argc;i++){
        cout<<argv[i]<<endl;
        if(string(argv[i])==string("-d")){
            debug = true;
        }
        else if(string(argv[i])==string("-o")){
            if(i<argc-1){
                ofilename = argv[i+1];
                cout<<ofilename<<endl;
                ++i;
            }
            else cerr<<"outfile is missing"<<endl;
        }
        else {
            ifilename = string(argv[i]);
        }
    }
    if(ifilename==string()) return true;
    else return false;
}

int main(int argc, char** argv){
    if(argc==1){
        help();
        return 1;
    }
    string ifilename;
    string ofilename;
    bool debug = false;
    if(!ReadArguments(argc,argv,debug,ifilename,ofilename)) return 2;
    //instructions ...
    return 0 ;
}
```

usage: prog.exe [options] filename  
list of options:  
-d: debug  
-o outfile: write outputs in a file

void help(){
 cout<<"usage: prog.exe [options] filename"<<endl;
 cout<<"list of options:"<<endl;
 cout<<" -d: debug"<<endl;
 cout<<" -o outfile: write outputs in a file"<<endl;
}

./prog.exe -d -o ofile.txt ifile.txt

ming/C++

48

# Call system

- Invokes the command processor
- **Warning:** *the command called is system/library dependent !*
- Possibility to parse the output value BUT not the output of the command

```
#include <iostream>
#include <stdlib.h> // required to call system

using std::cout;
using std::endl;

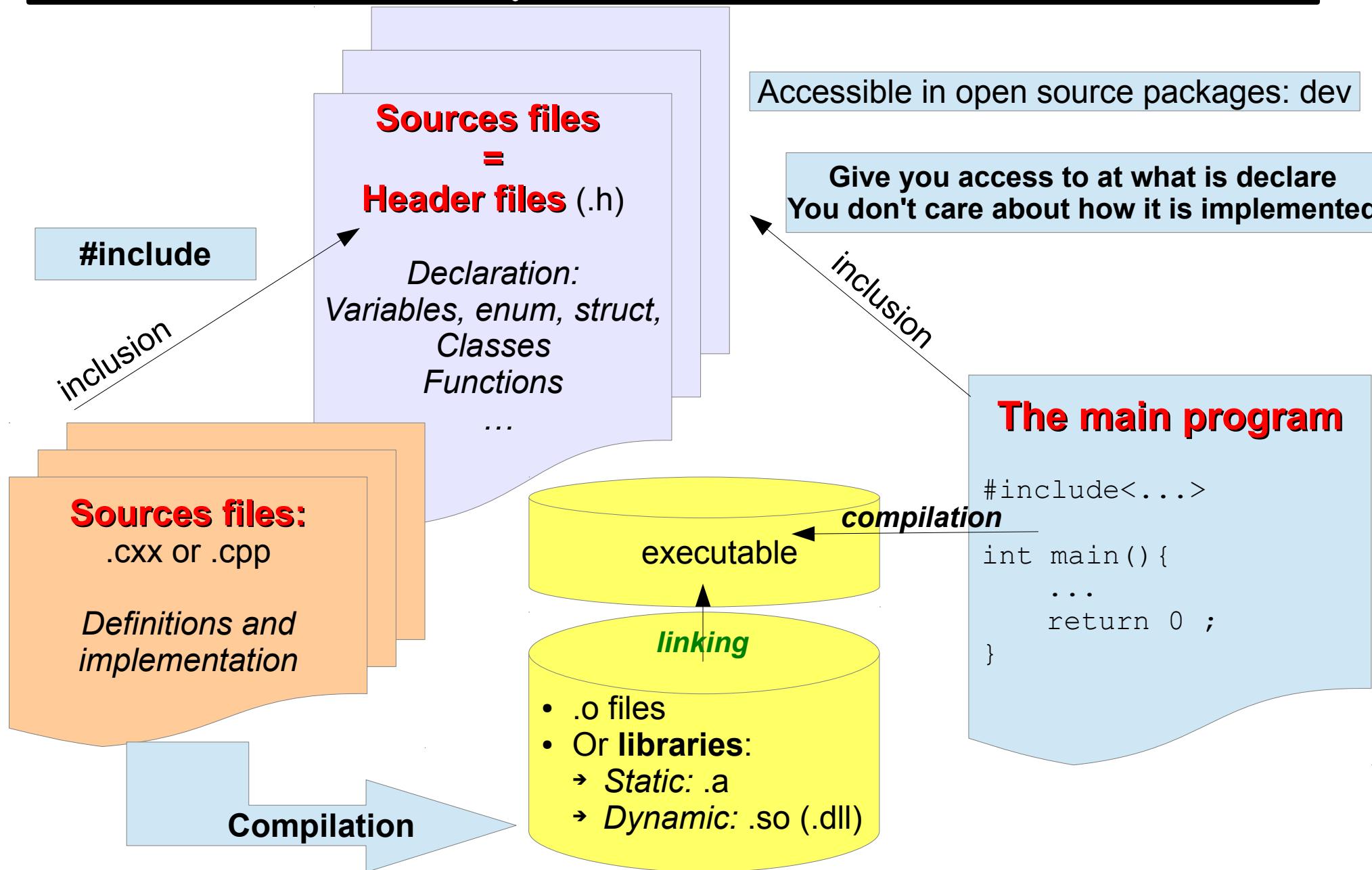
int main ()
{
    int i;
    cout<< "Checking if processor is available ...";
    if (system(NULL)) cout<< "Ok" << endl;
    else exit (EXIT_FAILURE);
    cout<< "Executing command ls ... " << endl;
    i=system ("ls");
    cout<<"The value returned was: "<< i << endl;
    return 0;
}
```

Could be convenient for many applications:

- Compile generated latex code
- Manipulation of files/folders
- ...

```
Checking if processor is available...ok
Executing command ls ..
myinclude.h      system.cpp
The value returned was: 0.
```

# Compilation chain



# Compilation chain

**Example of one class (*class.h & class.cpp file*) and a main program (*main.cpp*)**

## 1 step compilation

class.cpp will be recompiled even if only main.cpp changed

```
g++ class.cpp main.cpp -o main.exe
```

## 2 steps compilation

```
g++ -c class.cpp  
g++ class.o main.cpp -o main.exe
```

First line can be omitted if only main.cpp changed

## 2 steps compilation + use of libToto.so

```
g++ -c class.cpp  
g++ -I headerDir -L libDir -LToto class.o main.cpp -o main.exe
```

Toto.h is in headerDir  
libToto.so is in libDir  
LibDir might be in \$LD\_LIBRARY\_PATH

# Compilation chain

2 steps compilation + use of ROOT lib.

```
g++ -c class.cpp `root-config --cflags --glbs`  
g++ class.o main.cpp -o main.exe `root-config --cflags --glbs`
```

“3 steps” compilation with shared library

```
g++ -c class.cpp # do the same for other classes  
g++ -fPIC -shared class.cpp -o libPerso.so  
g++ main.cpp -o main.exe -L libPersoDir -IPerso
```

To know the symbols inside .so

```
nm -s --demangle libPoint.so
```

To list shared library dependencies:

```
ldd main.exe
```

# Few compilation options

- Previously listed
  - -o *outputfile*
- Warning options
  - -Wall: combination of many warnings ...
  - -Wfloat-equal ....
- Debugging options
  - -g: produce debugging info that could be used by the debugger program **GDB**
- Optimization options: *following options are needed to speed-up execution time*
  - *WARNING: by default compiler try to reduce compilation time*
  - -O1 (space/speed tradoff) -O2 (speed opt.) -O3 (inline,regist.) -Os (-O2 + code size reduction)
- Linker options:
  - -L *libdir* -l*library* -shared (to create .so)
- Compilation report:
  - ftime-report -fmem-report
- Preprocessor options
- ...

# Makefile

## Makefile

```
all: hello

hello: main.o factorial.o hello.o
    g++ main.o factorial.o hello.o -o hello

main.o: main.cpp
    g++ -c main.cpp

factorial.o: factorial.cpp
    g++ -c factorial.cpp

hello.o: hello.cpp
    g++ -c hello.cpp

clean:
    rm -rf *o hello
```

make # or make all

```
CC=g++
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.cpp hello.cpp factorial.cpp
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
    $(CC) $(LDFLAGS) $(OBJECTS) -o $@

.cpp.o:
    $(CC) $(CFLAGS) $< -o $@
```

## Using variables & comments

```
# I am a comment, and I want to say that the variable CC will be
# the compiler to use.
CC=g++
# Hey!, I am comment number 2. I want to say that CFLAGS will be the
# options I'll pass to the compiler.
CFLAGS=-c -Wall

all: hello

hello: main.o factorial.o hello.o
    $(CC) main.o factorial.o hello.o -o hello

main.o: main.cpp
    $(CC) $(CFLAGS) main.cpp

factorial.o: factorial.cpp
    $(CC) $(CFLAGS) factorial.cpp

hello.o: hello.cpp
    $(CC) $(CFLAGS) hello.cpp

clean:
    rm -rf *o hello
```

Parallelization can be useful  
for big projects

make -j NofNodes

.o files are not reproduced (compilation) if .cpp  
doesn't change

# Scopes

- **Global scope** (accessible everywhere)
- **Class scope**
- **Local scope** (between {..}: loop, functions,...)
- **Statement scope** (in a for-statement)

```
int x;           // global variable – avoid those where you can
int y;           // another global variable
int f() {
    int x;     // local variable (Note – now there are two x's)
    x = 7;      // local x, not the global x
    {
        int x = y; // another local x, initialized by the global y
                    // (Now there are three x's)
        x++;
    }
}               // increment the local x in this scope
// avoid such complicated nesting and hiding: keep it simple!
```

## Remarks

- A name in a scope can be seen from within its scope and within scopes nested within that scope
- A scope keeps “things” local
  - Prevent var. and functions to interfere with outside
  - Keep names as local as possible

# Namespaces

- A **namespace** is a named scope
- The syntax `::` is used to specify which **namespace** you are using and which (of many possible) objects of the same name you are referring to
  - **Ex:** You want to create your own class “string”. But it already exists ...
    - `std::string` will refer the class implemented in the stl
    - `your_name_space::string` will refer to your own implementation
- How to create my namespace ?
  - You can encapsulate things (functions, classes, enums, ...) as following

```
namespace Xproject{ // create a namespace called Xproject
    const double pi = 3.14159;           // variable
    double square(double a) {return a*a;} // function
    class X{                           // class
        //...
    };
}
```

- How to avoid calling everywhere the namespace ??
  - **Ex:** using `std::cout`;
  - **Ex:** using `namespace std;`

# Pointer & address

```
int a = 10;      //declare a integer
int* pa = &a;    //declare a pointer to an integer and initialize it to the adress of a
cout<<"a: "<<a<<" - its adress: " <<&a<<endl;
cout<<"pa: "<<*pa<<" - its adress: " <<pa<<endl;
cout<<"##### - its adress: " <<endl<<endl;

++a;
cout<<"a: "<<a<<" \t(*pa): "<<(*pa)<<endl;
++(*pa); // once you give one (or many) pointer to a variable, nothing prevent that the value could change
cout<<"a: "<<a<<" \t(*pa): "<<(*pa)<<endl;
++pa;
cout<<"a: "<<a<<" - its adress: " <<&a<<endl;
cout<<"pa: "<<*pa<<" - its adress: " <<pa<<endl;
cout<<"##### - its adress: " <<endl<<endl;
```

```
int b = 11;
int* pb;          //declare a pointer to integer which is not initialized !
const int* cpb = &b; //declarer a      pointer to an const integer
int* const pbc = &b; //declarer a const pointer to an integer
cout<<"pb: "<<*pb<<" - its adress: " <<pb<<endl;
pb = &b;
cout<<"b: "<<b<<" - its adress: " <<&b<<endl;
cout<<"pb: "<<*pb<<" - its adress: " <<pb<<endl;
cout<<"cpb: "<<*cpb<<" - its adress: " <<cpb<<endl;
//++(*cpb); // compilation error: increment of read-only location '* cpb'
cpb++; //allowed: the pointer is not const (the pointed value is const)
//pbc++; //compilation error: increment of read-only variable 'pbc'
cout<<"##### - its adress: " <<endl<<endl;
```

```
const int c = 12;
//int* pc = &c; // this lead to and compilation error
const int* cpc = &c; // this lead to and compilation error
cout<<"c: "<<c<<" - its adress: " <<&c<<endl;
cout<<"cpc: "<<*cpc<<" - its adress: " <<cpc<<endl;
```

```
//++c; //error - you cannot change a const variable - compilation error: increment of read-only variable 'c'
//++(*cpc); // compilation error: increment of read-only location '* cpc'
```

```
a: 10 - its adress: 0xffff772d3644
pa: 10 - its adress: 0xffff772d3644
#####
a: 11  (*pa): 11
a: 12  (*pa): 12
a: 12 - its adress: 0xffff772d3644
pa: 4196208 - its adress: 0xffff772d3648
#####
pb: 1999458643 - its adress: 0xffff772d3748
b: 11 - its adress: 0xffff772d3648
pb: 11 - its adress: 0xffff772d3648
cpb: 11 - its adress: 0xffff772d3648
#####
c: 12 - its adress: 0xffff772d364c
cpc: 12 - its adress: 0xffff772d364c
```

# Pointer & reference

```
int a = 2;
int b = 10;

const int& cr = a;
int& r = a;

cout<<"a = "<<a<<" b = "<<" ref-to-a: "<<r<<" const-ref-to-a: "<<cr<<endl;
// ++cr; // this is forbidden
++r;      // will modified both value of r and a
cout<<"a = "<<a<<" b = "<<" ref-to-a: "<<r<<" const-ref-to-a: "<<cr<<endl;
r = b; // r take the value of b but the reference does not change !
cout<<"a = "<<a<<" b = "<<" ref-to-a: "<<r<<" const-ref-to-a: "<<cr<<endl;
```

```
a = 2 b =  ref-to-a: 2 const-ref-to-a: 2
a = 3 b =  ref-to-a: 3 const-ref-to-a: 3
a = 10 b = ref-to-a: 10 const-ref-to-a: 10
```

- You can't modify an object through a const reference
- You can't make a reference refer to another object after initialization  
(difference from a pointer)

# Pointer & reference

	Pointer	Reference
<b>Must be initialized</b>	no	yes
<b>Can be null (=0)</b>	yes	no
<b>Can change the “pointed” variable</b>	yes	no
<b>Can change the value of the “pointed” variable</b>	yes (no if type* const)	yes (no if const type &
<b>Can delete the memory</b>	yes	no

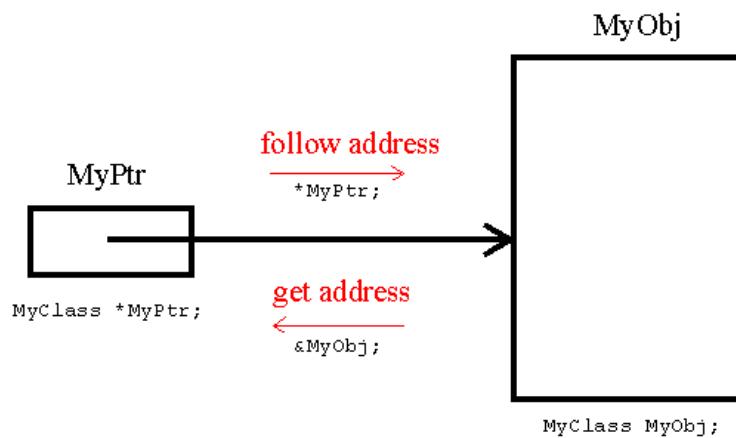
**There shall be no references to references, no arrays of references, and no pointers to references.**

# Pointer & reference: Memory

```
short si = 2;           short* psi = &si;           short& rsi = si;
int i = 2;              int* pi = &i;             int& ri = i;
double d = 2.;          double* pd = &d;          double& rd = d;
string s = "Hello, world!"; string* ps = &s;   string& rs = s;

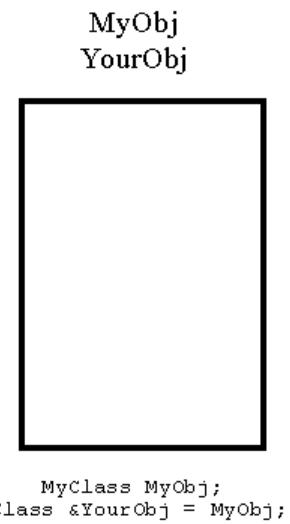
cout<<"Short : size = "<<sizeof(si)<<"\t size(pointer):"<<sizeof(psi)<<"\t size(ref):"\t<<sizeof(rsi)<<endl;
cout<<"Int   : size = "<<sizeof(i)<<"\t size(pointer):"<<sizeof(pi)<<"\t size(ref):"\t<<sizeof(ri)<<endl;
cout<<"Double: size = "<<sizeof(d)<<"\t size(pointer):"<<sizeof(pd)<<"\t size(ref):"\t<<sizeof(rd)<<endl;
cout<<"String: size = "<<sizeof(s)<<"\t size(pointer):"<<sizeof(ps)<<"\t size(ref):"\t<<sizeof(rs)<<endl;
```

A Pointer holds the address of an object



Reference is an alias !

References:  
another name for  
the same object



the size in memory of a pointer depends on the platform where the program runs

Short : size = 2	size(pointer):8	size(ref): 2
Int   : size = 4	size(pointer):8	size(ref): 4
Double: size = 8	size(pointer):8	size(ref): 8
String: size = 8	size(pointer):8	size(ref): 8

# Test on pointer

- It is always safer to test if a pointer is not null before accessing the pointed variable !
- Could be useful to not allocate and delete twice memory (see example below)

```
void AllocateMemory(int*& array, int size){  
    if(array==0) // or array==NULL  
        array = new int[size];  
        for(int i=0;i<size;i++) array[i] = 0;  
    }  
    else cerr<<"\tError: Memory has already been allocated" << endl;  
    return;  
}  
  
void FreeMemory(int*& array){  
    if(!array){  
        delete[] array;  
        array = 0;  
    }  
    else cerr<<"\tError: Memory has already been free" << endl;  
}  
  
int main(){  
    int* array = 0;  
  
    cout<<"First call of AllocateMemory" << endl;  
    AllocateMemory(array,10);  
    cout<<"Second call of AllocateMemory" << endl;  
    AllocateMemory(array,5);  
  
    cout<<"First call of FreeMemory" << endl;  
    FreeMemory(array);  
    cout<<"First call of FreeMemory" << endl;  
    FreeMemory(array);  
  
    return 0 ;  
}
```

## Tests on pointers:

- Pointer==0
- !Pointer

```
First call of AllocateMemory  
Second call of AllocateMemory  
    Error: Memory has already been allocated  
First call of FreeMemory  
    Error: Memory has already been free  
First call of FreeMemory  
    Error: Memory has already been free
```

# Arithmetic of pointers

- Several operators are also defined for pointers: **++**, **--**
- It will allow you to change the address and by consequence the pointed “object”
- The result of those operations are not guaranteed and protection have to be written
- The operation depends on the kind of object type used

```
#include <iostream>

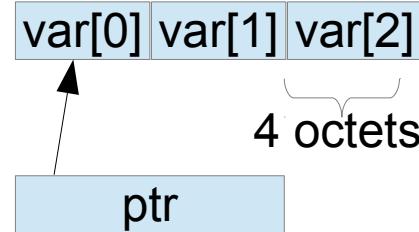
using namespace std;
const int MAX = 3;

int main ()
{
    int var[MAX] = {10, 100, 200};
    int *ptr;

    // let us have array address in pointer.
    ptr = var;
    for (int i = 0; i < MAX; i++)
    {
        cout << "Address of var[" << i << "] = ";
        cout << ptr << endl;

        cout << "Value of var[" << i << "] = ";
        cout << *ptr << endl;

        // point to the next location
        ptr++;
    }
    return 0;
}
```



```
Address of var[0] = 0xbfa088b0
Value of var[0] = 10
Address of var[1] = 0xbfa088b4
Value of var[1] = 100
Address of var[2] = 0xbfa088b8
Value of var[2] = 200
```

int\* ptr;  
++ptr move by 4 octets

double\* ptr;  
++ptr move by 8 octets

# Arrays

- It's all that C has – It's mainly used in many C++ packages
- Array don't know their own size
  - Often use their size as an arguments in functions
- Access to elements
  - First element has index 0. **Ex:** tab[0]
- Avoid arrays whenever you can:
  - largest source of bug in C and (unnecessarily in C++)
  - among the largest source of security violations:
    - Possibility to access non declared memory (runtime error or unexpected behavior)

# Arrays: initialization

```
char array_char[] = "Hello, world"; // array of 13 chars: 12 + 1 end character
                                         // the compiler counts it for you !
char array_char2[100]; // array of 100 uninitialized char
cout<<"first char array:"<<array_char<<endl;
cout<<"second char array:"<<array_char2<<endl;
cout<<"-----"<<endl;

int array_int[] = {1,2,3,4,5,6}; // array of 6 ints (no ending character for int)
int array_int2 [10] = {1,2,3,4,5,6}; // array of 10 ints, the 4 last are initialized to 0 by default

double array_float[10] = {};//array of 10 elements initialized to 0.0
double array_float2[10]; //array of 10 elements not initialized : VERY DANGEROUS !

for(int i=0;i<10;i++){
    cout<<array_float[i]<<"\t"<<array_float2[i]<<endl;
}
```

It is safer to **always** initialize the arrays !

```
first char array:Hello, world
second char array:♦00
-----
0      3.11043e-317
0      0
0      6.95327e-310
0      6.95327e-310
0      0
0      6.9341e-310
0      6.93405e-310
0      0
0      6.9341e-310
0      6.9341e-310
```

# Array: dynamical allocation

**Possible memory leak ....**

```
int main(){
    int size=100000;
    for(int i=0;i<100;i++){
        double* tab = new double[size];
        for(int j=0;j<i-1;j++) tab[j]=sqrt(j);
        //will lead to memory leak if memory is not free before the end of the loop ...
    }
    return 0 ;
}
```

**Always free memory:**

- ***when it will be not used anymore***
- ***when you still have access to the pointer !***
- ***when you are “owner” of the ‘memory’ (pb of double free)***

# Array and pointer

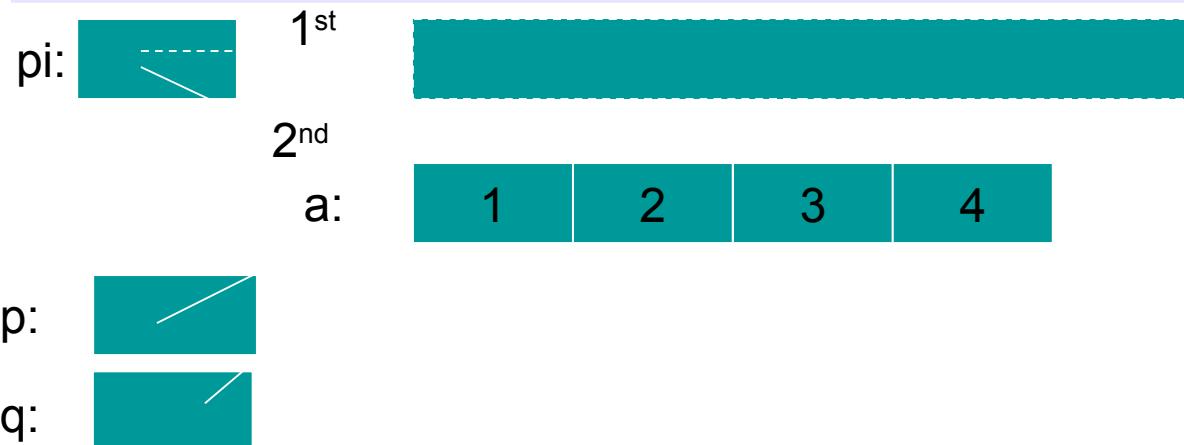
```
int ai[]={1,2,3,4,5};  
int* pai = NULL;  
pai = ai; //the name of an array name point to the first element  
cout<<"pai[0] = "<<pai[0]<<" pai[4] = "<<pai[4]<<endl;  
pai = &ai[2]; //pointer to ai's 3rd element (starting at 0)  
cout<<"pai[0] = "<<pai[0]<<" pai[4] = "<<pai[4]<<endl; //pai[4] is out of range ... (mistake !)
```

```
pai[0] = 1 pai[4] = 5  
pai[0] = 3 pai[4] = -1273218472
```



# Array and pointer

```
void f(int pi[ ]) // equivalent to void f(int* pi)
{
    int a[ ] = { 1, 2, 3, 4 };
    int b[ ] = a; // error: copy isn't defined for arrays
    b = pi;        // error: copy isn't defined for arrays. Think of a
                   // (non-argument) array name as an immutable pointer
    pi = a;        // ok: but it doesn't copy: pi now points to a's first element
                   // Is this a memory leak? (maybe)
    int* p = a;    // p points to the first element of a
    int* q = pi;   // q points to the first element of a
}
```



from B. Stroustrup slides

# Array and pointers

```
char* f()
{
    char ch[20];
    char* p = &ch[90];
    // ...
    *p = 'a';                      // we don't know what this'll overwrite
    char* q;                       // forgot to initialize
    *q = 'b';                      // we don't know what this'll overwrite
    return &ch[10];                  // oops: ch disappears upon return from f()
                                    // (an infamous "dangling pointer")
}

void g()
{
    char* pp = f();
    // ...
    *pp = 'c';                      // we don't know what this'll overwrite
                                    // (f's ch is gone for good after the return from f)
}
```

from B. Stroustrup slides

```

void f(int n, int* pai, int* pai2, int*& rpai, int*& rpai2){
    char ac[20]; //local array - "lives" until the end of the scope (end of the function) - on stack
    int ai[n]; //local array - size is known at execution time (was leading to error in the past)
    int* ai2 = new int[n]; // this works - dynamical allocation - BUT memory will not be deallocated at the end of the function

    for(int i=0;i<n;i++){
        ai[i] = i+1;
        ai2[i] = i+1;
    }
    pai = ai;
    rpai = ai;
    pai2 = ai2;
    rpai2 = ai2;
    cout<<"### In function f: "<<endl;
    cout<<"pointer adress (pai):"<<pai<<endl;
    cout<<"pointer adress (pai2):"<<pai2<<endl;
    cout<<"pointer adress (rpai):"<<rpai<<endl;
    cout<<"pointer adress (rpai2):"<<rpai2<<endl;
    cout<<"#####"<<endl;
}

```

```

int main(){

char ac0[10] = {};//global array - "lives" until the end of the program - in "static storage".
int max = 100;
int ai[max]; // allocated - not initialized
int* ai2 = new int[max]; //equivalent here

int* pai = NULL; // assign a NULL pointer
int* pai2 = 0; // does the same thing
int* rpai = NULL; // assign a NULL pointer
int* rpai2 = 0; // does the same thing
f(13,pai,pai2,rpai,rpai2);
cout<<"### In main: "<<endl;
cout<<"pointer adress (pai):"<<pai<<endl;
cout<<"pointer adress (pai2):"<<pai2<<endl;
cout<<"pointer adress (rpai):"<<rpai<<endl;
cout<<"pointer adress (rpai2):"<<rpai2<<endl;
cout<<"### access to elements: "<<endl;
cout<<"rpai[0]:"<<rpai[0]<<endl;
cout<<"rpai[1]:"<<rpai[1]<<endl;
cout<<"rpai2[0]:"<<rpai2[0]<<endl;
cout<<"rpai2[1]:"<<rpai2[1]<<endl;
cout<<"pai[0]:"<<pai[0]<<endl;

return 0 ;
}

```

```

### In function f:
pointer adress (pai):0x7fffb2f34dc0
pointer adress (pai2):0x246c1b0
pointer adress (rpai):0x7fffb2f34dc0
pointer adress (rpai2):0x246c1b0
#####
### In main:
pointer adress (pai):0
pointer adress (pai2):0
pointer adress (rpai):0x7fffb2f34dc0
pointer adress (rpai2):0x246c1b0
### access to elements:
rpai[0]:38191536
rpai[1]:0
rpai2[0]:1
rpai2[1]:2
Erreur de segmentation (core dumped)

```

# Dynamic allocation

- In some application, all memory needs cannot be determined before program execution by defining the variables needed.
- In that case, it is determined during runtime.
  - Ex: depends on user input(s), depends on the result of a calculus, ...
- Operators **new** and **new[]**

- build-in types
- Classes (lib/user)

```
//example with a simple int
int* a; //could be initialized as null pointer = 0; or = NULL
a = new int();

//example with an array of int
int* tab;
tab = new int[5]; //more generally size might be not defined before execution

int * foo;
foo = new (nothrow) int [5]; //nothrow is defined in <new>
//what happens when it is used is that when a memory allocation fails
//instead of throwing a bad_alloc exception or terminating the program,
//the pointer returned by new is a null pointer
//and the program continues its execution normally
if (foo == 0) {
    cerr<<" The dynamical allocation failed"<<endl;
    // error assigning memory. Take measures.
}

//example for an ROOT class
TH1F* h1; //pointer declaration
h1 = new TH1F("name1","title",10,0.,10.); //dynamic allocation

//All at once
TH1F* h2 = new TH1F("name2","title",10,0.,10.); //done in the same line
```

# Delete: free memory

- in most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory.
- This operation should be performed when variable is still in the scope
  - End of a loop or function
  - In the destructor of a class (if memory has been allocated in the constr.)
  - At the end of a program
- Operators delete
  - delete**: delete a single element in memory
  - delete[]**: delete an array of elements
- Pointer is not null after delete
  - You could do it yourself to ensure future test on pointers
- You can't delete twice memory: double free exception
  - If you have 2 pointers on the same element, make sure that only one of them will be deleted

```
delete a;
delete[] tab;
delete[] foo;
delete h1;
delete h2;
```

# std::vector

- Vector in C++ supersedes array defined in C
  - There are still a lot of applications using arrays rather than std::vector
- It properly deals with dynamic memory
- When vector is destructed, all its elements are deleted
- **Important:** the size of the vector is one of the data member of a std::vector ( contrary to an C array)
- Size is not fixed. Can be changed during program execution !

# std::vector

```
vector<int> ivec; // create a vector of integer
//other constructors
std::vector<int> second (4,100); // four ints with value 100
// iterating through second
std::vector<int> third (second.begin(),second.end());
std::vector<int> fourth (third); // a copy of third

//use of operator =: all elements are copied
ivec = second;

//fill the vector
ivec.push_back(10);
//idem in a loop
for(int i=0;i<10;i++) ivec.push_back(i*i);

//access to size and an element
if(ivec.size()>4) cout<<"Third element = "<<ivec[2]<<endl;

//--- loop over the vector
//with a "standard" for using .size()
for(int i=0;i<ivec.size();i++) cout<<"element "<<i+1<< ":" <<ivec[i]<<endl;
//similar with iterator
for(std::vector<int>::iterator it = ivec.begin() ; it != ivec.end(); ++it)
    cout<<"element: "<<*it<<endl;

//insert an element in 2nd position
ivec.insert(ivec.begin()+1,9999);
//possibility to insert an array
int myarray [] = { 501,502,503 };
ivec.insert (ivec.begin(), myarray, myarray+3);

//possibility to erase one or many elements
// ex: erase the first 2 elements:
ivec.erase (ivec.begin(),ivec.begin()+2);

//clear vector - it will free memory
ivec.clear();
```

# Ex: vector - pointer - delete

```
vector glob(10);                                // global vector – “lives” forever

vector* some_fct(int n)
{
    vector v(n);                                // local vector – “lives” until the end of scope
    vector* p = new vector(n);                // free-store vector – “lives” until we delete it
    // ...
    return p;
}

void f()
{
    vector* pp = some_fct(17);
    // ...
    delete pp;      // deallocate the free-store vector allocated in some_fct()
}
```

- **it's easy to forget to delete free-store allocated objects**
  - + so avoid **new/delete** when you can

from B. Stroustrup slides

# std::string

- **std::string** is a class that deals with character chains
- It “*supersedes*” `char*` (*inherited from C*)
- Many operations are easily possible
  - Access to size
  - Find a element
  - Retrieve a sub-string
  - Replace elements
  - Swap elements
  - ...

# string

```
string a("abcdef@1g"); // use constructor
string b = a+"!"; // affectation with use of operation+

//string comparison
if(a==b) cout<<"a & b are the same string"<<endl;
else cout<<"a & b are different strings"<<endl;

string c("abcdef@1h");
if(a>b) cout<<"a > b"<<endl;
else cout<<"a < b"<<endl;

//access to size
cout<<"size of the string: "<<a.size()<<endl;
cout<<"a = "<<a<<endl;

//retrieve a char* from a string
a.c_str(); //sometimes needed (ex: name of TH1F cannot be a string)

//access to a given element
cout<<"2nd element of a is: "<<a[1]<<endl;

//search
size_t pos = a.find("@"); // possibility to search for a char, char*, or string
cout<<"Charater @ is found in position :"<<pos<<" of string a"<<endl;
//check if it's found !
pos = a.find("cd");
if(pos!=std::string::npos){
    //performm a replace
    string rp = "CD";
    a.replace(pos,rp.size(),"CD");
}
cout<<"After replace: a = "<<a<<endl;

//clear
a.clear();
cout<<"a (after clear) ="<<endl;
```

**Many others things are possible**

```
a & b are different strings
a < b
size of the string: 9
a = abcdef@1g
2nd element of a is: b
Charater @ is found in position :6 of string a
After replace: a = abCDef@1g
a (after clear) =
```

# Int/float ↔ char\* conversion

- Char\* to number conversion : **#include<stdlib.h>**
  - **Atof**: convert char\* to float
  - **Atoi**: convert char\* to int

```
string a = "3.1416"; // initialize a string
float pi = atof(a.c_str()); // convert the char* extracted from the string to a float;
cout<<"a = "<<a<<" & pi = "<<pi<<endl;
```

a = 3.1416 & pi = 3.1416

- This is a common problem
- Above solution is coming from C, but we can used C++ tools (next slide)

# stringstream and conversion

#include <sstream>

```
int main(){

float f = 1.234;
stringstream ss; //create a stringstream
ss << f ; // add f to the stream
string s;
ss>>s; //redirect the stream to a string
cout<<"f = "<<f<<" s = "<<s<<endl;
//other possibility
string s2 = ss.str(); // return a string with the content of the stream
cout<<"f = "<<f<<" s2 = "<<s2<<endl;

cout<<"#####"<<endl;
stringstream ss2;
string sfloat ="1.2 3.4 5.6 7.8";
cout<<"string = "<<sfloat<<endl;
ss2 << sfloat;
float g = 0;
while((ss2>>g)){
    cout<<"float value: "<<g<<endl;
}
return 0 ;
}
```

```
f = 1.234 s = 1.234
f = 1.234 s2 = 1.234
#####
string = 1.2 3.4 5.6 7.8
float value: 1.2
float value: 3.4
float value: 5.6
float value: 7.8
```

# Preprocessor

**Preprocessor directives** are preceded by # (*only a single line*)

No use of semicolon to end the directive

**The preprocessor examines the code before the compilation**

**#define identifier replacement** (#**undefine**)

- It only replaces any occurrence of identifier
- Define a value
- Define function macros with parameters
- **#undefine**: ends the definition (could be used before changing the def.)

## Conditional inclusions

- **#ifdef**
- **#endif**
- **#if #else #elif**

```
#ifndef TABLE_SIZE  
#define TABLE_SIZE 100  
#endif  
int table[TABLE_SIZE];
```

```
#ifndef MYCLASS  
#define MYCLASS  
  
class MyClass{  
};  
  
#endif
```

Avoid multiple file inclusion

## #include

**#include <header>**: provided by the installed libraries (stl,...)

**#include “file.h”**: could be everywhere not only the installed packages

# Preprocessor: predefined macros

## Predefined macro names

- FILE
- LINE
- DATE
- TIME
- STDC
- STDC VERSION
- STDC HOSTED

```
// standard macro names
#include <iostream>
using namespace std;

int main()
{
    cout << "This is the line number " << __LINE__;
    cout << " of file " << __FILE__ << ".\n";
    cout << "Its compilation began " << __DATE__;
    cout << " at " << __TIME__ << ".\n";
    cout << "The compiler gives a __cplusplus value of " << __cplusplus;
    return 0;
}
```

This is the line number 7 of file /home/jay/stdmacronames.cpp.  
Its compilation began Nov 1 2005 at 10:12:29.  
The compiler gives a \_\_cplusplus value of 1

Those macro might be useful for **exception** and **error tracking**

# Preprocessor: predefined macros

## Predefined macro names

- FILE
- LINE
- DATE
- TIME
- STDC
- STDC VERSION
- STDC HOSTED

```
// standard macro names
#include <iostream>
using namespace std;

int main()
{
    cout << "This is the line number " << __LINE__;
    cout << " of file " << __FILE__ << ".\n";
    cout << "Its compilation began " << __DATE__;
    cout << " at " << __TIME__ << ".\n";
    cout << "The compiler gives a __cplusplus value of " << __cplusplus;
    return 0;
}
```

This is the line number 7 of file /home/jay/stdmacronames.cpp.  
Its compilation began Nov 1 2005 at 10:12:29.  
The compiler gives a \_\_cplusplus value of 1

Those macro might be useful for **exception** and **error tracking**

# Struct

A **struct** is a group of data elements grouped together under one name. These data elements, known as members, can have different types and different lengths

```
struct individual{  
    int age;  
    float weight;  
    string name;  
};
```

```
individual student1; //declare a struct of individual  
//initialization  
student1.age=23;  
student1.weight=62.3;  
student1.name="arnold";  
  
individual student2; //declare a struct of individual  
//initialization  
student2.age=24;  
student2.weight=65.6;  
student2.name="georges";  
  
//create a template of individual  
vector<individual> vind;  
//fill it  
vind.push_back(student1);  
vind.push_back(student2);  
  
//write name of individuals  
for(int i=0;i<vind.size();i++) cout<<vind[i].name<<endl;
```

# Oriented Object

- What is an object ? (Ex: vehicle)
  - ✚ Defined by its properties (Ex: *number of wheels, ...*)
  - ✚ Defined by its actions (Ex: *driving, honking,...*)
  - ✚ Objects can interact (Ex: blinking)
  - ✚ We can have many objects of the same **type** (*instances*)
  - ✚ We can have different category of object “inheriting” from the same mother category (Ex: *motorcycle, car, bus, truck ...*)
  - ✚ Object can interact with object from another category (Ex: *driver, ...*)
- Definition of an object is already an “abstract concept”
- Oriented Object Programming is a powerful tool that allows things that might be difficult to implement with a procedural language

# Class

- Represent directly a “concept” in a program
  - Ex: vector, matrix, string, picture, histogram, particle, detector,...
- It is a user-defined type that specifies how objects of its type can be created and used (and deleted)
- Classes are key building blocks for large programs

# Minimal class

## *point\_2D.h*

```
#ifndef POINT_2D
#define POINT_2D

//all necessary include
#include <iostream>

//Class that describe points in a 2 dim. space

class point_2D{

    //--- List of attributes (data members)
    private: //only accessible from the methods
        //no need to right 'private', it's by default

        //coordinates of the point (cart.)
        double x_; // cannot be initialized here ! (in constructors)
        double y_;

    //--- List of methods

public:
    //default constructor
    point_2D();
    // the following line would also does the implementation
    // point_2D(){ x_=0.0 ; y_=0.0; }
    // the function will then be "inline"

    //default destructor
    ~point_2D();
    // the following line would also does the implementation
    // ~point_2D(){}
};

#endif
```

## *point\_2D.cpp*

```
#include "point_2D.h"

//default constructor
point_2D::point_2D(){
    x_ = 0.0;
    y_ = 0.0;
}

//similar implementation
point_2D::point2_D():x_(0.0),y_(0.0){}

//default destructor
point_2D::~point_2D(){}
```

# Minimal class

## Compilation:

#creating a point\_2D.o (compiled code)  
g++ -c point\_2D.cpp

#creating an executable

g++ -l. -o main.exe point\_2D.o main.cpp  
# -l. Is needed to be able to access point\_2D.h  
# -o is needed if you want to specify the name of your executable (a.out by default)  
# code is “linked” to point\_2D.o

```
#include <iostream>
#include "point_2D.h"

using std::cout;
using std::endl;

int main(){

    point_2D a; // instantiate an point_2D object
    point_2D* b = new point_2D(); // create an pointeron point_2D and allocate it dynamically

    return 0;
}
```

**main.cpp**

# Struct vs class

	struct	class
<b>Contain variables</b>	<b>Yes</b> Public	<b>Yes</b> Private by default
<b>Initialization</b>	Done “manually”	Done by ( <b>default</b> ) constructor
<b>Contain methods</b>	No Could define external functions, all public	<b>Yes</b> Possible private methods

# constructor & copy constructor

point\_2D.h

```
//other constructor  
point_2D(const double& x, const double& y);  
//copy constructor  
point_2D(const point_2D& point);
```

```
//Accessors  
//const prevent the implementation of the methods  
//to change the attributes  
double GetX()const {return x_};  
double GetY()const {return y_};
```

Coord a:0 0  
Coord b:1 2  
Coord c:1 2

point\_2D.cpp

```
void point_2D::point_2D(const double& x){  
    x_ = x;  
    y_ = y;  
}  
  
void point_2D::point_2D(const point_2D& p){  
    x_ = p.GetX();  
    y_ = p.GetY();  
}
```

main.cpp

```
#include <iostream>  
#include "point_2D.h"  
  
using std::cout;  
using std::endl;  
  
int main(){  
  
    point_2D a;  
    point_2D b(1.0,2.0);  
    point_2D c(b);  
  
    cout<<"Coord a:"<<a.GetX()<<" "<<a.GetY()<<endl;  
    cout<<"Coord b:"<<b.GetX()<<" "<<b.GetY()<<endl;  
    cout<<"Coord c:"<<c.GetX()<<" "<<c.GetY()<<endl;  
  
    return 0;  
}
```

# Operators overload

## Mathematical operators

- $+, -, *, /, \%$
- $+=, -=, *=, /=, \%=$

*point\_2D.h*

```
//overloading the operator +
//first argument should not be const as it will be changed
point_2D operator+(point_2D& a);
//overloading the operator +=
point_2D operator+=(const point_2D& a);
```

*point\_2D.cpp*

```
//overloading the operator +
point_2D point_2D::operator+(point_2D& a){
    point_2D c; // call the default constructor
    //additionne the value of X and Y
    c.SetX(GetX()+a.GetX());
    c.SetY(GetY()+a.GetY());
    return c;
}

//overloading the operator +=
point_2D point_2D::operator+=(point_2D const& a){
    //directly modify the data members x_ & y_
    x_ += a.GetX();
    y_ += a.GetY();
    return *this; //'this' is pointer to the current instance of the class
}
```

*main.cpp*

```
point_2D a;
point_2D b(1.0,2.0);
a+=b;

point_2D c;
c=a+b;
```

# Operators overload

## Comparison:

- == , !=
- >, >=,<,<=

point\_2D.h

```
//-- Comparison operators
bool operator==(const point_2D& a) const;
bool operator!=(const point_2D& a) const;
```

point\_2D.cpp

```
//overloading the operation ==
bool point_2D::operator==(const point_2D& a) const{
    //compare both x & y value
    if (GetX() == a.GetY() && GetY() == a.GetY()) return true;
    return false;
}

bool point_2D::operator!=(const point_2D& a) const{
    //we can reuse the already defined operator ==
    if(*this==a) return false;
    return true;
}
```

main.cpp

```
point_2D a;
point_2D b(1.0,2.0);

if(a==b) cout<<"Equality: OK" << endl;
else cout<<"Equality: NO" << endl;
if(a!=b) cout<<"Difference: OK" << endl;
else cout<<"Difference: NO" << endl;
```

# Operators overload

## Flux operators: <<, >>

point\_2D.h

```
//-- flux operators
friend ostream& operator<<(ostream &os, const point_2D& a);
```

point\_2D.cpp

```
ostream& operator<<(ostream &os, const point_2D& a) {
    //You just have access to public methods and public data members (none)
    //That's why we use the accessor GetX() and GetY()
    os << " ("<<a.GetX()<<","<<a.GetY()<<") ";
    return os;
}
```

main.cpp

```
point_2D a;
point_2D b(1.0,2.0);

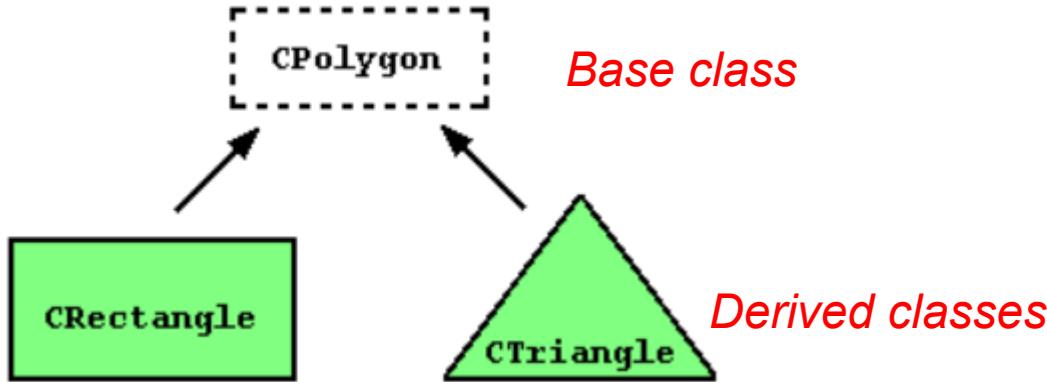
cout<<"point a:"<<a<<endl;
cout<<"point b:"<<b<<endl;
```

# Pointer & classes

- Pointer “*this*”: pointer to the current instance of the class
- Pointers to other classes:
  - Take care to the construction, copy constructor & destructor

# Inheritance

Example from <http://www.cplusplus.com>



Rectangle & triangle have common properties  
They are both polygons.

Access rights

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived class	yes	yes	no
not members	yes	no	no

## Rule of the “most restrictive access :

If **class Rectangle: protected Polygon**, the Public members of polygon would have been “protected” (not accessible) in Rectangle

In principle, a derived class inherits every member of a base class except:

- its constructors and its destructor
- its assignment operator members (operator=)
- its friends

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

class Polygon {
protected:
    int width, height;
public:
    void set_values (int a, int b)
    { width=a; height=b; }
};

class Rectangle: public Polygon {
public:
    int area ()
    { return width * height; }
};

class Triangle: public Polygon {
public:
    int area ()
    { return width * height / 2; }
};

int main () {
    Rectangle rect;
    Triangle trgl;
    rect.set_values (4,5);
    trgl.set_values (4,5);
    cout << rect.area() << '\n';
    cout << trgl.area() << '\n';
    return 0;
}
```

# Polymorphism

- One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class.
- Polymorphism is the art of taking advantage of this simple but powerful and versatile feature

```
int main () {
    Rectangle rect;
    Triangle trgl;
    Polygon * ppoly1 = &rect;
    Polygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    cout << rect.area() << '\n';
    cout << trgl.area() << '\n';
    return 0;
}
```

Only members inherited from Polygon can be accessed from ppoly1 & ppoly2 and not those of the derived class

If int area() had been defined in Polygon with a different implementation from the derived class:

*Rect.area() and ppoly->area() would have given different results !!*

Polymorphism might be useful by example to create a vector of pointer to polygon whatever are the derived class of objects

# Virtual methods

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

class Polygon {
protected:
    int width;
    int height;
public:
    Polygon(){};
    virtual ~Polygon(){};
    void set_values (int a, int b)
    { width=a; height=b; }
    virtual int area ()
    { return 0; }
};

class Rectangle: public Polygon {
public:
    Rectangle(){};
    ~Rectangle(){};
    int area ()
    { return width * height; }
};

class Triangle: public Polygon {
public:
    Triangle(){};
    ~Triangle(){};
    int area ()
    { return (width * height / 2); }
};
```

```
int main () {
    Rectangle rect;
    Triangle trgl;
    Polygon poly;
    Polygon * ppoly1 = &rect;
    Polygon * ppoly2 = &trgl;
    Polygon * ppoly3 = &poly;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly3->set_values (4,5);
    cout << ppoly1->area() << '\n';
    cout << ppoly2->area() << '\n';
    cout << ppoly3->area() << '\n';
    return 0;
}
```

Int area() method **can** be redefined in all derived classes  
ppoly1->area() refer to the method defined in Rectangle and  
not in Polygone !

This wouldn't have been the case if the methods would have  
not been virtual

The **destructor of the base class** (here Polygon), should be **virtual**. If not, the destructor of the base class will be called but not the one of the derived class, resulting in **resources leak** (for memory allocated in the derived class).

A class that declares or inherits a virtual function is called a *polymorphic class*

# Abstract base classes

- Can only be used for base classes allowed to have virtual member function without definition
- Those functions are called virtual functions definition is replaced by “=0”

Int area() **have** to be defined in all derived function inheriting from Polygon

```
// abstract class CPolygon
class Polygon {
protected:
    int width, height;
public:
    void set_values (int a, int b)
        { width=a; height=b; }
    virtual int area () =0;
};
```

- Classes that contain at least one pure virtual function are known as abstract base classes
- Abstract base classes cannot be used to instantiate objects but pointer of abstract base class is valid !

```
Polygon mypolygon; // not working if Polygon is abstract base class
```

# Polymorphism

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

class Polygon {
protected:
    int width, height;
public:
    Polygon (int a, int b) : width(a), height(b) {}
    virtual int area (void) =0;
    void printarea()
        { cout << this->area() << '\n'; }
};

class Rectangle: public Polygon {
public:
    Rectangle(int a,int b) : Polygon(a,b) {}
    int area()
        { return width*height; }
};

class Triangle: public Polygon {
public:
    Triangle(int a,int b) : Polygon(a,b) {}
    int area()
        { return width*height/2; }
};
```

```
int main ()
{
    Polygon * ppol1 = new Rectangle (4,5);
    Polygon * ppol2 = new Triangle (4,5);
    ppol1->printarea();
    ppol2->printarea();
    delete ppol1;
    delete ppol2;
    return 0;
}
```

Since Base's destructor is not virtual and b is a Base\* pointing to a Derived object, delete b has

**undefined behaviour.** In most implementations, the call to the destructor will be resolved like any non-virtual code, meaning that the destructor of the base class will be called but not the one of the derived

class, resulting in resources leak.

Those features can be applied to arrays of objects or dynamically allocated objects.

# Static members

- Static member variables only exist once in a program regardless of how many class objects are defined!
  - ↳ One way to think about it is that all objects of a class share the static variables.

```
class Something
{
private:
    static int s_nIDGenerator;
    int m_nID;
public:
    Something() { m_nID = s_nIDGenerator++; }

    int GetID() const { return m_nID; }
};

int Something::s_nIDGenerator = 1; // Initializer

int main()
{
    Something cFirst;
    Something cSecond;
    Something cThird;

    using namespace std;
    cout << cFirst.GetID() << endl;
    cout << cSecond.GetID() << endl;
    cout << cThird.GetID() << endl;
    return 0;
}
```

Initializer

1  
2  
3

# Static methods

- **static methods** are not attached to a particular object, they can be called directly by using the class name and the scope operator.
- Like static member variables, they can also be called through objects of the class type, though this is not recommended
- In the implementation of those functions: access to pointer this and to non static data members is forbidden

```
class Something
{
private:
    static int s_nIDGenerator;
    int m_nID;
public:
    Something() { m_nID = s_nIDGenerator++; }

    int GetID() const { return m_nID; }
    //Example of static function
    static int GetLatestID() { return s_nIDGenerator; }
};
```

```
t
1
2
3
Latest ID: 4
Latest ID: 4
```

```
int Something::s_nIDGenerator = 1;

int main()
{
    Something cFirst;
    Something cSecond;
    Something cThird;

    using namespace std;
    cout << cFirst.GetID() << endl;
    cout << cSecond.GetID() << endl;
    cout << cThird.GetID() << endl;

    cout << "Latest ID: " << Something::GetLatestID() << endl;
    // or
    cout << "Latest ID: " << cFirst.GetLatestID() << endl;
    return 0;
}
```

# enums

- Enumerated types are types that are defined with a set of custom identifiers ("enumerators"), as possible values.
- Objects of these enumerated types can take any of these enumerators as value
- Values are always assigned to an integer numerical equivalent internally, of which they become an alias.
  - If it is not specified otherwise, the integer value equivalent to the first possible value is 0, the equivalent to the second is 1

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};  
//One could also specify the integer value of each enumerator  
//enum colors_t {black=2, blue, green, cyan, red, purple, yellow, white}  
  
int main(){  
  
    colors_t mycolor;  
    mycolor = blue;  
    if(mycolor == green) cout<<"mycolor is green"<<endl;  
    else cout<<"my color is not green: "<<mycolor<<endl;  
  
    return 0 ;  
}
```

## Examples of applications:

- Colors
- Months
- Gender
- ...

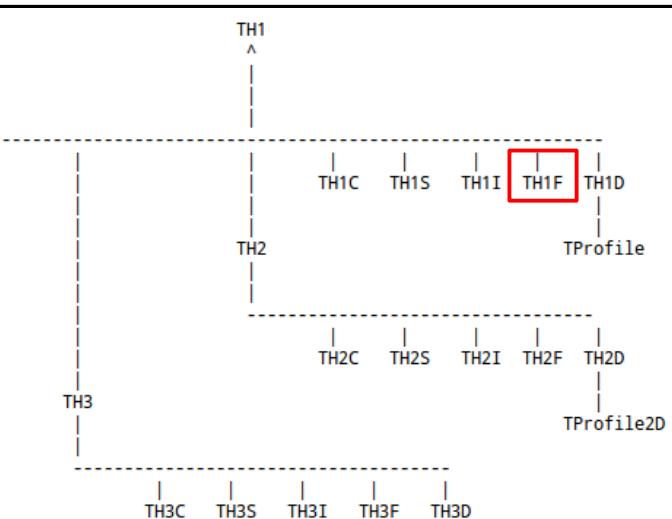
# enums can also be defined in class

```
class Display{
    public:
        enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
        Display(){}
        ~Display(){}
};

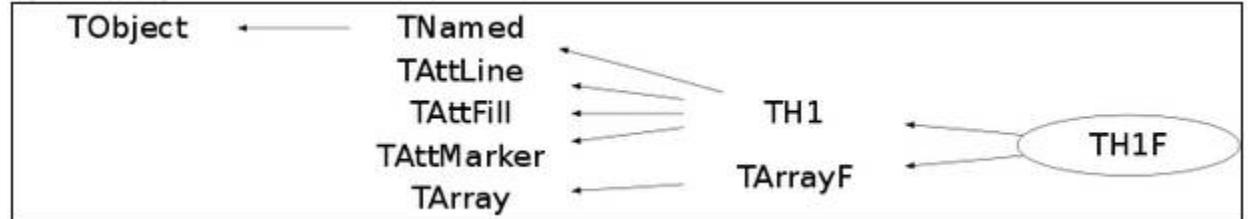
int main(){
    Display::colors_t mycolor;
    mycolor = Display::blue;
    if(mycolor == Display::green) cout<<"mycolor is green"<<endl;
    else cout<<"my color is not green "<<endl;

    return 0 ;
}
```

# Example with ROOT: TH1F



*multiple* inheritance



class TH1F: public **TH1**, public **TArrayF**

class TH1: public **TNamed**, public **TAttLine**, public **TAttFill**, public **TAttMarker**

*Short extract of the public 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 void TH1::Add(const TH1* h1, Double_t c1 = 1)
    virtual void TH1::Add(TF1* h1, Double_t c1 = 1, Option_t* option = "")
    virtual void TH1::Add(const TH1* h, const TH1* h2, Double_t c1 = 1, Double_t c2 = 1) MENU
        void TArrayF::AddAt(Float_t c, Int_t i)
    virtual void AddBinContent(Int_t bin)
```

# An example: TH1F

## Use of *enums*

```
public:  
    enum { kNoStats  
           kUserContour  
           kCanRebin  
           kLogX  
           kIsZoomed  
           kNoTitle  
           kIsAverage  
         };  
    enum TObject::EStatusBits { kCanDelete  
                               kMustCleanup  
                               kObjInCanvas  
                               kIsReferenced  
                               kHasUUID  
                               kCannotPick  
                               kNoContextMenu  
                               kIsValidObject  
                           };  
    enum TObject::EUnnamed { kIsOnHeap  
                            kNotDeleted  
                            kZombie  
                            kBitMask  
                            kSingleKey  
                            kOverwrite  
                            kWrtDelete  
                          };  
};
```

## Data members are *protected*

protected:	
Short_t	fBarOffset (1000*offset) for bar charts or legos
Short_t	fBarWidth (1000*width) for bar charts or legos
Double_t*	fBuffer [fBufferSize] entry buffer
Int_t	fBufferSize fBuffer size
TArrayD	fContour Array to display contour levels
Int_t	fDimension Histogram dimension (1, 2 or 3 dim)
TDirectory*	fDirectory fPointer to directory holding this histogram
Double_t	fEntries Number of entries
Color_t	TAttFill fFillColor fill area color
Style_t	TAttFill fFillStyle fill area style
TList*	fFunctions ->Pointer to list of functions (fits and user)
Double_t*	fIntegral Integral of bins used by GetRandom
Color_t	TAttLine fLineColor line color
Style_t	TAttLine fLineStyle line style
Width_t	TAttLine fLineWidth line width
Color_t	TAttMarker fMarkerColor Marker color index
Size_t	TAttMarker fMarkerSize Marker size
Style_t	TAttMarker fMarkerStyle Marker style
Double_t	fMaximum Maximum value for plotting
Double_t	fMinimum Minimum value for plotting
TString	TNamed fName object identifier
Int_t	fNcells number of bins(1D), cells (2D) +UOverflows
Double_t	fNormFactor Normalization factor
TString	fOption histogram options
TVirtualHistPainter*	fPainter fpointer to histogram painter
TArrayD	fSumw2 Array of sum of squares of weights
TString	TNamed fTitle object title
Double_t	fTsumw Total Sum of weights
Double_t	fTsumw2 Total Sum of squares of weights
Double_t	fTsumwx Total Sum of weight*X
Double_t	fTsumwx2 Total Sum of weight*X*X
TAxix	fXaxis X axis descriptor
TAxix	fYaxis Y axis descriptor
TAxix	fZaxis Z axis descriptor
static Bool_t	fgAddDirectory Flag to add histograms to the directory
static Int_t	fgBufferSize Default buffer size for automatic histograms
static Bool_t	fgDefaultSumw2 Flag to call TH1::Sumw2 automatically at histogram creation time
static Bool_t	fgStatOverflows Flag to use underoverflows in statistics

- **Variable are comments**
- Use “**rules**” for name
- Use of **static** variable
- **Pointers** are also used

# Another example: TLorentzVector

TLorentzVector is a general four-vector class, which can be used either for the description of position and time (x,y,z,t) or momentum and energy (px,py,pz,E).

## Inheritance

### Many constructors

```
TLorentzVector (const Double_t* carray)
TLorentzVector (const Float_t* carray)
TLorentzVector (const TLorentzVector& lorentzvector)
TLorentzVector (const TVector3& vector3, Double_t t)
TLorentzVector (Double_t x = 0.0, Double_t y = 0.0, Double_t z = 0.0, Double_t t = 0.0)
```

```
class TLorentzVector: public TObject
```

**Class:** TLorentzVector

**Header:** #include "TLorentzVector.h"

**Library:** libPhysics

### Overloaded operators

```
Bool_t operator!= (const TLorentzVector& q) const
Double_t operator() (int i) const
Double_t& operator() (int i)
TLorentzVector operator* (Double_t a) const
Double_t operator* (const TLorentzVector& q) const
TLorentzVector& operator*= (Double_t a)
TLorentzVector& operator*= (const TRotation& m)
TLorentzVector& operator*= (const TLorentzRotation&)
TLorentzVector operator+ (const TLorentzVector& q) const
TLorentzVector& operator+= (const TLorentzVector& q)
TLorentzVector operator- () const
TLorentzVector operator- (const TLorentzVector& q) const
TLorentzVector& operator-= (const TLorentzVector& q)
TLorentzVector& operator= (const TLorentzVector& q)
Bool_t operator== (const TLorentzVector& q) const
Double_t operator[] (int i) const
Double_t& operator[] (int i)
```

## Importance of the documentation

```
The Physics Vector package
=====
-* The Physics Vector package consists of five classes:
-* - TVector2
-* - TVector3
-* - TRotation
-* - TLorentzVector
-* - TLorentzRotation
-* It is a combination of CLHEPs Vector package written by
-* Leif Lonnblad, Andreas Nilsson and Evgeni Tcherniaev
-* and a ROOT package written by Pasha Murat.
-* for CLHEP see: http://wwwinfo.cern.ch/asd/lhc++/clhep/
-* Adaption to ROOT by Peter Malzacher
*
```

### TLorentzVector

TLorentzVector is a general four-vector class, which can be used either for the description of position and time (x,y,z,t) or momentum and energy (px,py,pz,E).

#### Declaration

TLorentzVector has been implemented as a set a TVector3 and a Double\_t variable. By default all components are initialized by zero.

```
TLorentzVector v1; // initialized by (0., 0., 0., 0.)
TLorentzVector v2(1., 1., 1., 1.);
TLorentzVector v3(v1);
TLorentzVector v4(TVector3(1., 2., 3.),4.);
```

For backward compatibility there are two constructors from an Double\_t and Float\_t C array.

# Type casting

## Type casting

```
double x = 10.3;
int y;
y = int (x);      // functional notation
//y = (int) x;    // c-like cast notation
cout<<"x = "<<x<<" y = "<<y<<endl;

int num = 1;
double z = 0;
z = num/3;
cout<<"z = "<<z<<endl;
z = double (num)/3;
cout<<"z = "<<z<<endl;
```

Can lead to code that while being syntactically correct can cause runtime errors or give undesired results

```
x = 10.3 y = 10
z = 0
z = 0.333333
```

## const\_cast

Manipulates the constness of the object pointed by a pointer, either to be set or to be removed

```
int main(){

const char* name = "Histogram";
//TH1F h(name,"title",10,0,10);
//first argument of TH1F construction is a char* and not a const char*
TH1F h(const_cast<char*> (name),"title",10,0,10);

return 0 ;
}
```

# Run-Time Type Information (RTTI)

```
// typeid, polymorphic class
#include <iostream>
#include <typeinfo> // Red box here
#include <exception>
using namespace std;

class Base { virtual void f(){} };
class Derived : public Base {};

int main () {
    try {
        Base* a = new Base;
        Base* b = new Derived;
        cout << "a is: " << typeid(a).name() << '\n';
        cout << "b is: " << typeid(b).name() << '\n';
        cout << "*a is: " << typeid(*a).name() << '\n';
        cout << "*b is: " << typeid(*b).name() << '\n';
    } catch (exception& e) { cout << "Exception: " << e.what() << '\n'; }
    return 0;
}
```

```
a is: class Base *
b is: class Base *
*a is: class Base
*b is: class Derived
```

- It can be applied on any build-in type or user-defined class
- **typeid** uses the **RTTI** to keep track of the type of dynamic objects.
  - When **typeid** is applied to an expression whose type is a polymorphic class, the result is the type of the *most derived complete object*

# Type casting

## dynamic\_cast

- Can be used with pointers and references to classes.
- Ensure that the result of the type conversion points to a valid complete object of the destination pointer type.
- Pointer is null (==0) if the cast failed

**Upcast:** converting from pointer-to-derived to pointer-to-base classes  
in the same way as allowed as an implicit conversion.

**Downcast:** converting from pointer-to-base to pointer-to-derived polymorphic classes  
if -and only if- the pointed object is a valid complete object of the target type.

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

class Base { virtual void dummy() {} };
class Derived: public Base { int a; };

int main () {
    Base * pba = new Derived;
    Base * pbb = new Base;
    Derived * pd;

    pd = dynamic_cast<Derived*>(pba);
    if (pd==0) cout << "Null pointer on first type-cast.\n";

    pd = dynamic_cast<Derived*>(pbb);
    if (pd==0) cout << "Null pointer on second type-cast.\n";
    return 0;
}
```

Null pointer on second type-cast.

# Friendship

## Friend methods

A non-member method can access the private and protected members of a class if it is declared a friend of that class

```
// friend functions
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle duplicate (const Rectangle&);
};

Rectangle duplicate (const Rectangle& param)
{
    Rectangle res;
    res.width = param.width*2;
    res.height = param.height*2;
    return res;
}

int main () {
    Rectangle foo;
    Rectangle bar (2,3);
    foo = duplicate (bar);
    cout << foo.area() << '\n';
    return 0;
}
```

24

# Friendship

## Friend class

An object from a class A can access the private and protected members of a class B if it is declared a friend of that class

```
// friend class
#include <iostream>
using namespace std;

class Square;

class Rectangle {
    int width, height;
public:
    int area ()
        {return (width * height);}
    void convert (Square a);
};

class Square {
    friend class Rectangle;
private:
    int side;
public:
    Square (int a) : side(a) {}
};

void Rectangle::convert (Square a) {
    width = a.side;
    height = a.side;
}

int main () {
    Rectangle rect;
    Square sqr (4);
    rect.convert(sqr);
    cout << rect.area();
    return 0;
}
```

16

class Rectangle is a friend of class Square  
BUT  
class Square is a friend of class Rectangle

Rectangle methods can access to square  
private/protected members

Be cautious while  
using friendship ...

# Function template

- Allow a **generalization** of a given “idea” to many different input **types**
  - Ex: you want to generalize the computation of a sum.  
Instead of having implementation for int, float, double ...  
you implement it once
- Compilation error** if you apply it to an **incorrect type**
  - Ex: *operator +* is not defined for all user-defined type
  - Ex: *%* is define for int but not for double

```
// function template
#include <iostream>
using namespace std;

template <class T> // T: template parameter name
T sum (T a, T b)
{
    T result;
    result = a + b;
    return result;
}

int main ()
{
    int i=5, j=6, k;
    double f=2.0, g=0.5, h;
    k=sum<int>(i,j); // Explicit specification of type T
    h=sum<double>(f,g);
    cout << k << '\n';
    cout << h << '\n';
    return 0;
}
```

```
int sum (int a, int b)
{
    return a+b;
}

double sum (double a, double b)
{
    return a+b;
}
```

```
// function templates
#include <iostream>
using namespace std;

template <class T, class U>
bool are_equal (T a, U b)
{
    return (a==b);
}

int main ()
{
    if (are_equal(10,10.0))
        cout << "x and y are equal\n";
    else
        cout << "x and y are not equal\n";
    return 0;
}
```

- Implicit specification is possible if unambiguous
  - Ex: *Sum(2,3)*
- Template with many several template types is possible

# Class templates

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
public:
    mypair (T first, T second)
        {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
    T retval;
    retval = a>b? a : b;
    return retval;
}

int main () {
    mypair <int> myobject (100, 75);
    cout << myobject.getmax();
    return 0;
}
```

“Famous” examples from the STL

**std::vector**

---

```
template < class T, class Alloc = allocator<T> > class vector; // generic template
```

**std::map**

---

```
template < class Key,
          class T,
          class Compare = less<Key>,
          class Alloc = allocator<pair<const Key,T> >
          > class map;
```

---

Template specialization possible

Ex: `vector<bool>`

Everything should be “rewritten”

# Exceptions

- Exceptions provide a way to react to exceptional circumstances (like runtime errors)
- Protect parts of the code
- Return an error message & decide what to do (abort the program ?)

```
double a = 12;
double b = 0;
double c = 0;

try{
    if(b==0) throw(0);
    c = a/b;
}
catch(int e){
    if(e==0) cerr<<"Division by 0 does not work !"<<endl;
    else cerr<<"Division failed !"<<endl;
    c=-1;
}
cout<<"c = "<<-1<<endl;
int i=0;
try{
    cout<<"Enter a number low than 100"<<endl;
    cin>>i;
    if(i>=100) throw i;
}
catch(...){ //default throw
    cerr<<"Too big number enter: File: "<<__FILE__<<" Line: "<<__LINE__<<endl;
}
```

```
Division by 0 does not work !
c = -1
Enter a number low than 100
123
Too big number enter: File: exceptions.cpp Line: 26
```

# Exceptions

```
#include <iostream>          // std::cerr
#include <typeinfo>           // operator typeid
#include <exception>          // std::exception

class Polymorphic {virtual void member(){}};

int main () {
    try
    {
        Polymorphic * pb = 0;
        typeid(*pb); // throws a bad_typeid exception
    }
    catch (std::exception& e)
    {
        std::cerr << "exception caught: " << e.what() << '\n';
    }
    return 0;
}
```

bad\_alloc      Exception thrown on failure allocating memory (class )  
bad\_cast      Exception thrown on failure to dynamic cast (class )  
bad\_exception      Exception thrown by unexpected handler (class )  
bad\_function\_call (C++)      Exception thrown on bad call (class )  
bad\_typeid      Exception thrown on typeid of null pointer (class )  
bad\_weak\_ptr (C++)      Bad weak pointer (class )  
ios\_base::failure      Base class for stream exceptions (public member class )  
logic\_error      Logic error exception (class )  
runtime\_error      Runtime error exception (class )

Indirectly (through logic\_error):

domain\_error      Domain error exception (class )  
future\_error (C++)      Future error exception (class )  
invalid\_argument      Invalid argument exception (class )  
length\_error      Length error exception (class )  
out\_of\_range      Out-of-range exception (class )

Indirectly (through runtime\_error):

overflow\_error      Overflow error exception (class )  
range\_error      Range error exception (class )  
system\_error (C++)      System error exception (class )  
underflow\_error      Underflow error exception (class )

Indirectly (through bad\_alloc):

bad\_array\_new\_length (C++)      Exception on bad array length (class )

exception caught: St10bad\_typeid

```
// using standard exceptions
#include <iostream>
#include <exception>
using namespace std;

class myexception: public exception
{
    virtual const char* what() const throw()
    {
        return "My exception happened";
    }
} myex;

int main () {
    try
    {
        throw myex;
    }
    catch (exception& e)
    {
        cout << e.what() << '\n';
    }
    return 0;
}
```

My exception happened.

- “standard” exceptions already managed
- Possibility to create your own class inheriting from std::exception (see example above)

# Basic programming rules

- Indentation of the code (*more readable ..*)
- Respect conventions for the variable name (*and even more generally*)
- Always initialize variables
- Be cautious with
  - Integer division
  - Type Casting
  - Usage of array
  - Dynamical allocation & delete
- Comments
- Documentation ( *possibility to use tools such as “Doxygen”*)
- Code protection and exceptions
  - Test on variables, pointers ...

# Computation

Developer goal is to express computation

- Correctly // means code protection ...
- Simply // means use the appropriate variable name, syntax, functions, lib, ..
- Efficiently // different options (cpu, memory, fiability, ...)

Organization of the code (cf UML)

- Divide big computations into many little ones (functions, classes)
- Avoid duplication
- Abstraction: provide a higher level concept that hides details

Usability:

- User-friendly: documentation, comments, abstraction

Organization of the data:

- |   |  |
|---|--|
| <ul style="list-style-type: none"><li>• Input/output format</li><li>• Protocols: how it communicates</li><li>• Data structure</li></ul> | <ul style="list-style-type: none"><li>• User build-in types, if not sufficient use library types and if it doesn't fit your goals, define your own class</li><li>• For your computation, check if existing libraries does not fit your needs</li></ul> |
|---|--|

**And in all cases, don't reinvent the wheel**

# Programming

- First step is the conception
- Start with a simple & robust implementation
- Than perform intensive tests:
  - ✚ Should produce the desired results for all legal inputs
  - ✚ Should give a reasonable error messages for illegal inputs
- Review your code
  - ✚ Code cleaning: remove useless variables, ...
  - ✚ Style: Comments / naming & coding rules / documentation
  - ✚ Maintenance: use of functions / parameters instead of “magic numbers” ...
- Let a colleague review your code
- Only then, add features. Go to a “full scale” solution based on 1st impl.
  - ✚ It will avoid problems, delay, bugs, ...
- Code can be used by users in a largest community ...

# UML: Unified Modeling Language

- General to all oriented object language
- It is a modelization language
- Allow to deal with complexity
- It can be a first step (conception) before implementation of the code
- Guidance: OMG UML: <http://www.omg.org>
- Will be more discussed during the computing sessions

# Optimization

- Having a code properly functioning is clearly the first and most important feature !
- But in many application, memory or CPU-cunsumption might be a bottleneck. In that cases, optimization would be required.
- Control of execution time (`<ctime>`, `<chrono>`, or even a simple time `./a.out`)
- Even if it is a whole topic by itself, this is few basic direction to follow
  - Prefer reference to pointer
  - Parameters might be reference (no copy)
  - Take advantage of stl containers
  - Avoid extensive usage of RTTI & Exception handling
  - Initialization is faster than assignment
  - Use inline functions in the most simple case
  - Compiler options can also help for optimization
  - ...