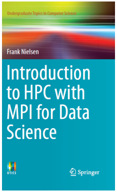
Introduction to HPC with MPI for Data Science

L1 : I. Introduction to High Performance Computing (HPC)
followed by
II. Introduction to C++ and Unix

Frank Nielsen

Frank.Nielsen@acm.org



https://franknielsen.github.io/HPC4DS/
https://www.springer.com/gp/book/9783319219028

The **objectives** of these lectures is to ...

1. **design** and **analyze** parallel algorithms on computer clusters ($\rightarrow$ distributed memory)
   Algorithms for Data Science

2. **implement** these algorithms in C++/STL with the standard and the library Message Passing Interface (MPI)

3. **debug** and **execute** these programs on machine clusters ($\rightarrow$ Unix, Shell + command lines)

Overview of the syllabus and hands-on sessions

8 blocks L_1 to L_8

- **programming in C++** with the *Standard Template Library* (**STL**)
- **program parallelization** with the *Message Passing Interface* (**MPI**), and key concepts of parallelism :
  → topologies, communications, collaborative computing, etc.
- **data analysis** on computer clusters :
  1. exploratory research (*clustering*)
  2. supervised learning (classification)
  3. linear algebra (linear regression)
  4. graphs (social network analysis)
- critical evaluation of results (Data Science) and performance analysis

First part :

Introduction to HPC

What is High Performance Computing (HPC) ?

- HPC = Sciences of *supercomputers* (`http://www.top500.org/`)
  Top 1 : Sunway TaihuLight, National Supercomputing Center in Wuxi, China.
  125 PetaFLOPS (PFLOPS), 10+ millions of cores... and 15 Megawatts of power
  1 MW = 100 euros/hour or 1 million euros/year
- but green HPC also evaluates the performances in **MFlops/Watt**, `http://www.green500.org/`
- HPC = the **domain** including paradigms of parallel programming , programming languages, software tools, information systems, with dedicated conferences (ACM/IEEE Super Computing), etc.

In April 2016, top 5 supercomputers in the world...

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945

LINPACK benchmark : Rmax = maximal performance obtained
Rpeak = theoretical maximal performance.
`http://www.top500.org/project/top500_description/`

In April 2017, top 5 supercomputers in the world

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
2	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
3	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
4	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
5	DOE/SC/LBNL/NERSC United States	**Cori** - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect Cray Inc.	622,336	14,014.7	27,880.7	3,939
6	Joint Center for Advanced High Performance Computing Japan	**Oakforest-PACS** - PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path Fujitsu	556,104	13,554.6	24,913.5	2,719

Total in 2016 : Pangea SGI ICE X 6.7 PFlops (petascale)

storage = 26 petabytes ($\equiv$ 6 millions of DVDs)

Pangea - SGI ICE X, Xeon Xeon E5-2670/ E5-2680v3 12C 2.5GHz, Infiniband FDR

Site:	Total Exploration Production
Manufacturer:	HPE/SGI
Cores:	220,800
Linpack Performance (Rmax)	5,283.11 TFlop/s
Theoretical Peak (Rpeak)	6,712.32 TFlop/s
Nmax	4,919,040
Power:	4,150.00 kW (Submitted)
Memory:	54,000 GB
Processor:	Xeon E5-2680v3 12C 2.5GHz
Interconnect:	Infiniband FDR
Operating System:	SUSE Linux Enterprise Server 11
Compiler:	N/A
Math Library:	Intel MKL
MPI:	SGI MPT

$\leftarrow$ Numerous applications (simulations)

Nowadays, it is easy to rent a low price HPC unit from cloud computing services such as AMZ AWS, MS Azure, etc.

Machine learning and Artificial Intelligence are the killer apps of High Performance Computing

$\rightarrow$ Data Science

Today is the age of Petascale and tomorrow is that of Exascale

kiloFLOPS	10^3
megaFLOPS	10^6
gigaFLOPS	10^9
teraFLOPS	10^{12}
petaFLOPS (PFLOPS, petascale)	10^{15}
exaFLOPS (EFLOPS, éxascale)	10^{18}
zettaFLOPS	10^{21}
yottaFLOPS	10^{24}
...	...
googolFLOPS	10^{100}

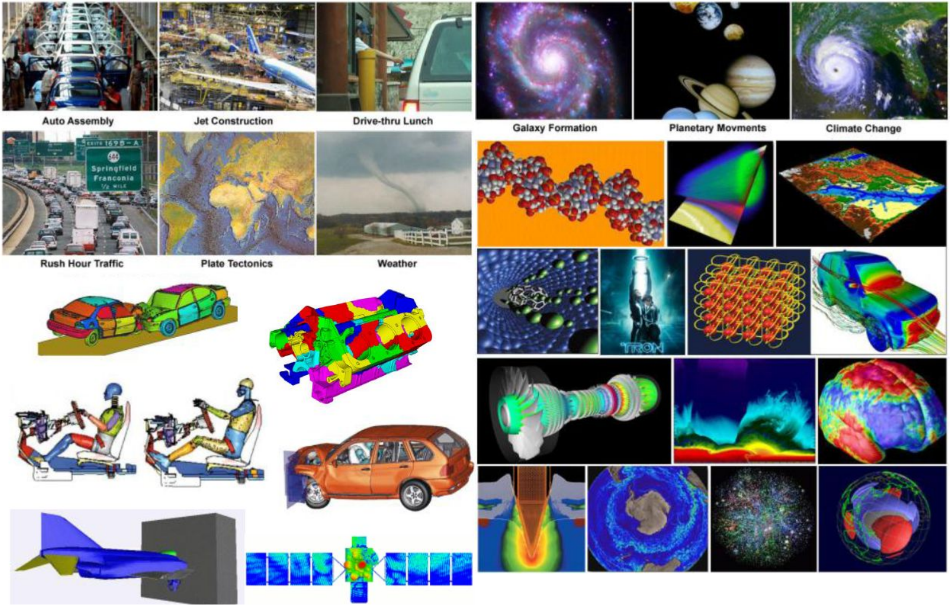
... but not only the computing power for supercomputers matters : memory (bytes), bandwidth of the network, etc.

Future : exaFlops (10^{18} in 2018-2020), zetaFlops (10^{21}) in 2030 ?
Specific Architectures for Deep Learning (TPU, etc.)

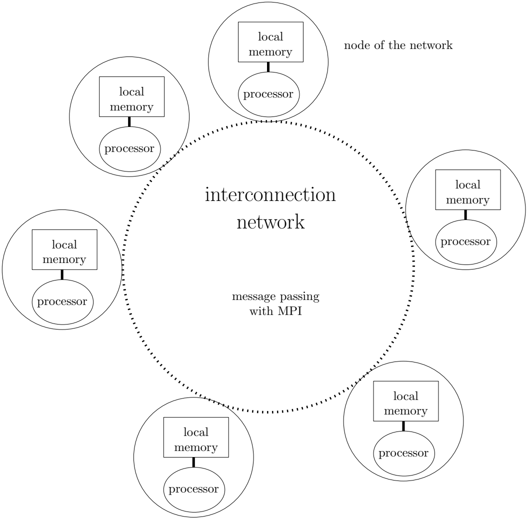
But why do we need HPC ? To be **more efficient** !

- Faster and more precise ! ($\rightarrow$ weather forecast)
- Solve complex problems ($\rightarrow$ simulation, $\rightarrow$ *big data*)
- **Save energy !** At same FLOPS power, use slower processors that consume less energy !
- **Simplify data processing** : some algorithms are *intrinsically parallel*
  video/image : filters `foreach pixel/voxel`, GPU & GPGPU
- Obtain the result **as fast as possible** *including development cost !* ($\rightarrow$ Business)
  easy-to-implement parallel algorithms rather than optimized sequential algorithms that are difficult to implement (by engineers). To have a final solution = implement an algorithm + execute this algorithm.

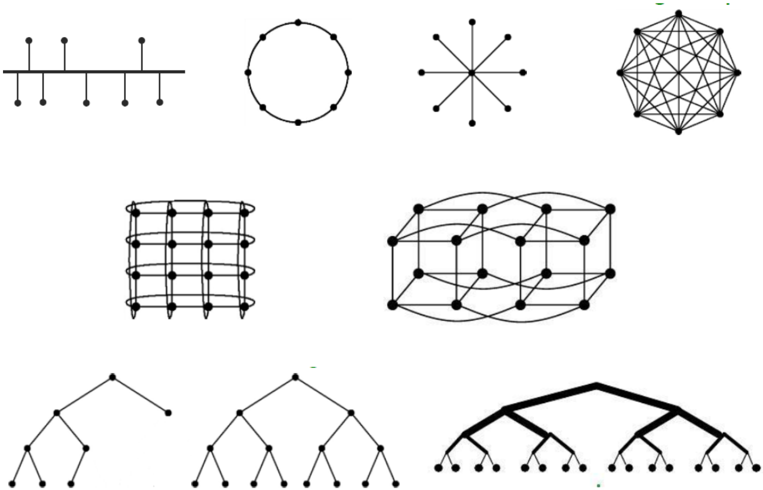
HPC illustrated



Architecture of a computer cluster



Topology of interconnection networks in a cluster

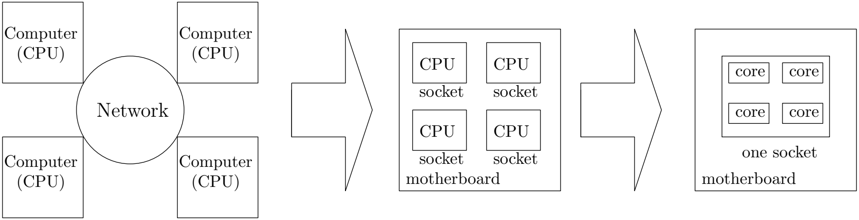


Physical/virtual topology is important for the design of parallel algorithms → Abstraction

How to broadcast data from one node to all other nodes?

Evolution of processors

From mono-processor architectures to multi-cores computers with **shared memory**



4 computers interconnected with a network   quad processor on a single board   quad core processor

But to scale up in High Performance Computing, we need to use computer clusters : **distributed memory** !

Ideal theoretical framework...

- Job : a process created by executing a program
- Manager : An administrator which assigns resources in the cluster to jobs (we shall use **SLURM**)

- Theoretical framework in this course for analyzing a parallel algorithm : **a process** P **runs on its own processor (a CPU mono-core) of a computer which is a node of the cluster.**
- In practice : heterogeneous computer clusters (multi-cores, with GPU). Multiple processes can be mapped by the administrator to a same processor (potentially in a same core)

HPC : granularity

granularity = proportion of computations (grains = local computations) with respect to the communications (inter-processes).

≡ Frequency of communications (or synchronization) between processes.

- **fine-grained** : many small jobs, data often transferred between processes after small computations (e.g., GPU).
  → well adapted to multi-cores architectures with shared memory

- **coarse-grained** : data are not exchanged regularly and only after big computations.
  → adapted to distributed memory clusters

  Extreme cases = embarrasingly parallel, very little communications.

Parallelism and concurrency

Two different notions in parallel computing :

Parallelism and concurrency :

- Parallelism : jobs executed literally in **the same time**,
  Physically, there are multiple computing units
- Concurrency : at least two jobs progressing *simultaneously in time*. Not necessarily in the same time.
  time-slicing on a same CPU, multi-task on a core
  For example, Windows™ with only one core : it seems that multiple applications executed in the same time but it is just an illusion !

Parallel programming models of nodes

- Vector programming model (SIMD, Cray)
- Distributed programming model : clusters
  exchanges of explicit messages $\rightarrow$ MPI
- Programming model with shared memory :
  multi-threading (OpenMP)

Big Data… 4V !

BigData = a buzzword widely advertised, hide may factors, (*large-scale*)

The 4 V on data :

- **V**olume (TB, PB, etc.)
- **V**ariety (heterogeneous)
- **V**elocity (data processed in real time, captors)
- **V**alue (not simulation but *valorization*)

Fault tolerance : a recurrent problem on clusters

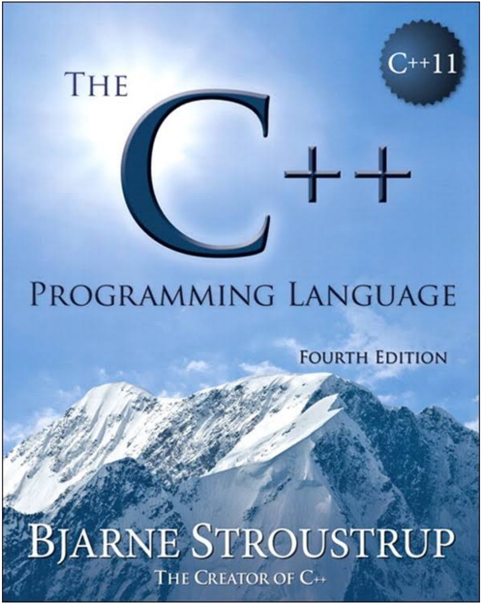
Fault tolerance of computers ?, networks ?, disks ?, etc. :

- MPI : **zero tolerance** but very easy to programming
- MapReduce C++ (or Java Hadoop) = programming paradigm : **high fault tolerance** but very limited computing model

We can do MapReduce (progamming model //) with MPI
"Towards efficient mapreduce using MPI," European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting. Springer Berlin Heidelberg, 2009

Some fallacies on distributed systems !

1. The network is reliable
2. **Zero latency**
3. **The bandwidth is infinite**
4. The network is sure
5. The network topology does not change
6. There is only one network administrator
7. Transportation cost is zero
8. The network is homogeneous



Successor of *C* ($\sim$ 1970), C+1 = C++ ! (1983)

An object-oriented (OO) language C++

- created by Bjarne Stroustrup in 1983
- **Object-Oriented** (OO) with static typing.
  $\rightarrow$ influence Java and other derives of C ($\approx$ 1970)
- Code is compiled **quickly** ($\neq$ Python interpreted), without virtual machines ($\neq$ Java with JVM)
- We need to manage the memory ourself : without Garbage Collector, GC. Pay attention to errors during execution (*system crash*, *segmentation fault*, *core dumped*)
- Passing by values, pointers or references ($\neq$ Java : passing by value or by reference for objects)
- File extensions : `.cc` `.cpp` `.cxx` `.c++` `.h` `.hh` `.hpp` `.hxx` `.h++`
- Use g++ (*GNU Compiler Collection*) of GNU

Compilator C++ (GNU)

Standards (ANSI C++, C++11, etc.) and other compilators
`https://gcc.gnu.org/`

```
[france ~]$ g++ --version
g++ (GCC) 4.1.2 20080704 (Red Hat 4.1.2-55)
Copyright (C) 2006 Free Software Foundation, Inc.
This is free software; see the source for copying
   conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS
   FOR A PARTICULAR PURPOSE.
```

⇒ exist many versions of g++ (C++98, C++11 ; etc.)
STL (Standard Template Library) by default in C++98

Compilators online : `http://cpp.sh/`, etc.
Install MinGW to have g++ on Windows

Plan

1. First program in C++

Welcome to C++

```
/* First program with
a comment on two lines */
// for the inputs and outputs (I/Os) :
#include <iostream>

int main()
{
  std::cout << "Welcome to INF442\n";
    return 0; // not mandatory
}
```

We compile with g++ :

```
console> g++ bienvenue.cpp -o bienvenue
console>bienvenue
Welcome to INF442
```

cout = short for c(onsole) out

Welcome to C++

```
#include <iostream>

// to avoid the need for writing std :: multiple times
using namespace std;

int promo=15;

int main()
{
  cout << "Welcome to  C++"<<promo<<endl;
    /* cout = Standard output stream
    we write in the flow cout with <<
    */
}
```

Plan

2. Inputs and outputs in C++

Welcome to C++ : inputs and outputs

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



int main()
{int x;
  cout << "Enter an integer : ";
    cin >> x; // we read the integer to x
    cout << "Square of x is : "<<x*x<<endl;
}
```

And also `cerr` (console error) which **displays immediately** important (error) messages to the console ...

Welcome to C++ : inputs and outputs

```
#include <iostream>

int main(int argc, char **argv)
{
  std::cout << "Hello everyone " << argv[1] << std::
      endl;
  return 0;
}
```

```
console> g++ helloEveryone.cpp -o helloEveryone
console> helloEveryone Frank
```

We obtain on the console :

```
Hello everyone Frank
```

Read a string of charaters

```
#include <iostream>
#include <string>

int main(int argc, char **argv)
{ // declare a variable of type string
  std::string promo;
  std::cout << "Enter the promotion : " << std::endl;
  std::cin >> promo;

  std::cout << "Welcome the " << promo << "s" << std::
     endl;
  return 0;
}
```

Redirection inputs and outputs

In a file `Promo.txt` :

```
X15
```

Redirect the content of the file `Promo.txt` to the program thanks to '<' :

```
console> helloEveryone  < Promo.txt
Welcome the X15s
```

Plan

3. Classes and objects

Objects and methods in C++

Be careful, we need to put a ; after the declaration of a class ($\neq$ Java)

```
class Boite
{public: // we put public to allow exterior access
      double horizontal;    // field object : width
      double vertical;   /* field object : height */
};

int main( )
{  Boite B1, B2;
   double surface = 0.0;
      // access to member with '.'
   B1.horizontal = 5.0;  B1.vertical = 6.0;
   surface = B1.horizontal * B1.vertical;
   cout << "Area of the box B1 : " << surface <<endl;
   return 0;
}
```

Objects : constructor(s) and destructor ~ in C++

it is possible to have **multiple constructors** (with different signatures) but always **only one destructor**.

```
class Boite
{public:
      double horizontal;    // width
      double vertical;   /* height */

Boite (double h, double v);
Boite (double s);
// we use the destructor by default  Boite()
};

// The body of the constructor defined outside of the class
Boite::Boite (double h, double v)
{horizontal=h; vertical=v;}
Boite::Boite (double s)
{horizontal=vertical=s;}
```

Member functions and static functions

```
class Boite
{public:
      double horizontal;    // width
      double vertical;  /* height */

Boite (double h, double v);

// member function : use the field
double area(){return horizontal*vertical;}

// static function
static double area(double c1,double c2){return c1*c2;}
};
```

- A member function has access to variables of the class.
- A static function does not have access to variables of the class.

```
cout << "Area of the box B1 : " << B1.area()<<endl;
```

Plan

4. Memory : execution stack and heap

Stack and heap

- When we call a function in C++ (we call the function `main()` by default when we execute a program), the variables of the functions are stored in the execution stack.
- When a function finishes its execution, the corresponding memory in the stack is freed.
- Functions can store objects created in the global memory (heap, accessible by all functions) by using the key word `new`
- There is no GC (GC = Garbage Collector), we need to free the memory with the key word `delete`

recursive function

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

int factorial(int n)
{if (n==0) return 1; else return n*factorial(n-1);}

int main()
{
cout<<factorial(10);        // 3628800
}
```

Everything is ok for 10! but pay attention to overflow : Integers have only a limited precision (on 32-bit or 64-bit architectures)

Recursive function and execution stack

```
int PlusBeaucoup(int x)
{
int tmp; // a variable for nothing, it will disappear in the optimized code
    or warning
return PlusBeaucoup(x+1);
}

int main( )
{
    PlusBeaucoup(442);
    return 0;
}
```

What happens ?

No terminal case for this recursion : It terminates abnormally when the execution stack becomes full !

Objects and local memory (stack)

```
Boite agranditBoite(Boite B, double dH, double dV)
{
// The object Boite stored in res is local (since there is no new)
Boite res=Boite(B.horizontal+dH,B.vertical+dV);
// we return the object
return res;}

int main( )
{  Boite B1(5,6);

   // we get the object result in the object B2
   Boite B2=agranditBoite(B1,1,2);

    cout<<B2.horizontal<<"x"<<B2.vertical<<endl;
   return 0;}
```

Explain what happens in the code!

Objects and global memory (heap)

- We define a variable pointer object `res` of type `Boite*`.
- We access to fields of a variable pointer object by `->`

```
Boite* agranditBoite(Boite B, double dH, double dV)
{
// Here we create the object in the global memory, the heap, with new
Boite* res=new Boite(B.horizontal+dH,B.vertical+dV);

// we return pointer
return res;}

int main( )
{  Boite B1(5,6);
   Boite* B2=agranditBoite(B1,1,2);

    cout<<B2->horizontal<<"x"<<B2->vertical<<endl;
    delete B2;
   return 0;}
```

Summary on objects

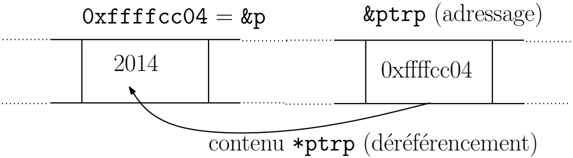
- a class contains *members variables* and *members functions/procedures* (procedure = function which returns nothing, the type `void`)
- for creating an object in the stack, we don't use `new`
- for creating an object in the heap (global memory), we use `new`
  Do not forget to `delete` the object when we don't use it anymore !
- a member static function never has access to member data of the object

Plan

5. Pointers

Random access memory : the ribbon memory and pointers

```
int p=2014;
int * ptrp = &p; // declare a pointer on p
cout<<"address of the cell of p :"<<ptrp<<endl;
(*ptrp) = p+3; // we modify the content of the cell
cout<<p<<endl; // we get 2017!
```



$\&p$: get the address of p

$*p$: dereferencing, we access to the content of p

(the content itself can be a memory address)

Pointers in C++ and variable typing

- Declaration of variable pointers :

```
int * ptr_entier, *ptr1, *ptr2;
char * ptr_caractere;
double * ptr_real;
```

- Referencing operator (Getting the address) : &

```
int var=1;
int *var2; // pointer to a variable of type integer
var2=&var1; // var2 points to var1
```

- Dereferencing operator : *

```
/* Take an integer in the cell referenced by var2 */
int var3=(*var2); // we dereference var2
```

C++ : pointers in action !

```
#include <iostream>
using namespace std;
int main ()
{
  int var1=442;
  int *var2;
   var2=&var1; // var2 points to var1
   cout<<"value of var2 : "<<var2<<endl;
  int var3=(*var2); // we dereference
   cout<<"value of var3 : "<<var3<<endl;
   return 0; // terminate without problems - :)
}
```

```
console> g++ program.cpp -o monprogram.exe
console> monprogram.exe
```

```
value of var2 : 0x7a30f960c59c
value of var3 : 442
```

Why do we need to manipulate pointers ?

pointer = <u>typed variable</u> which saves the address of another variable.
value of a pointer = memory address

```
int var1 =442; var2 = 2015;
int * Ptr1 , * Ptr2 ;
Ptr1 = &var1 ; Ptr2 = &var2 ;
```

Facilitate the implementation of dynamic data structures
→ linked list, trees, graphs, etc.

In C++/C, pointers allow :

- allocate memory for a variable and return a pointer to this memory area
- access to the value of the variable by dereferencing : `*Ptr1`
- free manually the memory

* : dereferencing operator = “ value pointed by ”

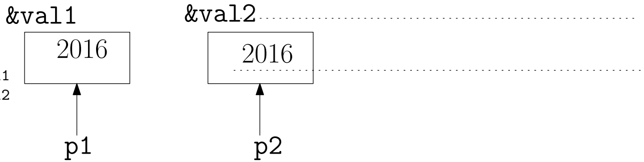
References and alias

```
int val1=442;
int val2=2017;

// alias
int & refVal1=val1;

cout<< refVal1 <<endl; //442
refVal1=val2;
// below, the alias phenomenon
cout<< val1 <<endl; //2017
```

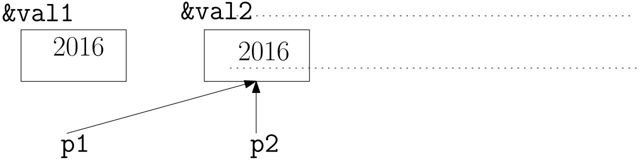
```cpp
#include <iostream>
using namespace std;

int main () {
 int val1 = 2015, val2 = 442;
 int * p1, * p2;
 p1 = &val1; // p1 = address of val1
 p2 = &val2; // p2 = address of val2
 *p1 = 2016; // value pointed by p1 = 2016
 *p2 = *p1; // value pointed by p2 = value pointed by p1
 p1 = p2; // p1 = p2 (value du pointer copiée)
 *p1 = 441; // value pointed by p1 = 441

 cout << "val1=" << val1 << endl;   // display 2016
 cout << "val2=" << val2 << endl; // display 441
 return 0;
}
```

Illustrations on next slides!

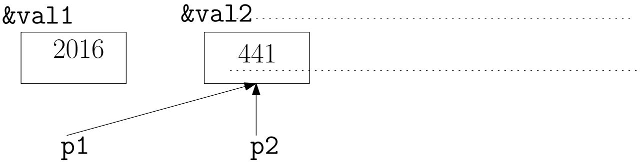
```
int val1 = 2015, val2 = 442;
int * p1, * p2;
p1 = &val1; // p1 = adresse de val1
p2 = &val2; // p2 = adresse de val2
*p1 = 2016;
*p2 = *p1;
```



```
p1 = p2;
```



```
*p1 = 441;
```



Pointers to pointers

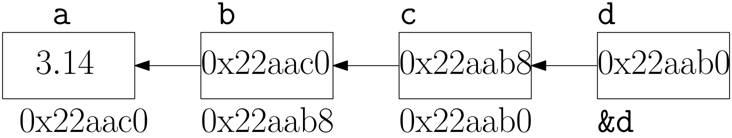
Reminder : pointer = typed variable whose value is the reference memory of another variable.

```
double a;
double* b;
double** c;
double*** d;

a=3.14159265;
b=&a;
c=&b;
d=&c;

cout<<b<<'\n'<<c<<endl<<d<<endl;
```

Illustration on the next slide!

Pointers of pointers

```
double a;
double* b;
double** c;
double*** d;

a=3.14;
b=&a;
c=&b;
d=&c;
```



Null pointer NULL

NULL=0

- useful in the recursive construction of data structures (lists, trees, graphs, sparse matrices, etc.)

- does not point to a valid reference or any memory address :
  ```
  double * ptr=NULL;
  ... else return new Noeud("feuille", NULL, NULL);
  ```

- pay attention to *segmentation faults* :
  ```
  T * ptr; ptr=mafunctionSuper442();
  cout<< (*T)<<endl;
  // can explode if T=NULL or if T points to a non-declared memory cell!
  ```

Pointers and references

- A reference is always definite, of a given type, and **never change**. No arithmetic for references or change of type.
- in C++, passing by **value** or by **reference** : If the value is a pointer, the function can change the content of the pointed memory cells, **pointer arguments stay unchanged**.
- Passing by reference **does not copy** the object to the stack of function calls :

```
int functionpassParRef( MaClasse& classeobject )
    {...}
```

6. Function calls and argument passing

Modes of argument passing : value or reference

The arguments of a function can be passed in three different ways :

- Passing by value : we **evaluate** the expression of the argument and **copy** its value to the stack.
- Passing by reference : we avoid copying to the stack the argument by giving only its **reference**. We manipulate the argument thanks to its reference, and so if the function change its value, these changes are kept after the function terminates.
- Passing by "pointer" (= by value of a memory address). It is a pass by value

Pass by value

```
int fois(double a, double b)
{return a*b;}

int main()
{// we evaluate the arguments and put the result to the stack
    cout<<fois(5+2-1,4/2.0+3)<<endl;        //30
}
```

Pass by value

```
int plusplus2(double a, double b)
{a=a+1; b=b+1;
    return a+b;}


int main()
{int a=2, b=3;
    cout<<plusplus2(a,b)<<endl;    //7
    /* a and b do not change their values since
       plusplus2 is
    a pass by value */
}
```

Pass by value of objects

```
// Passing by value :does not work
// B is copied to the stack

void DoubleDimension(Boite B)
{
B.horizontal*=2; B.vertical*=2;
}

int main( )
{   Boite B1(5,6);
    cout<<B1.horizontal<<"x"<<B1.vertical<<endl;

    DoubleDimension(B1);
     // pass by value, B1 does not change!
     // we copied the object B1 to the stack

    cout<<B1.horizontal<<"x"<<B1.vertical<<endl;
    return 0;}
```

Passing by reference

```
// we pass the argument by reference
void decrement(int& a)
{a--;}

int main()
{int a=443;
decrement(a);
cout<<a<<endl; // 442
return 0;
}
```

Passing by reference of objects

```
// pass by reference
// the reference of B is put to the stack
// We don't copy B to the stack

void DoubleDimension(Boite& B)
{
B.horizontal*=2; B.vertical*=2;
}
```

Passing by "pointer" = by value of the memory address

```
void decrement(int* a)
{(*a)--;}
// we change the content of a but its address does not change

int main()
{int a=443;
decrement(&a);
cout<<a<<endl;
return 0;
}
```

Passing by "pointer" of objects

```
// passing by pointer
// We don't copy B to the stack
void DoubleDimension(Boite* B)
{
B->horizontal*=2; B->vertical*=2;
}
```

Passing by "pointer" of objects

```
// passing by pointer
// We don't copy B to the stack

void DoubleDimension(Boite* B)
{
B->horizontal*=2; B->vertical*=2;
}
// we change the content of B
// but its address does not change
```

Passing by “pointer” of objects

```
// pass by pointer = pass by value of an address
// Does not work
// When we finish the procedure, the pointer on B does not change

void DoubleDimension(Boite* B)
{
    B=new Boite(2*B->horizontal,2*B->vertical);
}
```

We lost the memory space on the heap!

```
// argument passing with an unary operator
int plus442(int x)
{return x+442;}

void plus442val(int x)
{x=plus442(x);}

void plus442ref(int& x)
{x=plus442(x);}

void plus442ptr(int* x)
{(*x)=plus442(*x);}

int main()
{int x=1;
plus442val(x); cout<<x<<endl; //1
plus442ref(x); cout<<x<<endl; //443
plus442ptr(&x); cout<<x<<endl; //885
}
```

Passing by values and passing by references

```
void swap (int& x, int& y) // by reference
{ int temp = x;   x = y;    y = temp;}

void swapPtr (int* Ptr1, int* Ptr2) // Attention!
{int * Ptr; Ptr=Ptr1; Ptr1=Ptr2; Ptr2=Ptr;}

// We swap the content of the variables
void swapGoodPtr (int* x, int* y) // ok!
{ int temp = *x;   *x = *y;    *y = temp;}

int main ()
{
int a = 2, b = 3;
swap( a, b ); cout<<a<<" "<<b<<endl;// OK
a=2; b=3; int* Ptra =&a,* Ptrb =&b;
swapPtr( Ptra, Ptrb );
cout<<*Ptra<<" "<<*Ptrb<<endl; // non!
swapGoodPtr( Ptra, Ptrb );
cout<<*Ptra<<" "<<*Ptrb<<endl; // oui!
}
```

Plan

7. Arrays in C++

Arrays in C++ : static allocation

Indices begin at 0 as in Java, but we can not do `tab.length` !

We need to give the length of the array in argument of functions

```
int nombrePremiers [4] = { 2, 3, 5, 7 };
int baz [442] = { }; // values initialised to zero



// bidimensionnal array
int matrice [3][5]; // choose a convention : 3 lines 5 columns.

void procedure (int table[]) {}
```

Later, we will almost always use `vector` of STL which manages arrays in a dynamic way...

```
// Arrays and pointers : arithmetic of pointers

int main ()
{
 int tab[5];
 int * p;
 p = tab; *p = 10;
 p++; *p = 20;
 p = &tab[2]; *p = 30;
 // arithmetic of pointers!
 p = tab + 3; *p = 40;
 // arithmetic of dereferenced pointers!
 p = tab; *(p+4) = 50;

 for (int n=0; n<5; n++)
   cout << tab[n] << " ";
 return 0;}   // 10 20 30 40 50
```

Arrays : Dynamic allocation in C++

We have to manage memory space **ourself** in C++ (not as in Java !), and we must free the memory when we no longer use it .

```
int taille=2015;
int *tab;
tab=new int[taille];

// ... use this array then FREE it!

delete [] tab;
```

The type string : program MiroirTexte.cpp

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

string renverse(string txt)
{
string result="";
int n=txt.size();
for(int i=0;i<n;i++)
{result+=txt[n-1-i]; // concatenation of strings
}
return result; }

int main()
{
string msg="Ambulance";
cout<<msg<<endl;
cout<<renverse(msg)<<endl; // ecnalubmA
}
```

Overload of operators in C++ (here for string)

== (double equal) is overloaded for the type string

```
bool estCeUnPalindrome(string msg)
{return (msg==renverse(msg));}

int main()
{
string msg="mon nom";
cout<<estCeUnPalindrome(msg)<<endl;
msg="Cours";
cout<<estCeUnPalindrome(msg)<<endl;
}
```

Arrays of characters : the length must be given !

```
char * DNAdual(char *sequence, int n)
{char * result=new char[n];
int i;
for(i=0;i<n; i++)
{
if (sequence[i]=='A') result[i]='T';
if (sequence[i]=='T') result[i]='A';
if (sequence[i]=='C') result[i]='G';
if (sequence[i]=='G') result[i]='C';
}
return result;}
int main()
{ // ATCGATTGAGCTCTAGCG
char sequence[]={'A','T','C','G','A','T','T','G','A','
   G','C','T','C','T','A','G','C','G'};
char * brinComplementaire=DNAdual(sequence,n);
return 0;}
```

Arrays of characters : Length must be given !

```
void printLine(char *carray, int n)
{int i; for(i=0;i<n;i++) cout<<carray[i];
cout<<endl;}
char * ARNTranscription(char *sequence, int n)
{char * result=new char[n];
int i;
for(i=0;i<n;i++)
{if (sequence[i]=='T') result[i]='U'; else result[i]=
   sequence[i]; }
return result;   }
int main()
{ // ATCGATTGAGCTCTAGCG
char  sequence[]={'A','T','C','G','A','T','T','G','A','
   G','C','T','C','T','A','G','C','G'};
int n=18;
char * brinARN=brinARN=ARNTranscription(sequence,n);
printLine(brinARN,n);
return 0;}
```

Pointers and arrays : some remarks

The value of an array variable `tab` is the **memory address of its first element**

```
int     tab[442];
int  *  ptr;
```

The pointer `ptr` is a variable which stores a memory address of an `int` (4 bytes = 32 bits, on 32 bits architecture). Therefore we can do :

```
ptr=tab;
```

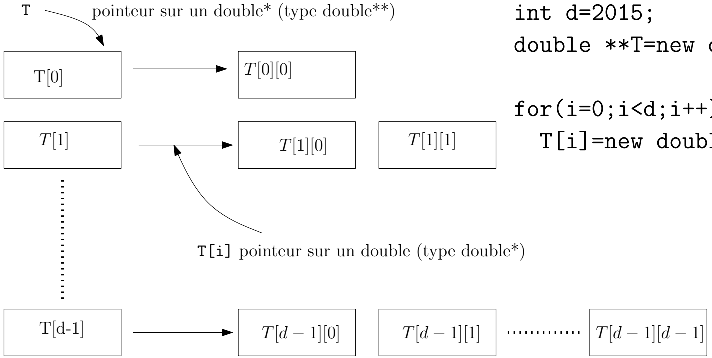
A static array is considered as a constant pointer.
it is therefore **not allowed** to do :

```
tab=ptr;  // not autorized
```

8. Multi-dimensional arrays in C++

Allocation of multi-dimensional arrays

```
int main(int argc, char *argv[])
{
double ** matriceTriangulaire;
int i,j, dimension=20;
// we have to create a 1D array of pointers of type double *
matriceTriangulaire=new double*[dimension];

// now we create lines
for(i=0;i<dimension;i++)
     matriceTriangulaire[i]=new double[dimension];

// matrice identite
for(i=0;i<dimension;i++)
     for(j=0;j<=i;j++)
        if (i==j) matriceTriangulaire[i][j]=1;
              else matriceTriangulaire[i][j]=0;
     ...
return 0;
```




```
int d=2015;
double **T=new dou

for(i=0;i<d;i++)
  T[i]=new double
```

Display of multi-dimensional arrays

We need to choose between the conventions line-column or column-line for the indices of the array.

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

int main(int argc, char *argv[])
{
double ** matriceTriangulaire;
int i,j, dimension=20;


...

    for(i=0;i<dimension;i++){
     for(j=0;j<=i;j++)
     {cout<<matriceTriangulaire[i][j]<<" ";}
    cout<<endl;
    }
...
```

The dangers of pointers : *dangling pointer*

A pointer which points to nothing = *dangling pointer*

```
int main ()
{ int * arrayPtr1;
int * arrayPtr2 = new int[442];

arrayPtr1 = arrayPtr2;
delete [] arrayPtr2;

cout << arrayPtr1[441];

 return 0;}
```

Many unexpected possible side effects : depends on the utilization history of the heap (*heap*)

The dangers of pointers : non-accessible zones

We may reserve memory zones which are no longer accessible :

```
int * Ptr1= 2015;
int * Ptr2 = 442;
Ptr1 = Ptr2;
```

Now imagine :

```
int * Ptr1= new int[2015];
int * Ptr2 = 442;
Ptr1 = Ptr2;
```

out of memory!

There exist some dynamic visualization tools for tracking the memory during the execution of programs. `http://valgrind.org/`

Plan of the course A1 en C++

1. First program in C++
2. Inputs and outputs in C++
3. Classes and objects
4. memory : execution stacks and heaps
5. Pointers
6. Function call and argument passing
7. Tables in C++
8. Multi-dimensional tables in C++

Summary

- HPC helps to be **more efficient** :
  faster, finer-grained simulations, larger amount of data, etc.
  We can simulate a parallel computer on a sequential machine but it is much more slower then !

- C++ is a compiling object-oriented language, built on C

- Unix is a multi-task operational system, written in C

Summary of key notions in C++

- understand *local memory* (stack) versus *global memory* (heap)
- passing by value, passing by reference of arguments (or passing by pointer)
- dynamic allocation (`new`) and manual management of memory (`delete`)
- classes and objects

Summary on pointers and references

& : reference operator = " address of "

* : dereference operator = " value pointed by "

- pointers : values = memory addresses. Save a reference on another variable.
  - pointers and arrays (→ constant pointers), pointers of pointers, ...
  - pointers `void` point on any type but can not be dereferenced (type casting)
  - pointers `NULL`
  - pointers and memory of heap : dangling pointers (unallocated memory → segmentation fault), no longer accessible (garbage)

- references : useful for passing of arguments to functions. No arithmetic for references, casting. A reference never changes and can not be `NULL`

Hands-on session 1 : Fundamentals of C++

Nothing can replace experience when programming !

- Multiple choice questions (5-15 minutes)
- Some Unix commands
- Hello world !
- Debug a palindrome program
- Swap by references
- Swap by pointers
- Transposition of matrices
- Multiplication of matrices

Practice for first hands-on session

- create a diagonal matrix
- print the matrix in output console
- create symmetric matrices

```
#include <iostream>
using namespace std;
// we don't the length of the diagonal
// we need to pass its length as an argument
double ** diagMat(int dim, double* diag)
{
int i,j;
double **res;

res=new double* [dim];
for(i=0;i<dim;i++)
        {res[i]=new double[dim];}

for(i=0;i<dim;i++)
    {for(j=0;j<dim;j++)
        {if (i==j) res[i][i]=diag[i];
                else res[i][j]=0;}
  }
return res;}
```

Procedure = function which does not return a result : (void)

```
void printMat(double **M, int dim)
{int i,j;
for(i=0;i<dim;i++)
{for(j=0;j<dim;j++)
     {cout<<M[i][j]<<"\t";}
cout<<endl;
}
}

int main()
{
  double diag[3]={1,2,3};
  double** Mdiag;

  Mdiag=diagMat(3,diag);   printMat(Mdiag,3);
  return 0;
}
```

A more geek version, not recommended, but we may find it in some codes...
← issue of C syntax

```
double ** diagMat(int dim, double* diag)
{
int i,j;
double **res;

// par default, les valeurs sont egales a zero
res=new double* [dim];
for(i=0;i<dim;i++)
        res[i]=new double[dim];

for(i=0;i<dim;i++)
    for(j=0;j<dim;j++)
    res[i][j]=( (i==j) ? diag[i] : 0);

return res;
}
```

```
#include <iostream>
// pour drand48(), inclure
#include <stdlib.h>
using namespace std;

double ** symMat(int dim)
{int i,j;
double **res;
res=new double* [dim];
for(i=0;i<dim;i++) res[i]=new double[dim];


for(i=0;i<dim;i++)
     for(j=0;j<=i;j++)
     {res[i][j]=drand48();res[j][i]=res[i][j];}

return res;
}
```

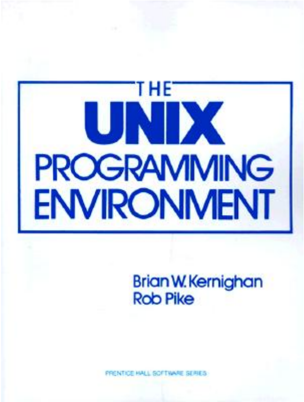
Not recommended but we can rewrite this code as below :

```
double ** symMat(int dim)
{int i,j;
double **res;
res=new double* [dim];
for(i=0;i<dim;i++) res[i]=new double[dim];


for(i=0;i<dim;i++)
    for(j=0;j<=i;j++)
    {res[i][j]=res[j][i]=drand48();
    // avant : res[i][j]=drand48();res[j][i]=res[i][j];
    }

return res;
}
```

A short introduction to Unix

UNIX

Unix is an operating system (OS) developed in the 1970s at Bell Labs of AT&T by Ken Thompson and Dennis Ritchie.



Some elementary commands of Unix

- Who am I? `id`

```
[france ~]$ id
uid=11234(frank.nielsen) gid=11000(profs) groups=11000(profs)
```

- List, rename and delete files : `ls`, `mv` (*move*) et `rm` (*remove*, option `-i` by default)
- Create a file or change its timestamps : `touch`
- Visualize and concatenate files : `more` et `cat`

```
more files
```

Elementary commands of Unix

Inputs/Outputs and pipe |

```
[france ~]$ cat fichier1.cpp fichier2.cpp |wc
     26       68     591
```

Access the manual :

```
[france ~]$ man wc
```

Redirections :

```
programme <input >output 2>error.log
```

Unix command : jobs)

- List all running processes (their numbers, pid) : ps (with options like ps -a)
- Suspend a process with Control-Z (Ctrl)

  sleep 10000
  Ctrl-Z

- Place a suspended job in process to the background :

  bg

- Kill processes or send signals to pids : kill

```
[france ~]$ sleep 5000 &
[1] 13728
[france ~]$ kill %1
[1]+  Terminated                  sleep 5000
```

Command `shell` (Unix)

- Open a window `shell` (in computer lab, shell = `bash`)
- Read the **initial configuration** file (= your file `.bashrc`) in your folder "home" (~).

```
more .bashrc
```

Modify it by using a text editor (`kate`, `nedit`, `vi`, `emacs`, ...)

Then read the configuration again at any moment in a session with :

```
source .bashrc
```

An example of .bashrc

For curiosity :

```
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi
# Prompt
PS1="[\h \W]\\$ "

alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
alias mm='/usr/local/openmpi-1.8.3/bin/mpic++ -I/usr/local/boost-1.56.0/include/
 -L/usr/local/boost-1.56.0/lib/ -lboost_mpi -lboost_serialization'

export PATH=/usr/lib/openmpi/1.4-gcc/bin:${PATH}
export PATH=/usr/local/boost-1.39.0/include/boost-1_39:${PATH}

LS_COLORS='di=0;35' ; export LS_COLORS
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/openmpi-1.8.3/lib/:/usr/local
/boost-1.56.0/lib/
```

Acknowledgments

- The initial translation of these slides from french to english was performed by Van-Huy Vo of École Polytechnique. Many thanks to him !
- When preparing the release of these english slides, I cleaned this translation a bit.
- Beware that this is not final release as some more translation work need to be done (in particular, in figures and codes)

On Internet

https://franknielsen.github.io/HPC4DS/