CERN School of Computing

Lecture 2

Scientific Programming: A Modern Approach

Danilo Piparo – CERN, EP-SFT

CERN School of Computing

This Lecture

The Goals:

1) Review C++ features usable for highly efficient and not-error-prone implementations of algorithms and data structures
2) Understand basic commonalities and differences of elementary data structures

CERN School of Computing

A Matter of Choices

- C++ has been chosen as the language for all examples and exercises
- Python will be considered too, for its conciseness, intuitiveness and because it can be easily interfaced with existing C++ libraries
- The principles illustrated throughout the lecture are of course also valid for programming in general!

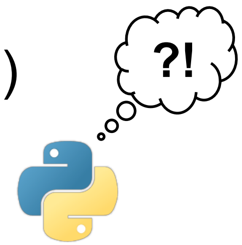
Wetting your Appetite

An incomplete selection of appealing, correctness and performance related C++ features

Vegan alternative

The `auto` keyword

- C++ is a “strongly typed” language
  - Type safety enforced (at least encouraged: casts are possible)
- The **auto** keyword: automatic type deduction
  - Improves readability and overall maintainability (correctness first)




```
[…]
int a = 5;
float b = 3.3f;
const char* c =
“my example\n”;
char *d = new char(‘c’);


[…]
```

```
namespace longName1{
  namespace longName2{
    class myClass{[…]};
  }
}
longName1::longName2::myClass
          createMyClassI(){[…]};

longName1::longName2::myClass inst1=
                              createMyClassI();
```

CERN School of Computing

The auto keyword

AAA Style
Almost Always Auto

- C++ is a “strongly typed” language
  - Type safety enforced (at least encouraged: casts are possible)
- The **auto** keyword: automatic type deduction
  - Improves readability and overall maintainability (correctness first)

?!

```
[…]
auto a = 5;
auto b = 3.3f;
auto c = “my example\n”;
auto d = new auto(‘c’);

auto error;
error = 5+3.;

[…]
```

Wrong! (typesafety)

```
namespace longName1{
  namespace longName2{
    class myClass{[…]};
  }
}
longName1::longName2::myClass
       createMyClassI(){[…]};

auto inst1 = createMyClassI();
```

CERN School of Computing

Range-based Loops

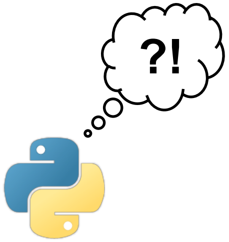
- Writing loops is fun again!
  - More concise and expressive (less mistakes possible)
  - **Uniform approach** with all collections offering a `begin()` and `end()` methods (`map`, `set`, `vector`, `myColl`,…)
- The compiler has all the information to put in place optimisations!

```
#include <map>
#include …
int main(){

std::map<int,std::string> myCont
               {{1,"one"},{2,"two"},{3,"three"}};

for (auto& p: myCont){
  std::cout << p.first << " -> "
            << p.second << std::endl;}
}
```

C++: initialiser list



Range-based Loops

- Writing loops is fun again!
  - More concise and expressive (less mistakes possible)
  - **Uniform approach** with all collections offering a `begin()` and `end()` methods (`map`, `set`, `vector`, `myColl`,...)
- The compiler has all the information to put in place optimisations!

Even more powerful parsing

Same initialisation!

Syntax for the loop: identical!

```
#include <vector>
#include …
int main(){

std::vector<std::pair<int,std::string>> myCont
              {{1,"one"},{2,"two"},{3,"three"}};

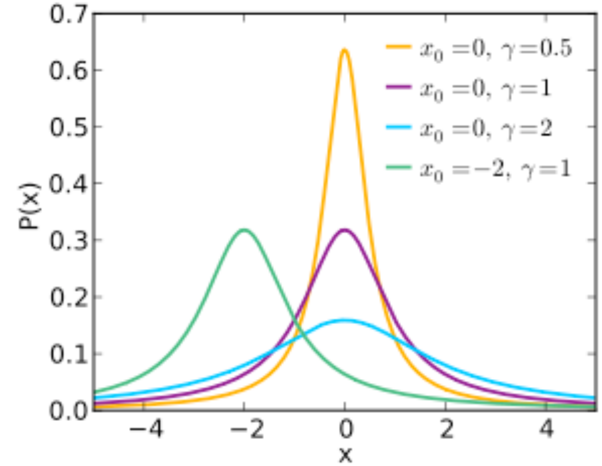
for (auto& p: myCont){
  std::cout << p.first << " -> "
            << p.second << std::endl;}
}
```

?!

Focus on the iteration and not on the “bureaucracy”

Random Generation

- Generating random numbers in C++ was cumbersome
  - Lots of external libraries: what if you can’t use them?
  - Use C `srand`? Normalisation? Period?
  - Distribution: hit or miss? What else?



A rich collection!

Just “#include <random>”!

Engines: Linear Congruential, Marsenne Twister, subtract with carry (Ranlux)

C++ names: mt19937, mt19937_64, ranlux_24, ranlux_48 …

Distributions (some of them): uniform, Bernulli, binomial, Poisson, normal, log-normal, Cauchy, …

Batteries Included!

Random Generation

```
#include <random>
#include <functional> // For std::bind
#include <iostream>

int main(){
  std::mt19937_64 myEngine;
  std::normal_distribution<float> myDistr(125.,12.);
  float oneNum = myDistr(myEngine);

  // Improve clarity!
  auto myGaussian = std::bind(myDistr,myEngine);

  for (int i=0;i<10;++i)
       std::cout << myGaussian() << std::endl;
}
```

Bind: yet another very expressive and handy construct!

- **No global states**: C++ random generation is **thread safe!**
- A huge improvement wrt `rand`, `srand` and `RAND_MAX`
- Pre-packed, well tested and standardised random number generation

CERN School of Computing

Lambda Functions

?!

λ

- An unnamed function inlined in code (also called "closures")
- Easy to pass as argument to other functions
  - Functor concept in C++03
- Composed by: capture specification **[…]**, argument list **(…)**, body **{…}**.
  - Last two: very well known already!
  - Capture specification: make available to the function variables from the scope in which the lambda is defined

```
int main() {
[] () {}; // empty lambda, a statement with no effect ☺
auto f1 = [](){std::cout << "Hello World!\n";};
f1();
auto f2 = [](const char* name){std::cout<<"Hello "<<name<<"!\n";};
f2("Bob");
}
```

Lambda Functions

```
int a=3;
auto f3 = [a](){return a*a;};// capture copy of "a" by value
auto f4 = [&a](){a*=a;};// capture reference to "a" by reference
auto f5 = [=](){…};// capture all vars in the scope by value
auto f6 = [&](){…};// capture all vars in the scope by reference
auto f7 = [=,&a](){…};// all vars by value, "a" by reference

// Create a vector and fill it w/ rndm numbers
std::vector<float> v(10);
std::generate(v.begin(),v.end(),myGaussian); // From the rand example!

float factor = 3.14;
std::for_each(v.begin(),v.end(),
              [factor](float x){return x*factor;});
```

- Concise, expressive: **a veritable work item** ← ***Crucial concept for the task parallelism***
- Extremely important when used with stl algorithms!

CERN School of Computing

Constexpr

- `Constexpr`: specifier for functions and variables
  - Meaning: **evaluate at compile time**!
  - Much more powerful than preprocessor macros
- Possible usecases: **tabulated values calculated once at compiletime**!

```
// Recursion again!
constexpr int factorial(int n) {
  return n <= 1 ? 1 : (n * factorial(n-1));
 }
```

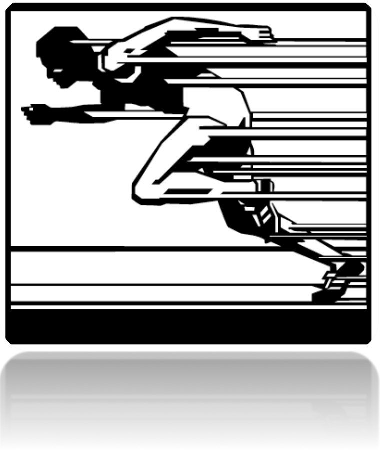
It could be done with templates, but not that readable!

```
// Max of two values
template <typename T>
constexpr T max(T a,T b){
  return a < b ? b : a;
 }
```

Constexpr: powerful tool to perform operations at compile time.

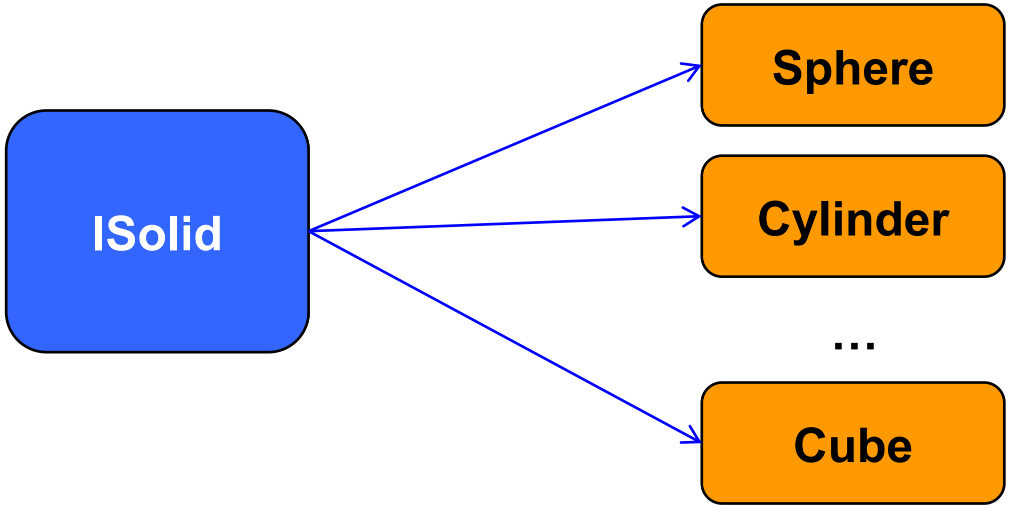
CERN School of Computing

Achieving Correctness and Good Performance



C++ and Inheritance

- Inheritance: one of the most powerful features of C++
  - Allow for maximum flexibility
  - Separation of interface and implementations: **clean code**
  - **Unified treatment of components behind the same interface**
- Comply to interfaces: easy mixing of components
  - E.g. Library developer provides interfaces, user complies to them when writing implementations



C++ and Inheritance

```
class ISolid{
public:
virtual bool IsInside(const Particle&) = 0;
virtual double DistanceToBoundary(const Particle&) = 0;
};
```

```
class Cube: public ISolid {
public:
bool IsInside(const Particle&){…};
double DistanceToBoundary(const Particle&){…}
};
```

```
class Sphere: public ISolid {
```

```
class Cylinder: public ISolid {
public:
bool IsInside…
double DistanceToBoundary…
};
```

Etc..

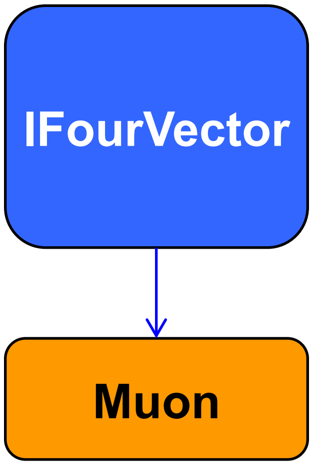
CERN School of Computing

C++ and Inheritance

Present in basically all existing codebases

- Virtual interfaces:
  - **Method to call decided at runtime!**
  - Have a sizeable price in terms of performance (~an additional function call per call)
    - Especially visible for small functions, tight loops …
- Indirection is present
  - Position of class subobject not known at compile time
  - Implemented with a vtable
- Can we do something about this?
  - Yes, there are several approaches ("devirtualisation")
  - One of them could be **using templates**
- Name of the game: avoid indirection

Less Then Optimal Practices

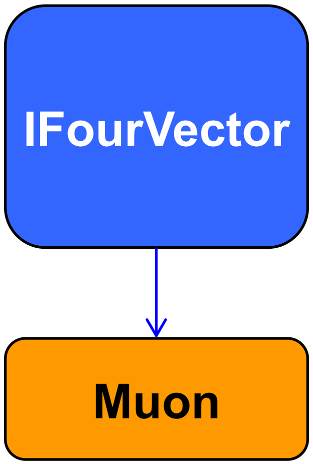


Provides virtual methods for getting Pt, Eta, Phi, …
Very general and clean right?
Remember the cost of indirections!!

```
for (auto const & particle : particles) {
    auto pt = particle.Pt();
}
```

How often in code this will happen?

Less Then Optimal Practices



Provides virtual methods for getting Pt, Eta, Phi, …
Very general and clean right?
Remember the cost of indirections!!

```
for (auto const & particle : particles) {
    auto pt = particle.Pt();
}
```

How often in code this will happen?

All the time!

CERN School of Computing

What's a template

- An **abstraction above the concept of classes and functions**
  - Example: std::vector<int>
- Templates: "family of classes/functions"
  - Create concrete entities **specialising** a "model" (the template) with data types, booleans or integers
- Objective: Re-use code
  - Generic programming: same code valid for all types
- New types, called "template instantiations" **created at compile time**
  - Catch mistakes early
  - Runtime budget unaltered
- **Can be used as alternative to runtime techniques**

What's a template

```
template <typename T>
class MyClass{
public:
  MyClass(T i):_i(i){};
  T& getI () const { return _i; }
private:
  T _i;
};
```

```
MyClass<int> myI(3);
MyClass<float> myF(3);
MyClass<double> myD(3);
[…]
```

What's a template

```
Class MyNonCopiable{
public:
[…]
MyNonCopiable(const MyNonCopiable &) = delete;
[…]
};
```

```
template <typename T>
class MyClass{
public:
  MyClass(T i):_i(i){};
  T& getI () const { return _i; }
private:
  T _i;
};
```

Error! It does not even compile

```
MyClass<int> myI(3);
MyClass<float> myF(3);
MyClass<double> myD(3);
[…]
MyNonCopiable a;
MyClass<MyNonCopiable> myNC(a);
```

Template Metaprogramming

- **Principle: move operations from runtime to compile time**
  - Can also gain performance!
- Can increase compile time (by very little, very affordable price anyway!)
- De facto, a veritable “language in the language”

```
template <typename T, int SIZE> class MyColl{
public:
  MyColl():_arr( new T(SIZE) ), _index(0){}
  void unsafePushBack(const T& v)
                                { _arr[_index++] = v; }
  T unsafeAt(unsigned int i){ return _arr[ i ]; }
  ~MyColl() { delete[] _arr;}
private:
  T* _arr;
  unsigned int _index; };
```

```
[…]
MyColl<float,5> a;
MyColl<MyColl<bool,3>,7> b;
[…]
```

Templates: powerful strategy to achieve **reusability and performance**

A Note

- Must we avoid virtual inheritance at all costs everywhere?
  - No.
- Use a grain of salt: understand what is the code you write in the design phase
  - Will the virtual methods be called often?
  - How much will be the performance penalty if at all?
  - Do the advantages of the abstraction outweigh the performance degradation, if any?

Interlude: Let the compiler Help you

Vegan alternative

Let the Compiler Help You

- Compiler technology is steadily evolving since years
  - Open source: two excellent competing products
  1) GCC: GNU Compiler Collection
  2) Clang: Based on LLVM

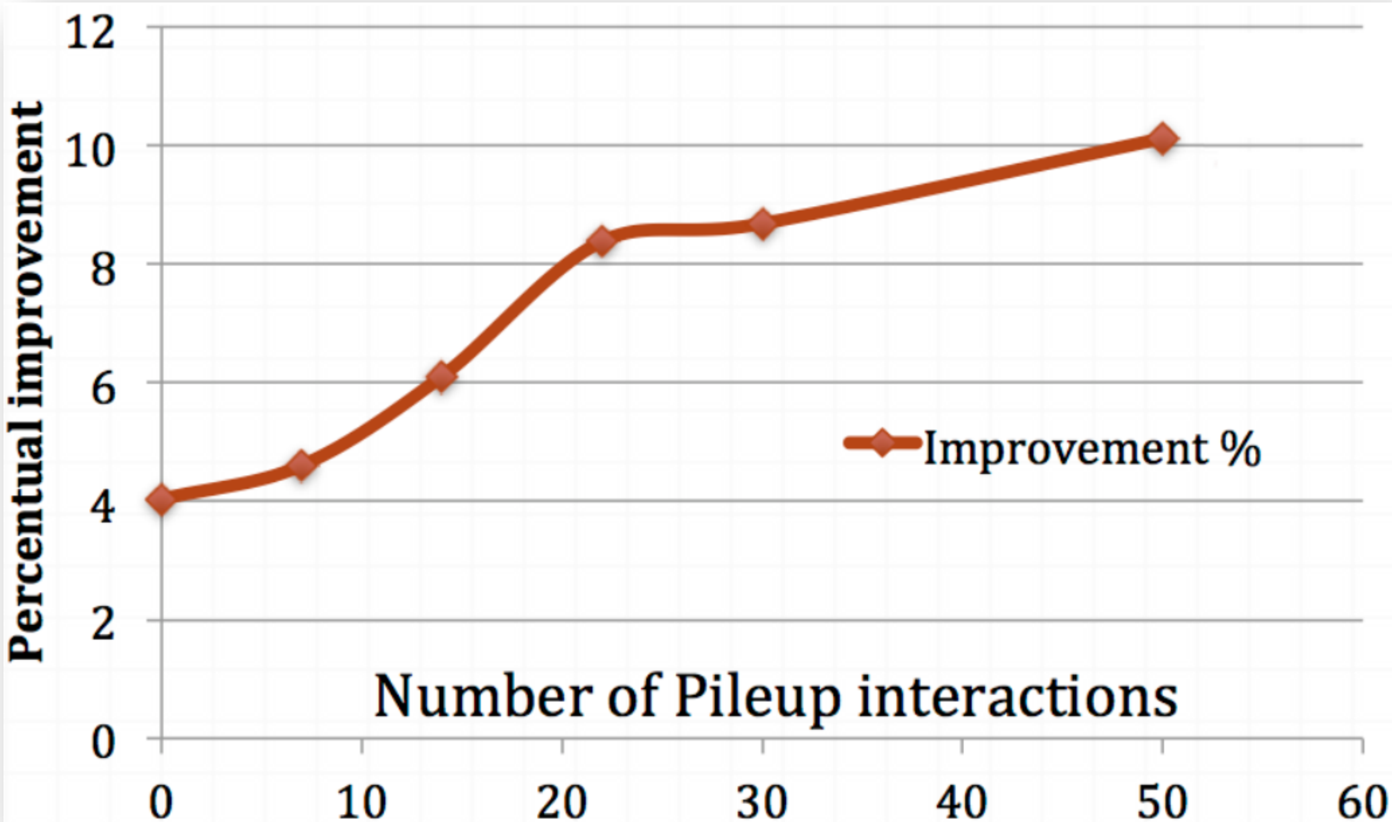
- **Leverage compiler features to achieve peak performance**, e.g.:
  - Functions inlining
  - Optimisation flags
  - Autovectorisation, super word parallelism (SLP)
  - Dare to use “the latest greatest” version
  - **Prefer compile-time to dynamic (runtime) mechanisms**

Let the Compiler Help You

- Most powerful tool at disposal when targeting peak performance
- Knowledge of its capabilities and the flags necessary to steer them always rewards with performance, e.g.
  - Treatments of FP numbers
  - Optimisation levels
  - Link time optimisation

CERN School of Computing

An Example From CMS



D Giordano and G Sguazzoni 2012 *J. Phys.: Conf. Ser.* 396 022044 doi:10.1088/1742-6596/396/2/022044
CMS reconstruction improvements for the tracking in large pile-up events

Increasing event occupancy, instantaneous luminostiy, track combinatorics.
"Event Complexity"

- CMSSW reconstruction
- gcc 4.3 → gcc4.6
- Autovectorisation enabled

CERN School of Computing

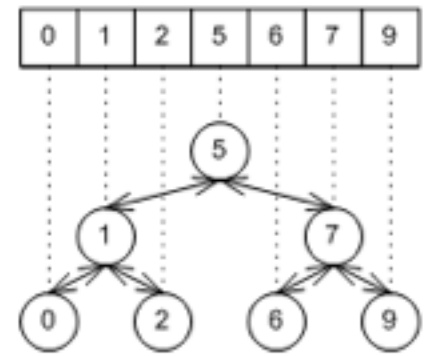
N^2

Constant

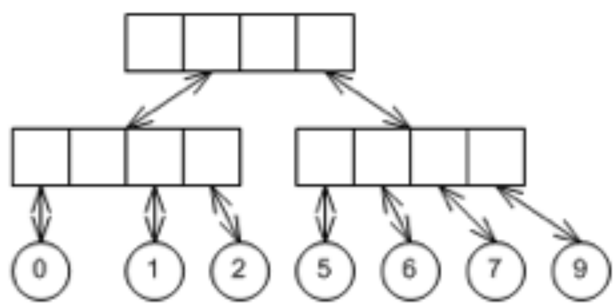
N^N

NlogN

logN



Data structures and Algorithms



CERN School of Computing

Foreword

- **Not a lecture on algorithms** and data structures
  - Tons of books (since >50y out there)
  - We would need a semester (at least)
- Rather a "pragmatic primer" about algorithms and data structures natively offered by C++
- A reasonably good initial choice of algorithm and data structures always rewards with performance!
  - The wrong choice would kill performance
  - Changing algorithms and data structures after the application is released is hard

CERN School of Computing

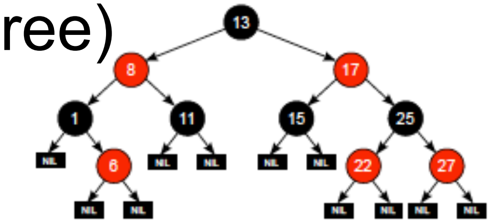
The STL Containers

- STL in C++03 offers efficient containers, among which:

`vector<T>:` consecutive in memory. A powerful class!

`list<T>:` double linked list

`map<T,K>:` associative container (red-black tree)



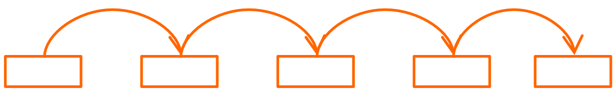
[Wikipedia / GFDL]

`set<T>:` unique elements

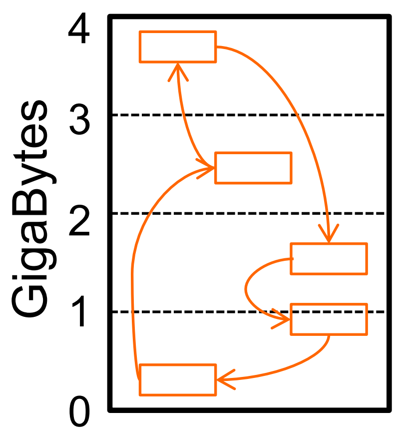
- Try to make use of those: a combination of efficient implementation and generality
  - Gift of meta programming!

Containers in Real Life

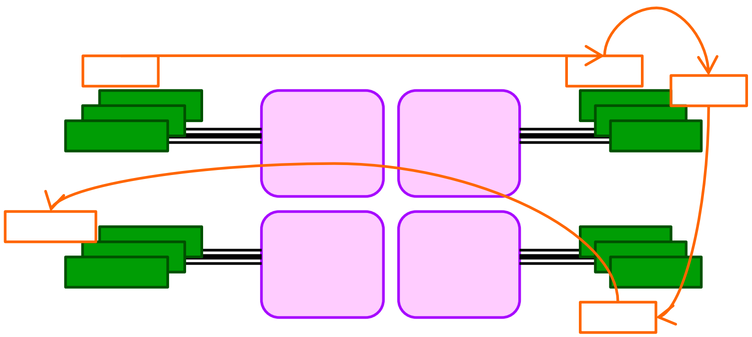
- List and vector: almost the same, right?
  - A sequence of ordered elements
  - List offers a couple of goodies like `push_front, sort, erase..`
- Wrong! For example, iteration:



Logically, this is what happens



Actually, the elements may be scattered in the virtual memory like this!



And on a NUMA architecture, like this!

STL Containers: some C++11 goodies

- `std::array`: safer re-incarnation of the C array
  - std::array<int,12> intArraySize12 {1,2,3,4};
- New containers: `unordered_{map, multimap, set}`
  - Hashed key containers: C++11 offers efficient hashing for many classes natively. Can be expanded (template specialisation)
  - Efficient lookups in presence of complex objects as keys (e.g. strings)
- Initialiser lists: std::vector<int> v {1,2,3,4};
  - Less code, less mistakes, more correctness!
- Not only inserting, but emplacing. E.g.:
  - ```
    template< class... Args >
    void std::vector<T>::emplace_back( Args&&... args );
    ```
    Move semantics
  - Avoid copies and moves: always prefer `emplace_back` to `push_back`
    ```

CERN School of Computing

Move Semantics in a Nutshell

- One issue with C++: unintentional triggering of copies
  - Memory churn → serious performance loss
  - Modern C++ offers new ways of coping with this

```
[…]
std::vector<int>
timesTwo(const vector<int>& v)
{
    std::vector<int> tmp;
    tmp.reserve(v.size());
    for (auto itr = v.begin();
         itr != v.end(); ++itr ){
        tmp.push_back( 2 * *itr );
    }
    return tmp;
}

int main(){
    std::vector<int> v; v.reserve(100);
    for ( int i = 0; i < 100; i++ )
      v.push_back( i );
    v = timesTwo ( v );
}
```

Temporary!

Would'n it be nice to "move" (rather than copying) the content of the tmp out of the function scope and "move" it then within v (rather than assigning)?

Copy back the full vector and throw away the temporary!

Not accessible anymore!

CERN School of Computing

A Copy which is not a Copy

Copy Constructor

```
template <class T>
class avector {
    T * fBegin;
    T * fEnd;
 [...]
public:
  avector( const vector & tmp ){
   clear();
   reserve(tmp.size());
   for (auto& i:tmp)
     push_back(i);
  }
}
```

Copy elements

Move Constructor

```
template <class T>
class avector {
    T * fBegin;
    T * fEnd;
  [...]
public:
  avector( vector && tmp )
  : fBegin ( tmp.fBegin )
  , fEnd( tmp.fEnd ){
   tmp.fBegin = nullptr;
   tmp.fEnd   = nullptr;
  }
}
```

Transfer ownership!

- All stl containers have move ctors and assignment implemented!!
- && is the notation for an “rvalue reference”
  - Beyond the scope of this lecture
- Some classes are move only: e.g. std::thread

Useful reading:
The C++ programming Language, 4th ed. B. Stroustrup

The STL Algorithms

- STL provides a variety of useful pre-packed algorithms
  - `#include <algorithm>`
- `find, find_if, shuffle, rotate, copy_if, sort, stable_sort …`
- General purpose low-level functionalities, often used in programs of all kinds
- Performant and correct:
  - Hard to reach the same quality implementing from scratch
- Can replace the stl implmentation behind, user code unchanged!
  - **STLXXL**: huge collections (~TB!), http://stxxl.sourceforge.net
  - Parallel mode STL: http://gcc.gnu.org/onlinedocs/libstdc%2B%2B/manual/parallel_mode.html

The STL Algorithms

```
#include <algorithm>

std::vector<int> v={1,2,3,4,5};

// Randomise content
std::shuffle(v.begin(), v.end(),
             std::default_random_engine(seed));

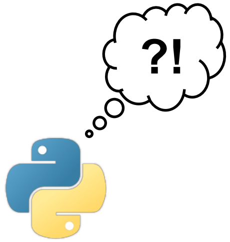
// Sort and reverse sort
std::sort(v.begin(), v.end());
std::sort(v.begin(),v.end(),
          [](int i, int j){return j<i;});

// contains
decltype(v) vv={1,2,3};
bool incl = std::includes(v.begin(),v.end(),
                          vv.begin,vv.end());

// Apply function to range
std::for_each(v.begin(), v.end(),
              [](int i){return i*2;});
```

Internally, moves are used not to imply huge overheads!

The predicate can be changed!



Great synergy!

CERN School of Computing

Take Away Messages

- C++ evolve{s,d}! High throughput applications can take advantage of it:
  - Clearer, more modern syntax
  - Lots of building blocks available: don't reinvent the wheel
  - Metaprogramming has even more potential
- Move whatever you can to compile time
  - Templates, constexpr
- New STL: containers, algorithms and their interplay with other language features (like lambdas)

Backup

Example: Visitation

- Problem
  - A big data structure (“S”)
  - Need to visit all of its nodes
  - Need to perform small (user defined) operations on some
  - Skeleton for the “visitor” class provided

It works, but the performance would be less than ideal because of indirections ☹

- **Solution 1: abstract interface**

At run time, the call is forwarded to the right method!

```
class Visitor{
 public:
 int scanBDS(){
  return callAllVisitNodes()};

 virtual bool visitNodeType1() = 0;
…
 virtual bool visitNodeTypeN() = 0;
};
```

Provided by the developer of “S”

```
class MyVisitor: public Visitor{
 public:
 virtual bool visitNodeType1(){
   doWork();}
[…]
};

MyVisitor scanner; scanner.scanBDS();
```

Take advantage from the interface offered

Provided by the user using the “S”

CERN School of Computing

Curiously Recurring Template Pattern

- **Solution 2: templates!**

At compile time, the call is forwarded to the right method!

Provided by the developer of BDS

```
template<class Derived>
class VisitorCTRD {
 public:
 bool visitNodeType1(){(static_cast<Derived>(this))->visitNodeType1();}
[…]};
```

Provided by the user of the BDS

Inherits from something templated with itself. Recursion!

Still take advantage from the interface offered!

```
class MyVisitor: public VisitorCTRD<MyVisitor>{
 public:
 bool visitNodeType1(){doWork();}
[…]};

MyVisitor scanner; scanner.scanBDS();
```