16-06-2017

Class

- *Virtual functions*
  - Are we guaranteed that member functions of the right classes will be used?

```
void Print_it( const studente &s) {
  s.Print();
}
...
int main(){
  studente a("Francesco");
  studente_dottorato b("Cafagna",1);
  ...
  Print_it(a);
  Print_it(b);
  return 0;
}
```

Which Print() will be used? The one of studente or studente_dottorato?

If would be used the one of studente the information on the doctoral years would be lost.

Class

- *Virtual functions*
  - This problem can be avoided declaring a member function as: virtual

    virtual return_type name(parameter_list);
  - The virtual prefix stands for the fact that the name of the function will behave like an interface between the base class function and the one defined in the derived class.
  - For this exact reason the formal arguments must be identical, as well as the return type.
  - Usually are the virtual member the one called: *method*

Class

```
class studente {
        private:
                string _name;
                int _anno_iscrizione;
                int _numero_matricola;
                vector<string> _esami_sostenuti;
        public:
                void SetName(string &s){ _name=s;}
                string GetName() {return _name;}
                virtual void Print(){ …}
                …
        };
        class studente_dottorato :public studente {
        private:
                int _ciclo_dottorato;
                vector <string> _classi_dottorato;
        public:
                void SetName(string &s){studente::SetName(s);}
                void Print(){…; studente::Print();}
                …
        };
```

***virtual* warns the compiler that the member function could be re-defined in the derived class …**

… as in fact is.

Class

```
void print_all(const vector<studente *> &vs){
    for(vector<studente *>::const_iterator p=vs.begin(); p!=vs.end();++p)
        (*p)->Print();
}

int main(){
    Studente laure;
    Studente_dottorato dot;
    Vector<studente *> vs;
    Fill_vs(laure,vs);
    Fill_vs(dot,vs);
    ...
    Print_all(vs);
}
```

When a studente_dottorato would be used Print of studente_dottorato will be used

Passing studente, will use studente::Print()

Passing studente_dottorato as it was a studente it will use studente_dottorato::Print()

Class

- *Virtual destructors*
  - What does happen if I destroy an object of a derived class using a pointer to its base class type?

```
void Print_it( const studente &s) {
  s.Print();
}
...
int main(){
  studente a("Francesco");
  studente *b= new studente_dottorato("Cafagna",1);
  ...
  Print_it(a);
  Print_it(*b);
  delete b;
  return 0;
}
```

Using studente_dottorato as it was a studente

What is actually deleted? The studente or the studente_dottorato?

Class

- *Virtual destructors*
  - Base class destructor: *studente*, knows nothing about the derived class: *studente_dottorato*, so it doesn't know what it should do to destruct the latter.
  - To do it also the base class destructor must be declared as virtual.
  - The fact that a destructor is virtual in the base class insure that any derived class destructor will be used. Besides if in the derived class no destructor has been declared then at least the base class one will be called.

Class

- Abstract methods
  - What about if the class describes general objects which could have som functionality only if specilized to an actual, of derived classes, object?
  - For example:

```
class writable {
private:
  ofstream * ofile;
public:
  void SetFile(ofstream * of){ ofile=of;}
  ofstream GetFile() const { return *of;}
  virtual void Write();
};
class studente: public writable{
...
};
```

  - Does the class writable knows what should be done in a Write?

Class

- Abstract methods
  - The writable class describe object that can be written into an ostream, very general and doesn't know what data should be written in a Write().
  - Pay attention, a simple virtual method must be defined. So Write() must be defined as doing nothing, using an empty scope: *virtual void Write(){};*

```
class writable {
private:
  ofstream * ofile;
public:
  void SetFile(ofstream * of){ ofile=of;}
  ofstream GetFile() const { return *of;}
  virtual void Write(){};
};
class studente: public writable{
…
};
```

  - It might work. The studente programmer should have been defined Write().
  - But what if he will forget? What about if no Write() is defined in the derived class?

Class

- Abstract methods
  - It is possible to declare methods, i.e. virtual *member functions*, as abstract and force the compiler to check if the actual one has been defined in the derived class. To do that I just need to assign *0* to the method name in the declaration:

```
class writable {
private:
  ofstream * ofile;
public:
  void SetFile(ofstream * of){ ofile=of;}
  ofstream GetFile() const { return *of;}
  virtual void Write()=0;
};
class studente: public writable{
…
};
```

  - In this way I am forced to supply the member function Write() in my derived class.

Class

- Abstract Classes
  - What about if is a whole class that describe an object that has no real counterpart, that is is abstract?
  - In this case all its methods will be astract ones and the class itself will be named as abstract.
  - Hierarchies of abstract classes provide a way of expressing concepts without complication derived by the implementation details.
  - An abstract class is an interface.

Class

- Multiple inheritance
  - Our studente_dottorato is a studente.
  - Our studente_dottorato is also an object that can be written to a file, so is a writable.
  - How can I inherith both studente and writable?
  - Both base classes can be inherited:

    class studente_dottorato: public studente, public writable { … };
  - It can be done listing, comma separated, the base classes …
  - … of course it is possible to let studente be a Writable …

    class studente: public writable { // ricordati di scrivere Write … };

    class studente_dottorato: public studente { // Speriamo che studente abbia un Write astratto o almeno virtuale … };

Exercise

- We need two types: an angle, a cosine.
- It should be possible to assign a double to both types.
- It should be possible to treat them as doubles, i.e. to implicitly cast them to double.
- It should be possible to assign an angle to a cosine and vice versa.

Exercise

```
#include <cmath>

class CAng;

class Ang {
private:
  double ang_;
public:
  Ang( const double &t=0.) : ang_(t) {};
  Ang( const Ang &t) { ang=t.ang_; }
  operator double() const { return ang_; }
  Ang & operator=(const double t) { ang_ =t; return *this; }
  Ang & operator=(const CAng &t);
};

class CAng {
private:
  double cang_;
public:
  CAng( const double &t=0.) : cang_(t) {};
  CAng( const CAng & t) { cang_ =t.cang_; }
  operator double() const { return cang_; }
  CAng & operator=(const double t) { cang_ =t; return *this; }
  CAng & operator=(const Ang &t) { cang_ =cos(t); return *this; }
};


Ang & Ang::operator=( const CAng &t) { ang=acos(t); return *this; }
```

Will work ?

YES!!!

Exercise

```
class Angles {
public:
  typedef Ang Angle;
  typedef CAng CAngle;

  Angles() { SetAngle(0);  }
  Angles(const CAngle & ct) { SetCAngle(ct); }
  Angles(const Angle & t) { SetAngle(t); }
  Angles(const Angles & a) { SetAngle(a.GetAngle()); }

  Angle GetAngle() const { return angle_; }
  CAngle GetCAngle() const { return cangle_; }
  void SetAngle( Angle t ) { cangle_=angle_=t; }
  void SetCAngle( CAngle ct ) { angle_=cangle_=ct; }

  Angles & operator= ( const Angles & a) {
    SetAngle(a.GetAngle());
    return *this;
  }

  Angles & operator= ( const Angle &a) {
    SetAngle(a);
    return *this;
  }

  Angles & operator= ( const CAngle &a) {
    SetCAngle(a);
    return *this;
  }

  void Print( std::ostream & o=std::cout) const {
    o << " Angle: " << angle_ <<", Cosine: " << cangle_ << std::endl;
  }

private:
  Angle angle_;
  CAngle cangle_;

};
```

Exercise

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

const double GRAD2RAD=acos(0.)/90.;

int main() {
  double t;
  Angles::Angle a;
  Angles b;
  std::cout << " Enter an angle (Degree):" ;
  while(std::cin >> t) {
    a=GRAD2RAD*t;
    b=a;
    b.Print();
    std::cout << " Enter an angle (Degree):" ;
  }
  return 0;
}
```

- test_angle.tgz, https://owncloud.ba.infn.it/public.php?service=files&t=dd78377687db39ec48e7006924c3789b

Exercise

```
#include <cmath>

class CAng;

class Ang {
private:
  double ang_;
public:
  Ang( const double &t=0.) : ang_(t) {};
  Ang( const Ang &t) { ang=t.ang_; }
  operator double() const { return ang_; }
  Ang & operator=(const double t) { ang_ =t; return *this; }
  Ang & operator=(const CAng &t);
};

class CAng {
private:
  double cang_;
public:
  CAng( const double &t=0.) : cang_(t) {};
  CAng( const CAng & t) { cang_ =t.cang_; }
  operator double() const { return cang_; }
  CAng & operator=(const double t) { cang_ =t; return *this; }
  CAng & operator=(const Ang &t) { cang_ =cos(t); return *this; }
};


Ang & Ang::operator=( const CAng &t) { ang=acos(t); return *this; }
```

What does differentiate the Ang and Cang classes?

Just the two transformation functions

Exercise

```
#include <cmath>

struct cos_double {
    double operator() (const double&a ) { return cos(a); }
};

struct acos_double {
    double operator() (const double &a ) { return acos(a); }
};

class CAng;

class Ang {
private:
  double ang_;
   acos_double  func_;
public:
  Ang( const double &t=0.) : ang_(t) {};
  Ang( const Ang &t) { ang_ =t.ang_; }
  operator double() const { return ang_; }
  Ang & operator=(const double t) { ang_ =t; return *this; }
  Ang & operator=(const CAng &t);
};

class CAng {
private:
  double cang_;
  cos_double func_;
public:
  CAng( const double &t=0.) : cang_(t) {};
  CAng( const CAng & t) { cang_ =t.cang_; }
  operator double() const { return cang_; }
  CAng & operator=(const double t) { cang_ =t; return *this; }
  CAng & operator=(const Ang &t) { cang_ =func_(t); return *this; }
};


Ang & Ang::operator=( const CAng &t) { ang=func_(t); return *this; }
```

What about introducing functors?

Exercise

```
#include <cmath>

struct unary_function_double{
    typedef double arg_type;
    typedef double ret_type;
};

struct cos_double : public unary_function_double{
    ret_type operator() (const arg_type &a ) { return cos(a); }
};

struct acos_double : public unary_function_double{
    ret_type operator() (const arg_type &a ) { return acos(a); }
};

class CAng;

class Ang {
private:
  double ang_;
   acos_double  func_;
public:
  Ang( const double &t=0.) : ang_(t) {};
  Ang( const Ang &t) { ang=t.ang_; }
  operator double() const { return ang_; }
  Ang & operator=(const double t) { ang_ =t; return *this; }
  Ang & operator=(const CAng &t);
};

class CAng {
private:
  double cang_;
  cos_double func_;
public:
  CAng( const double &t=0.) : cang_(t) {};
  CAng( const CAng & t) { cang_ =t.cang_; }
  operator double() const { return cang_; }
  CAng & operator=(const double t) { cang_ =t; return *this; }
  CAng & operator=(const Ang &t) { cang_ =funct_(t); return *this; }
};


Ang & Ang::operator=( const CAng &t) { ang=func_(t); return *this; }
```

Can we express commonalities?

Exercise

Can we generalize?

```
#include <cmath>

struct unary_function_double{
   typedef double arg_type;
   typedef double ret_type;
};

struct cos_double : public unary_function_double{
   ret_type operator() (const arg_type &a ) { return cos(a); }
};

struct acos_double {
   ret_type operator() (const arg_type &a ) { return acos(a); }
};

template<class Function> class transformable {
public:
  typedef Function function_type;
  typedef double value_type;
  typedef transformable<Function> self_type;
  transformable( const value_type &t=0.): val_(t) {};
  transformable( const self_type & t): val_(t.val_) {}
  operator value_type() const {  return val_;  }
  self_type & operator=(const value_type t) {val_=t; return *this;}
  template<class Trans_from> self_type & operator=(const Trans_from &t){
     val_=func_(t); return *this; };
private:
  function_type func_;
  value_type val_;
};

typedef transformable<acos_double> Ang;
typedef transformable<cos_double> CAng;
```

Exercise

Can we generalize?

```
class Angles {
public:
  typedef Ang Angle;
  typedef CAng CAngle;

  Angles() { SetAngle(0);  }
  Angles(const CAngle & ct) { SetCAngle(ct); }
  Angles(const Angle & t) { SetAngle(t); }
  Angles(const Angles & a) { SetAngle(a.GetAngle()); }

  Angle GetAngle() const { return angle_; }
  CAngle GetCAngle() const { return cangle_; }
  void SetAngle( Angle t ) { cangle_=angle_=t; }
  void SetCAngle( CAngle ct ) { angle_=cangle_=ct; }

  Angles & operator= ( const Angles & a) {
    SetAngle(a.GetAngle());
    return *this;
  }

  Angles & operator= ( const Angle &a) {
    SetAngle(a);
    return *this;
  }

  Angles & operator= ( const CAngle &a) {
    SetCAngle(a);
    return *this;
  }

  void Print( std::ostream & o=std::cout) const {
    o << " Angle: " << angle_ <<", Cosine: " << cangle_ << std::endl;
  }

private:
  Angle angle_;
  CAngle cangle_;

};
```

Exercise

Can we generalize?

```
template<class First, class Second >class synch_pair{
public:
  typedef First first_type;
  typedef Second second_type;
  typedef synch_pair<First, Second> self_type;

  synch_pair() { SetFirst(0);  }
  synch_pair(const second_type & ct) { SetSecond(ct); }
  synch_pair(const first_type & t) { SetFirst(t); }
  synch_pair(const self_type & a) { SetFirst(a.first()); }

  first_type first() const { return first_; }
  second_type second() const { return second_; }

  synch_pair & operator= ( const self_type & a) {
    SetFirst(a.first());
    return *this;
  }

  synch_pair & operator= ( const first_type &a) {
    SetFirst(a);
    return *this;
  }

  synch_pair & operator= ( const second_type &a) {
    SetSecond(a);
    return *this;
  }

  void Print( std::ostream & o=std::cout) const {
    o << " First: " << first_ <<", Second: " << second_ << std::endl;
  }

private:
  first_type first_;
  second_type second_;
  void SetFirst( const first_type & f ) { second_ =first_ =f; }
  void SetSecond( const second_type & s ) { first_ =second_ =s; }

};
typedef synch_pair<Ang,CAng> Angles;
```

Exercise

```
#include <iostream>
#include "angles_advanced.h"
const double GRAD2RAD=acos(0.)/90.;

int main() {
  double t;
  Angles::first_type a;
  Angles b;
  std::cout << " Enter an angle (Degree):" ;
  while(std::cin >> t) {
    a=GRAD2RAD*t;
    b=a;
    b.Print();
    std::cout << "Angle: " << a << std::endl;
    std::cout << " Enter an angle (Degree):" ;
  }
  return 0;
}
```

- test_angles_advanced.tgz, https://owncloud.ba.infn.it/public.php?service=files&t=01c505b61c1b2f6091fa262cd199d496

Static

- How can we let a variable survive to a scope (or function), if we don't want to declare it global (that is we don't want to be seen or used by other functions)?
- We can declare it as: static

```
void f(int a){
    while(a--) {
      static int n = 0;
      int x = 0;
      cout << "n==" << n++ <<" , x==" << x++ << endl;
    }
}
int main() { f(3); return 0; }
```

Will print:

N== 0, x== 0
N== 1, x== 0
N== 2, x== 0

Static

- A static variable is allocated once and only once and used every time is encountered in the code.
- What about classes?
  - We can declare static members
  - A static member can be used (called) like any other else and referred to any specialized object, i.e. just using the class namespace.
- Static members must always be defined. So they can be declared if the type is not yet defined.

Static

```
class Date {
        Private:
                Int _d,_m,_y;
                Static Date default_date_;
        Public:
                Date(int d=0, int m=0, y=0);
                //…
                Static void set_default(int, int, int);
};
Date::Date( int d, int m, int y) {
        _d=d ? d: default_date_.GetD();
        _m=m ? m: default_date_.GetM();
        _y=y ? y: default_date_.GetY();
}
Date Date::default_date_(15,05,2007);
Void Date::set_default(int d, int m, int y){
        Date::default_date_ = Date(d,m,y);
}
```

Date type is not yet defined. I would not be able to define not static members.

Static members must always be defined.

- In my code I can write:

```
Int main() {
//…
Date::set_default(4,5,1945);
//…
Return 0;
}
```

I am using a static member without specifying the object (variable)!!!

Typename, inline

- What are these new prefixes: *typename, inline*?
- *typename*
  - A *template* can have members that are type and other that are not (data or a function) :

```
template<class T> struct Vec {
        typedef T value_type;  // un membro che è un tipo
        static int count; // un membro che non è un tipo, è
                          //  un dato
};
int x=Vec<MyClass>::count; // per default membri si assumono
                           // di tipo dato
typename Vec<MyClass>::value_type x; // devo specificare che si
                                     // tratta di un tipo non di un dato
```

Typename, inline & explicit

- *inline*
  - When used for functions or member functions "suggests" to the compiler to insert directly the function assembler code instead of a function call.
  - In practice we ask the compiler to avoid a jump to the function code but to directly add the function code to the executable code.
  - It should produce a larger executable.
  - The behavior is strongly compiler and optimization level depended. It is not obvious to foreseen the result. Compiler could decide or not to use the inline for a function.

Standard Library: fstream

- fstream
  - Support I/O functionality to write to or read from a file.
  - They can be of input, output or both types.
  - In the declaration it is possible to use a specify the kind of file. Specialized type are provided.
    - ifstream , input type file. Only input operations are supported, i.e. read from.
    - ofstream, output type file. Only output operations are supported, i.e. write to.
    - fstream, generic type file. Both input and output operations are supported.

Standard Library: fstream

- fstream
  - Usage?
    - In the declaration, file name and I/O mode can be specified using the functional form.

```
#include <fstream>
...
std::ofstream file_di_output("nomedelmiofile.out");
std::ifstream file_di_input("nomedelmiofile.inp");
std::fstream file_generico("nomedelmiofile.dat");
std::ofstrem file_di_output("nomedelmiofile.out");
// File generico, specializzato con parametri I/O
std::fstream file_di_output("nomefile.out",ios_base::out);
```

Standard Library: manipulator

- *endl*, *hex*, *dec*, are *iostream* manipolators, that is global functions, that can be used with the operators *<<* and *>>*, modifying the state and properties or format options of a *stream*
- They are defined into the *namespace std*
- Format flags can be also modified directly using *stream* methods.

Standard Library: manipulator

- *boolalpha:* print a bool value as a string (true,false)
- *dec:* print integer as decimals
- *endl:* insert new line and flush buffer
- *ends:* insert a NULL
- *fixed:* use a fixed-point format
- *flush:* flush the stream buffer
- *hex:* print integer in hexadecimal base
- *internal:* fill with a character field up to the internal value
- *left:* left allign the text
- *noboolalpha:* print a bool value as a digit (0,1)
- *noshowbase:* don't print the numerical base
- *noshowpoint:* don't print the decimal point
- *noshowpos:* don't print the plus sign
- *noskipws:* don't skip blanks
- *nounitbuf:* don't force a *flush* after insertion
- *nouppercase:* don't print in uppercase
- *oct:* print an integer in octal base
- *resetiosflags:* reset a format flag
- *right:* right aligned text
- *scientific:* print in scientific notation
- *setbase:* set the numerical base
- *setfill:* set the fill character
- *setiosflags:* set a format flag
- *setprecision:* set a numerical precision
- *setw:* set the field width
- *showbase:* print the numerical base
- *showpoint:* print the decimal point
- *showpos:* print the plus sign
- *skipws:* skip blanks
- *unitbuf:* force a *flush* after every insertion
- *uppercase:* print in uppsercase
- *ws:* delete whitespaces

Standard Library: S(T)L

- Containers
  - In the STL there is support for containers, i.e. objects that are able to store any other objects of any given type.
  - There are two kind of containers: sequential, i.e. like an array of objects having an integer index:
    - *vector*
    - *list*
    - *deque*
  - … or associative, i.e. an array of objects having a not integer index:
    - *map*
    - *set*
  - We can cite as "similar to containers" also: *strings*, *valarrays* e *bitsets* … and *arrays*, of course.
  - From a sequence containers, can be derived the so called *sequence adapters:*
    - *queue*
    - *priority_queue*
    - *stack*
  - From an associative containers can be derived:
    - *multimap*
    - *multiset*

Standard Library: S(T)L

- Containers: iterators
  - The most common usage of a container is an iteration on every element.
  - This is supported by a special type that "knows" how to navigate or access a container: *iterator*.
  - Iterators are specilized according to the container type must be accessed, but they share the same interface. For all of them member functions: *T* first()* e *T* next()* , are present.
  - These methods to return a pointer to the very first object of type *T* collected in the container or, respectively, the next one. Zero is returned when the sequence is over.
  - In this way a generic algorithm can be written regardless from the container type:

```
template< class T> int count(iterator<T> & ii, T compar) {
   int c=0;
   for( T* p=ii.first(); p; p=ii.next()) if(*p==compar) c++;
   return c;
}
```

Standar Library: S(T)L

- Containers
  - You can think about a container as an ordered sequence according to the corresponding forward or reversed iterator you like to use.
  - For an associative container the order will be base on the comparison criteria associated to the container index. By default this is: < .
  - Iterators can be obtained by the container using the methods:
    - *begin()* forward iterator pointing to the first element.
    - *end()* forward iterator pointing to the last element.
    - *rbegin()* reverse iterator pointing to the first element.
    - *rend()* reverse iterator pointing to the last element.
  - To access element there are methods and operator:
    - *first()* returns the first element.
    - *back()* returns the last element.
    - *operatore []* array subscript operator. Not supervised random access (not defined for a *list*)
    - *at()* Supervised random access. Boundary check is performed (defined only for *vector* and *deque*)

Standard Library: S(T)L

- Containers
  - Sequential containers support sequential operations via function methods:
    - *push_back()* add an element at the end of the sequence.
    - *pop_back()* removes last element in the sequence.
    - *push_front()* add an element at the beginning of a sequence (only for *list* and *deque*
    - *pop_front()* removes the first element (only for *list* and *deque*)
  - … and some standard operations for a list:
    - *insert(p,x)* adds x before p.
    - *insert(p,n,x)* adds n copies of x before p.
    - *insert(p,first,last)* adds elements in a sequence [first:last[before p.
    - *erase(p)* removes the p element.
    - *erase(first,last)* removes the element in the sequence [first:last[
    - *clear()* removes all the elements.
  - Besides few more operations:
    - *size()* number of elements in the containers.
    - *empty()* is it the container empty?
    - *max_size()* largest dimension of the container.
    - *swap()* swap elements between two containers.
    - *==* is it the contents of two containers equals?
    - *!=* are two containers different?
    - *<* does a container precede another?

Standard Library: S(T)L

- Containers
  - Constructors and assignment operations:
    - *container()* default creator, empty container.
    - *containern)* creates a container with n element created using the element default creator (not for the associative)
    - *container(n,x)* creates a container with n copied of x (not for the associative)
    - *container(first,last)* creates a container with initial elements copied from the sequence *[first:last[*
    - *container(x)* creates a container copying elements of container x (copy).
    - *=* assigns, copies all elements from a container into another.
    - *assign(n,x)* assigns *n* copies of x (not for the associative)
    - *assign(first,last)* assigns in the range *[first:last[*
  - Associative containers support element accesses using a key. So sort or search operations are needed on a key:
    - *[k]* subscribed access using *k as an index (k* is a name of any given type on which a comparison operator has been defined. For example: *<)*
    - *find(k)* search for the element paired (associated) to the *k* key.
    - *lower_bound(k)* search for the first element associated to a key lesser than *k.*
    - *upper_bound(k)* search for the first element associated to a key greater than *k.*
    - *equal_range(k)* search for *lower_bound and l'upper_bound* of elements associated to a key *k.*
    - *key_comp()* copy the object used to apply the comparison criteria.
    - *value_comp()* copy the object used to apply the comparison criteria to the object mapped by a key.

Standard Library: S(T)L

- Containers
  - In a container class definition are also defined:
    - *value_type* the element type, *i.e. typedef T value_type*
    - *size_type* the subscript (index) type, element counts, etc.
    - *iterator* the iterator type associated to the container, *i.e. value_type**
    - *const_iterator* like *iterator* but for constant elements, *i.e. const value_type**
    - *reverse_iterator* iterator to backward parse the container.
    - *const_reverse_iterator* like *reverse_iterator* but for constant elements
    - *reference* *value_type &*
    - *const_reference* *const value_type &*
    - *key_type* key *k type* (for associative only)
    - *mapped_type* the type of the element mapped by the key *k* (for associative only)
    - *key_compare* the type of the comparison criterion (for associative only)

Standard Library: S(T)L

- containers
  - *vector :*

```
typedef vector<int> my_vi;
my_vi caf;
for(my_vi::value_type a=0;a<10;++a){
   caf.push_back(a);
}
std::cout << " Dim. : " << caf.size() << std::endl;
int c=0;
for(my_vi::const_iterator i=caf.begin(); i!= caf.end(); ++i)
   std::cout << " Elemento " << ++c <<": " << *i << std::endl;
for( my_vi::size_type i=0; i<caf.size(); ++i)
   std::cout << " Elemento " << i <<": " << caf[i] << std::endl;
```

Template specialization

Grows up dynamically !!!

Asking to the object its size

Element access using an iterator

Element access using the operator *[]*

Exercise on STL

Write a function that:

- *Given a string.*
- *Given a delimitation character, i.e. delimiter.*
- *Returns position of the character after the n-th position of the delimiter.*
- *The search must start from any given position in the string.*
- *For example:*
  - *delimiter: ',' *
  - *String: "Stringa, di, prova, con, molte, virgole"*
  - *N=3*
  - *Starting search position: 9*
  - *The function returns: 25*
- The function must:
  - *Use recursivity.*
  - *Its scope must not contains more than two lines of code.*
- *Hint:*
  - Get a look to: http://www.cplusplus.com/reference/string/string/ , for more string functionalities.
  - The function interface …

```
std::string::size_type  SearchNDelimiter(const std::string &s, int nth, char delim,
    std::string::size_type pos  )
// check for string::find(…)
```