14/06/2017 afternoon

User defined type: Class

- Constructor
  - Being a member function, constructor, also if special, can have initialized parameter in the parameters declaration:

```
studente(sting n="no lastname", int i=1) :
  name(n), anno_dottorato(i){};

studente primo("Francesco");
```

  - This avoids constructor proliferation.

User defined type: Class

- Destructor
  - The same way I do need some operations to be done during initialization, actions could be needed when objects are destroyed.
  - This possibility, complements the constructor one, and is implemented by: destructors
  - Like constructors they have the same class name, but prefixed by a complement sign: *~*

```
~ studente(){ std::cout <<" All has been destroyed " << std::endl;
  }
```

User defined type: Class

- Copy
  - Objects of a base type, can be copied, so the user defined ones:
    studente a("caf"); studente b=a; // assign 'a' data members to 'b',
    // i.e. studente b(a);
  - The default copy copies every data member into each other object. If a copy constructor has been defined then the copy constructor will be called.
  - A copy constructor is a constructor having one and only parameter: a constant reference to a class object:

    studente::studente(const studente &a) { name=a.name;}
- Constant member functions
  - If a method does not modify any member of a class, then it can be declared such a way it can be called also for a constant object. This is done using the postfix: *const*, in the declarator:

    void Print_student_name() const;

    ...

    void Print_my_student(const student &a) { a.Print_student_name();}

Calling a method of a constant object

User defined type: Class

- *this*
  - *this* returns the pointer to the actual object which member function has been called.
  - So in a member function of a *class C*, the returned type by *this* is: *C **
  - In a member function const of a class C, the returned type by *this* is: *const C **

```
studente * studente::where_in_memory(){ return this;}
…
studente Cafagna("F S",1);
std::cout << "Locazione di memoria dell'oggetto di nome Cafagna"
   << Cafagna.where_in_memory() << endl;
```

Standard Library: S(T)L

- C++ is shipped with a *Standard Library*.
  - *Standard Library* is defined in the *namespace std*, and include features to:
    - Support Run-time operations (typeid, memory operation, etc.)
    - Implement the C standard library (printf, scanf, etc.)
    - Support for string and I/O *streams* (string, cout, cin, etc.)
    - Support for numeric calculations (complex, etc.)
    - Implement Containers and standard algorithms that use them (vector, list, map, iterator, for_each, etc.)
  - It is heavily based on *templates*

Standard Library: I/O

- I/O library: *iostream*
  - *cout*: define a standard output for every type;
  - *cin*: define a standard input for every type;
- A I/O stream can be specilized for files: *fstream,ofstream, ifstream*
- *… or strings: sstream, istringstream, ostringstream*

- A stream provides a mechanism to convert values of a given type into characters sequences.
- User declares a stream and the operator write to << or "read from" >> provides convenient conversions according to the type to be written to or read from.

Standard Library: I/O

- To use the library names declaration is needed for all the module we want to use.
- For that the usual include operation is needed.
- For the I/O stream the header file to be included is: iostream (note the absence of extension)
- This is a standard file (try to read it: less /usr/include/c++/iostream) and can be included with the usual preprocessor directive: *#include*
- STL names are defined in the namespace *std.* So this module name must be specified.

Standard Library: I/O

- The name will be used more are:
  - *cout* : standard output
  - cin : standard input
  - cerr : standard error
- Usually a function to terminate and flush the buffer is used:
  - endl

```
std::cout << " Proviamo a scrivere in output" <<
  std::endl;
std::cin >> t;
```

Exercise

```
#include <iostream>
using std::cout;
using std::endl

void Print( const int & a) {
  cout << " Il valore della variabile : " << a << endl;
};
int main(){
   int a;
   cout <<" Immettere un valore e premere Invio "
        << endl;
   while (cin >> a) Print(a);
   return 0;
```

Include header(s) for STL facilities, namespaces, classes etc. etc.

Declare objects of the namespaces to be used

Will work?

Function definition

Namespace?

1. ***g++ -o test_iostream test_iostream.cpp***
2. ***./test_iostream***

Exercise

```
#include <iostream>
using std::cout;
using std::endl

void Print( const int & a) {
  cout << " Il valore della variabile : " << a << endl;
};
int main(){
   int a;
   cout <<" Immettere un valore e premere Invio "
        << endl;
   while (std::cin >> a) Print(a);
   return 0;
```

Will work?

What's going on here?

1. ***g++ -o test_iostream test_iostream.cpp***
2. ***./test_iostream***

Standard Library: string

- There is a way to not depend from an array, dealing with characters?
- How can I handle string I don't know the length?
- In the STL we can use: string
- These are object storing a character container, NULL terminated, dynamically increasing, according to the need.
- Operators are defined for such an user defined type.
- The write to << and read from >> operators are defined as well, to conveniently be used in conjunction with the iostream.

Standard Library: string

- string
  - Usage?
  - Just like any base type

```
#include <string>
…
using std::string;
…
string name="definition";
string name2("functional form initialization");
…
name+=name2;
name=name2+" try a string";
name+=" how handy!";
```

Standard Library: string

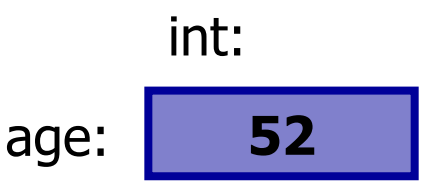
- string
  - Overview of string operators
    - *bool operator==(const string& c1, const string& c2);*
    - *bool operator!=(const string& c1, const string& c2);*
    - *bool operator<(const string& c1, const string& c2);*
    - *bool operator>(const string& c1, const string& c2);*
    - *bool operator<=(const string& c1, const string& c2);*
    - *bool operator>=(const string& c1, const string& c2);*
    - *string operator+(const string& s1, const string& s2);*
    - *string operator+(const char* s, const string& s2);*
    - *string operator+(char c, const string& s2);*
    - *string operator+(const string& s1, const char* s);*
    - *string operator+(const string& s1, char c);*
    - *ostream& operator<<(ostream& os, const string& s);*
    - *istream& operator>>(istream& is, string& s);*
    - *string& operator=(const string& s);*
    - *string& operator=(const char* s);*
    - *string& operator=(char ch);*
    - *char& operator[](**size_type** index);* // Subscript operator! Same as arrays

L&R values

- What's happening if a function returns a reference?

  I can use a function name as an lvalue !

- LVALUE ?!?!?!?!?!?
  - "something that can be on the left side of an assignment"
  - Let's go back to our named box containing the value:



  - Sometimes I do refer to the name, sometimes to the value:

    int a=age; // I do refer to the value, ***rvalue. It is at the right side***

    *age=b*5; // I do refer to the name,* ***lvalue. It is at the left side***

  - If I have a function *get_some(int),* can I write: *get_some(5)=a*;?

Class

- *this*
  - Can I use *member functions* as *lvalue ?*
  - Is it just enough to return the reference to the type the member function is part of?

```
studente & aggiungi_corsi(int n){
   corsi_seguiti+=n;
   return studente(name,anno_dottorato,corsi_seguiti,buoni_cattivi);
}
```

  - I must return the actual object the function has been applied to: *this*

```
studente & aggiungi_corsi(int n){
   corsi_seguiti+=n;
   return *this;
}
studente a("caf",5);
a.aggiungi_corsi(2).Print();
```

Class

- Operators
  - If any user defined type has the same support of the base type, how can I implement operations between objects of a class?
  - That is, how can I write?

```
studente a("fra"),b("caf",2);
...
if(a==b) { ... }
```

Creating to variables a and b of class type.

What does means when a student is equal to another?

  - I must "instruct" the compiler on how to operate on objects of a given class, i.e. define operators with the same name of the one used for the base types: *operator overloading*.
  - For this purpose any helper or member function can be used. These methods can redefine operators.

Class

- Operators
  - To define an operator an *operator function* must be declared in the *class definition.*
  - This is just a function member having as a name an operator sign preceded by the expression: *operator*.

    type operator sign (parameter list) { /...*/ }*
  - An user can overload (re-define) operators:
    - *+, -, *, /, =, +=, -=, *=, /=, ++, --*
    - *<, >, ==, !=, <=, >=, &&, ||*
    - *<<, >>, <<=, >>=*
    - *%, &, ^, ! , |, ~, &=, ^=, |=, %=*
    - *[], (), ,, ->*, ->*
    - *new, delete, new[], delete[]*

Class

- Operators
  - For example let's define *==* for a student:

```
class studente {
...
   bool operator == ( const studente & a){
         return GetName() == a.GetName();
   }
...
};
...
studente a( "fra"),b("caf");
...
if(a==b) { ... }
if(a.operator==(b)){ ... }
```

I can use the operator! Write generic code using the same symbol!!!

I can use the member function operator, explicitly applying it to the object.

Class

- Operators
  - Let's define *==* as an helper function:

```
class studente {
...
};
bool operator == ( const studente & a, const studente & b){
        return a.GetName() == b.GetName();
   }
...
studente a( "fra"),b("caf");
...
if(a==b) { ... }
if(operator==(a,b)){ ... }
```

I can use the operator explicitly like any other function!