14/06/2017

Namespaces

- **Namespaces**
  - A namespace is a mechanism for expressing logical grouping (… hope so).
  - Often declaration are logically connected but cannot be grouped in the same module. The construct *namespace* offers support for this…
  - … grouping in a single scope definitions that the user would like to logically connect.
  - A namespace is a scope.

Namespaces

- **Namespaces**
  - A namespace can contain declarations and definitions:

```
namespace my_utils {
   int fac(int );
   int MAXINT=200;
   int pow(int ,int );
}
```

  - Names grouped in a *namespace*, can be specified using the scope resolution operator: *::*

```
int a=my_utils::fac(3);
if(a < my_utils::MAXINT ) b=my_utils::fac(a);
```

Namespaces

- **Namespaces**
  - It is possible to include declarations in a namespace scope that can be easier included into an header file. Definition can be written elsewhere but **specifying the full name**.
  - Once again interfaces can be grouped into a "lighter" (shorter) logical block. Implementation can go into "heavier" (longer) files.

my_func.h

```
namespace my_utils{
        int fac( int );
}
```

fac.cpp

```
int my_utils::fac( int b){
 return (n<2) ? 1 : n*
my_utils::fac(n-1);
}
```

Namespaces

- **Namespaces**
  - A *using-declaration* declares what are the names of one namespace you would like to use: *using full_name_to_be_declared*

```
using my_utils::fac; // from now on use the my_utlis fac
using my_utils::pow;
int a=fac(3)*pow(2,3);
```

  - Note that *using* is a declaration so it can be used anywhere in the code. Better if used where needed.
  - There is also a *using-directive* that makes avaiable all the names in namespace: *using namespace*

```
using namespace my_utils;
```

Namespaces

- **Namespaces**
  - Global, *i.e.* outside the *main* scope, *using-directives* are a tool for transition and are otherwise best avoided.

```
#include <iostream>
using namespace std; // global using-directive!!!
int main() {
...
}
```

  - Think the *using–directive* as a **tool for namespace composition** or, in a function, **for notational convenience**:

```
namespace MyMathTools{
   double sqrt(double);
// Namespace composition
   using namespace MyConversion;
   using namespace MyErrorHandlers;
    ...
}
```

```
int my_fun(double b) {
// Notational convenience
      using namespace MyMathTools;
      double a=sqrt(b)
              ...
      return my_value;
}
```

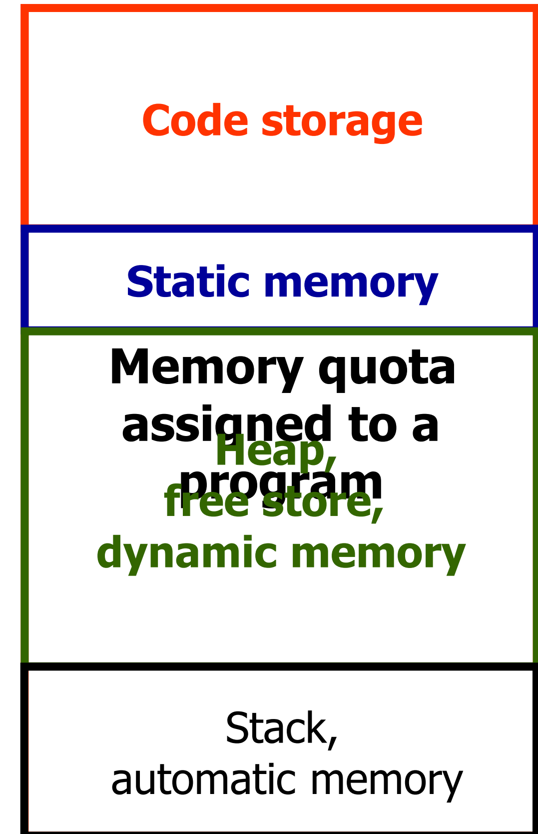
Namespaces

- **Namespaces**
  - It is possible to define *namespace aliases* to simplify typing and ease of portability. The namespace alias has to be defined using a define statement:
    namespace alias_name=actual_namespace_name;

```
// Very long name, reduced to a shorter one
namespace bop=boost::program_options;
bop::variables_map vm;
…
namespace  MyLib=IEEE_double_precision_math_v3;
double a=MyLib::sqrt(9.);
…
// Let's change the main library to test a new version
namespace  MyLib=IEEE_double_precision_math_v4;
double a=MyLib::sqrt(9.);
…
```

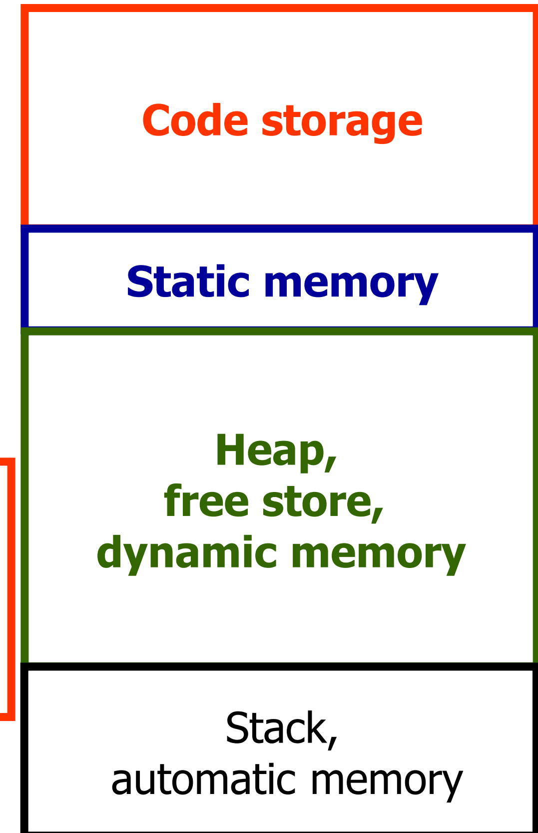
Dynamic Memory

- Ho can I dynamically manage memory allocation in my program, that is how can I create objects just when I need them? Besides, how I can let them survive the scope have been created into?
- While a C++ program is executed, a memory quota is assigned to it. This quote is partitioned into:
  - Code storage: memory area storing the code to be executed.
  - Static memory: memory area storing global objects.
  - *Stack (automatic) memory*: memory area storing local objects and function parameters.
  - *Dynamic (heap) memory*: free memory store to be allocated in case is needed.



Dynamic Memory

- To use an object outside the scope has been created into, I have few possibilities. Among these:
  - I can declare it as global
  - I can create into a memory area and make it survive to the scope is created into.
- In the latter case we can use the *dynamic* (*heap) memory* or *free store*
- It is possible to create and destroy objects in the *dynamic* (*heap) memory* using operators: new and delete.
- Note that in this case an object is created with no name. It's memory address must be assigned to a pointer otherwise it will not be possible to use it anymore (*memory leak*).
  - *new* creates an object in the heap and returns a pointer to it (to the first created object).
  - *delete* destroy and object created in the heap.



Dynamic Memory

- Warning: there is no *garbage collection*
- Users must use *delete* to be sure used memory will be re-allocated by another *new*.
- Ho can I use it? It must be specified the type to be created or the pointer to the memory area to be de-allocated:

  *double *pd=new double; *pd=3; delete pd;*

- What about arrays?
  - Functional notation must be used for *new*;
  - *delete* must be followed by the *[]* postfix: *delete [] pointer*;

    *int *i= new int(59); delete[] i;*

Dynamic Memory

```
enum {NON_CREATO};
Studente * crea_studente(bool g){
  Studente * temp=0;
  switch (g){
    case TRUE: temp=new Studente(); break;
    default: temp=0;
  }
  return temp;
}

int main() {
   ...
  bool h=TRUE;
  Studente *s=crea_studente(h);
  switch(s){
     case NON_CREATO:
      cout << " Niente da fare " << endl;
      break;
    default:
     cout << " Creato studente " << s->name << endl;
  }
  ...
  delete s;
   ...
   return 0;
}
```

A Studente is created only if is needed and will survive to the scope is created in.

Constructors

- Constructors
  - It is possible to create an object of a type T and assign a value to it using the functional notation: T(), *T(in)*
  - Be careful *T(in)* for a base type could hide implicit casts, you could experience strange or not predictable behaviors because of convertions between type of different size or because of system portability issues.
  - Infact *T(in)* is also called *functional-style cast*, that is: *T(in) == (T)in*
  - This is why I cannot inizialize a pointer using this syntax, i.e. I cannot type: *int*(20)* (you can cheat it using *typedef*)

```
void test_it(double d) {
    int i=int();
    float f=float(d); // d is truncated to a float
    int b=int(d); // d is casted to an integer
    …
}
```

User defined type: Class

- Class
  - *"The aim of the C++ class concept is to provide the programmer with a tool for creating new types that can be used as conveniently as the built-in-types",* B.S.
  - *"A type is a concrete representation of a concept. (…) A class is a user-defined type. We design a new type to provide a definition of a concept that has no direct counterpart among the built-in types.",* B.S.
  - *"A program that provides types that closely match the concepts of the application tends to be easier to understand and easier to modify than a program that does not. A well-chosen set of user-definited types makes a program more concise.",* B.S.

User defined type: Class

- Class
  - *"The fundamental idea in defining a new type is to separate the incidental details of the implementation (e.g., the layout of the data used to store an object of the type) from the properties essential to the correct use of it (e.g., the complete list of functions that can access the data). Such separation is best expressed by channeling all uses of the data structure and internal housekeeping routines through a specific interface.", B.S.*

User defined type: Class

```
struct studente {
  std::string name;
  int anno_dott;
  bool buoni_e_cattivi;
  int corsi_seguiti;
};
void Print_student_name( studente *s){
  std::cout << s->name << std::endl;
}
void Set_Name( studente *s, const std::string & n) {
  s->name=n;
}
void Add_corsi( studente &s) {
  s.corsi_seguiti++;
}
```

… the incidental details of the implementation (e.g., the layout of the data used to store an object of the type) …

No explicit connection between data and function

… from the properties essential to the correct use of it (e.g., the complete list of functions that can access the data). …

User defined type: Class

```
struct studente {
 std::string name;
 int anno_dott;
 bool buoni_e_cattivi;
 int corsi_seguiti;
};
void Print_student_name( studente *s){
  std::cout << s->name << std::endl;
}
void Set_Name( studente *s, const std::string & n) {
 s->name=n;
}
void Add_corsi( studente &s) {
s.corsi_seguiti++;
}
```

Explicit connection between data and function including functions as members of the structure.

Object name is not needed. Data are now in the same scope (namespace) of data. They “know” the name of the variable to be used for a calculation.

User defined type: Class

```
struct studente {
 std::string name;
 int anno_dott;
 bool buoni_e_cattivi;
 int corsi_seguiti;
 void Print_student_name( ){
   std::cout << name << std::endl;
 }
 void Set_Name( const std::string & n) {
  name=n;
 }
 void Add_corsi( ) {
   corsi_seguiti++;
 }
};
```

Esplict connection between data and functions including functions as members of the structure.

INFN Istituto Nazionale di Fisica Nucleare Sezione di Bari

User defined type: Class

- *class*
  - If only functions declared in a structure definition should be able to use structure data members, then we must use classes: *class*.
  - Access to any class member is not permitted to a function not declared in a class definition, unless a member is part of a list of accessible members following the label: *public*.
  - This label separates any public sector by a private one, in a class.
  - Private sector can be explicitly tagged using the label: *private*
  - *struct* is a special class case. Is a class having all public members.

INFN Istituto Nazionale di Fisica Nucleare Sezione di Bari

User defined type: Class

```
struct studente {
 std::string name;
 int anno_dott;
 bool buoni_e_cattivi;
 int corsi_seguiti;
 void Print_student_name( ){
   std::cout << name << std::endl;
 }
 void Set_Name(const std::string & n) {
  name=n;
 }
 void Add_corsi( ) {
   corsi_seguiti++;
 }
};
```

I don't want data to be accessed

Only methods can modify data

...Such separation is best expressed by channeling all uses of the data structure and internal housekeeping routines through a specific interface

```
class studente {
private:
 std::string name;
 int anno_dott;
 bool buoni_e_cattivi;
 int corsi_seguiti;
public:
void Print_student_name() ;
void Set_Name(const std::string & n);
void Add_corsi();
void Set_b_e_c()
   {buoni_e_cattivi=true;};
void Set_anno_dott(const int &i){
   anno_dott=i;};
std::string Get_Name(){ return nome;}
int Get_anno_dott(){return anno_dott;}
bool Get_b_e_c(){return buoni_e_cattivi;}
int Get_corsi(){return corsi_seguiti;}
};
```

User defined type: Class

- *Class definition*
  - The syntax: *class C { … };*
    it is called: *class definition*, because actually define a new type.
  - In reality is a *declaration*, so it can be included in a code using: *#include.*

User defined type: Class

- *Member functions*
  - Functions declated in a class definition are called: member functions.
  - They are called using the same syntax used for data members: "*.*", for an object (variable); "*->*", for pointers;
  - In any member functions member names can be used without any declaration in the formal option list (no reference to the type):

```
void Set_Name( const string & n) {
  name=n;
 }
```

  - A member function definition can be written outside a class definition, *e.g.* into another file, in this case the class name must be used.
  - A class is a namespace so the class name must be used along with the usual scope resolution operator: *::*

User defined type: Class

```
struct studente {
 std::string name;
 int anno_dott;
 bool buoni_e_cattivi;
 int corsi_seguiti;
 void Print_sudent_name();
 void Set_Name( const std::string & n) {
  name=n;
 }
 void Add_corsi();
};
```

Member functions declaration and definition in a class definition

```
void studente::Print_student_name( ){
  std::cout << name << std::endl;
 }
void studente::Add_corsi( ) {
  corsi_seguiti++;
 }
```

Member functions definition outside a class definition. It can be written in another file.

User defined type: Class

- *Helper functions*
  - Sometimes I do need functions which know how to access a class but don't need to access to the representation, i.e. members.
  - In this case they can be useful to declare them inside the class header, i.e. they are in the header file.
  - These are the so called: helper function. For example the iostream manipulators are helper functions.

```
bool IsCafagna( studente &p) {
        return p.name=="Cafagna";}
template<class T> T sum_up( const T &a, const T &b)
  { T temp(a); temp+=b; return temp;} // only
  operator += would be definited in a user defined
  class …
```

User defined type: Class

- Classes are complex objects. They contain data, methods, operators, etc. etc. Having in the same file class definitions (that is the class interface) and method definitions (that is what the class actually does) could result in a poorly readable file.
- For that usually is preferred to store only the class definition in an header file (.h, .hh, .hpp etc. etc. extensions) and method definitions in a source file (.cpp, .C, .cxx etc. etc. extensions).
- In a class definition it is best to have only method definitions that are short, only few lines of code, or can be easily inlined.

User defined type: Class

```
struct studente {
 std::string name;
 int anno_dott;
 bool buoni_e_cattivi;
 int corsi_seguiti;
 void Print_sudent_name();
 void Set_Name( const std::string & n) {
  name=n;
 }
 void Add_corsi();
};
```

Member functions declaration and definition in a class definition

studente.h

```
void studente::Print_student_name( ){
  std::cout << name << std::endl;
}
void studente::Add_corsi( ) {
  corsi_seguiti++;
}
```

Member functions definition outside a class definition. It can be written in another file.

studente.cpp

INFN Istituto Nazionale di Fisica Nucleare Sezione di Bari

User defined type: Class

- *Class definition*
  - Watch out for the so called: *one-definition rule.* A class, function, *template, enumeration,* etc. etc. can be defined only one time in a code.

User defined type: Class

- To avoid multiple definitions, preprocessor directive are used: *ifndef*, *endif* e *define*.
  - *ifndef VARIABLE:* selection directive that executes all the statements following up to an: *endif* or *else*, is found if *VARIABLE* has not be definited.
  - *define VARIABLE*: define the variable named *VARIABLE* and assign it the value: TRUE
- All the definitions in an header are included in the selection directive scope, so that it will be included in the actual code only if the variable has not be definited.

User defined type: Class

- studente.h

```
#ifndef STUDENTE_H
#define STUDENTE_H
#include <string>
using std::string;
struct studente {
   string nome;
   int anno_dott;
   bool buoni_e_cattivi;
   int corsi_seguiti;
   void Print_student_name( );
   void Set_Name( const string & n) ;
   void Add_corsi( ) ;
};
#endif //STUDENTE_H
```

INFN Istituto Nazionale di Fisica Nucleare Sezione di Bari

User defined type: Class

- Constructor
  - What about member initializations?
  - You can write a method to initialize members, but this is considered poor programming style. You might forget to call it or, even worst, call it more than once.
  - C++ support the definition of a special member having the responsibility of member initializations. This special function is the constructor.
  - It easy to recognize because is the only member not returning anything (not even void) having the very same name of the class.
  - If a class constructor is declared, then this method will be called every time an object of the class type will be initialized.

INFN Istituto Nazionale di Fisica Nucleare Sezione di Bari

User defined type: Class

- Constructor
  - I can define any number of constructors.
  - They can have parameters.
  - Compiler will try to match the most suited to be called, matching the parameter types.
  - Default constructors can be defined not having any parameters or initializing them to a default value in the formal argument declarations.

User defined type: Class

- studente.h

```
#ifndef STUDENTE_H
#define STUDENTE_H
#include <string>
using std::string;
struct studente {
  studente(string n) {nome=n; anno_dott=0; corsi_seguiti=0;
buoni_e_cattivi=TRUE;}
  studente(string n,int ad) {nome=n; anno_dott=ad; corsi_seguiti=0;
buoni_e_cattivi=TRUE;}
   string nome;
   int anno_dott;
   bool buoni_e_cattivi;
   int corsi_seguiti;
   void Print_student_name( );
   void Set_Name( const string & n) ;
   void Add_corsi( ) ;
};
#endif //STUDENTE_H
```

Constructor, same class name, i.e. type.

Initializes members and does all the actions need to correctly create an object in memory.

These are not defaults constructors, i.e. they do have a list of arguments.

User defined type: Class

- Constructor
  - Write a member with the same class name, without returning any name.
  - List the input parameters, in case there is any.
  - After the parameter list, i.e. *()*, list member initializations using the functional form. Initialization list, must be preceded by *:* . The list it is comma separated: *,*
  - After the initialization list, the scope can be written with the code to be executed upon object creation:

```
studente(){name="no name"; anno_dott=0;
   corsi_seguiti=0; buoni_e_cattivi=TRUE;}
studente(string n) {name=n; anno_dott=0;
   corsi_seguiti=0; buoni_e_cattivi=TRUE;}
studente(string n,int ad) {name=n; anno_dott=ad;
   corsi_seguiti=0; buoni_e_cattivi=TRUE;}
studente(string n, int ad, int c): name(n), anno_dott(ad),
   corsi_seguiti(c), buoni_e_cattivi(TRUE) {};
```

User defined type: Class

- Constructor
  - If no parameters are specified when the object is created, then the default constructor, if any, will be used:

    studente nessuno; // Equivalente a:studente nessuno();
  - If a constructor is specified no default constructor will be created by the compiler (See pag. 152).
  - Compiler will try to much the most suited to be called, matching the parameter types:

    studente cafagna("Francesco Saverio");

    // it will be used: studente(string)

    string nome("Francesco Saverio");

    studente cafagna(nome,1);

    // It will be used: studente(string ,int);