

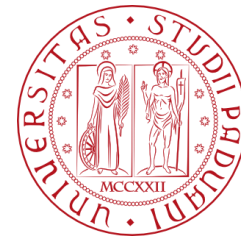
# Tutorato 1

25/10/2023

Programmazione ad Oggetti – 2023-2024

Gabriel Rovesti

2103389 – LM Computer Science



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA



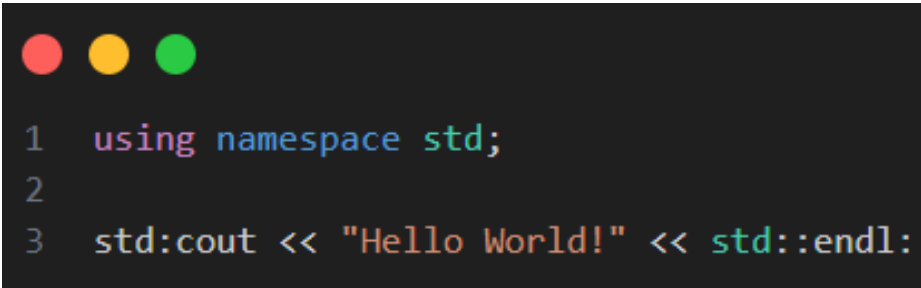
```
1 class orario
2 {
3     private:
4         int sec;
5     public:
6         int Ore() const;
7         int Minuti() const;
8         int Secondi() const;
```

**Abstract Data Type** – dato astratto per esprimere operazioni

Importante distinguere come accediamo questi dati:

- **private** (*information hiding*)
  - accessibili con operatore (.) oppure con scoping
- **public**
  - funzionalità pubblicamente accessibili)

- Usiamo una programmazione modulare, quindi utilizziamo i *namespace* per identificare una collezione di nomi ed identificatori appartenenti ad una stessa classe
- Un esempio classico è il namespace *std* (con la direttiva «*using*»)
- Spesso utilizziamo l'operatore di scoping per riferirci ai suoi membri al suo interno



```
1  using namespace std;  
2  
3  std::cout << "Hello World!" << std::endl;
```

- Importante distinguere dalla dichiarazione *inline* di metodi

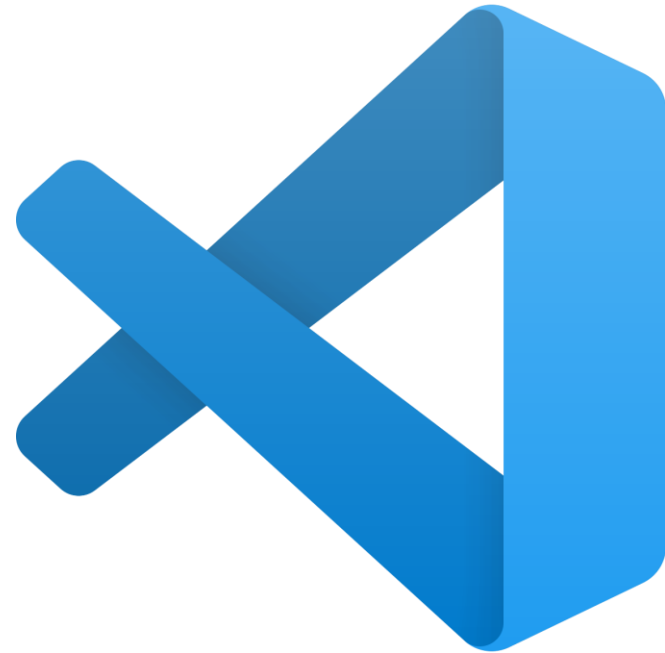
```
1  class Orario{
2      private:
3          int variabile;
4      public:
5          int metodo() const{ //dichiaro il metodo e lo uso
6              return variabile;
7          }
8  };
```

- ... dalla dichiarazione *modulare*
  - Definizione *all'esterno della classe*
  - Definizione in due file separati (.h) e (.cpp)

# Partiamo da un esempio...



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA



Classe «orario»



# Il puntatore «this»



I metodi di una classe possiedono un parametro implicito this di puntatore di tipo puntatore ad oggetti della classe stessa.

L'esempio più semplice è quello in cui un metodo deve restituire l'oggetto stesso di invocazione:

```
1 class A{
2     private: int a;
3     public: A f();
4 };
5
6 A A::f(){
7     a = 5;
8     return *this;
9 }
```



Sono dei metodi con lo stesso nome della classe e senza tipo di ritorno che vengono invocati automaticamente quando viene dichiarato (e quindi costruito) un oggetto.

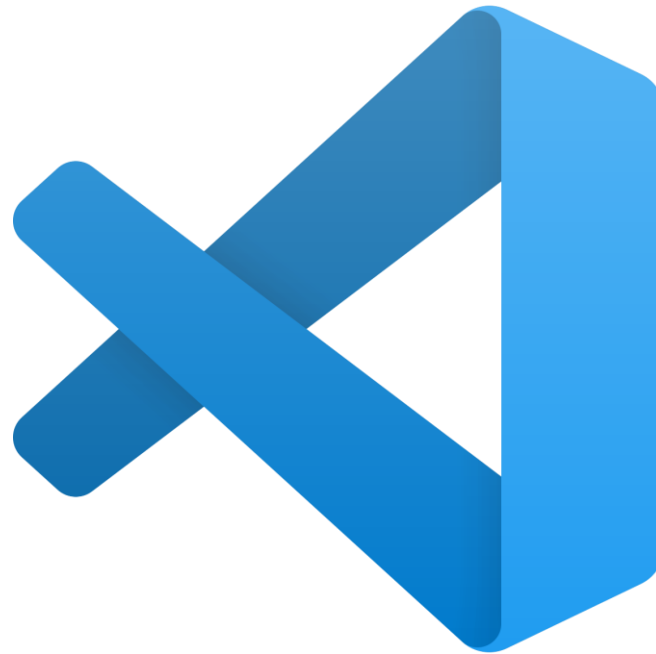
Ne esistono di vario tipo:

- **Default** (costruttore senza parametri)

```
1 class orario {  
2     public:  
3     orario();  
}
```

```
1 //costruttore di default  
2 orario::orario(){  
3     sec = 0;  
4 }
```

- Se in una classe non viene dichiarato esplicitamente alcun costruttore, è automaticamente disponibile il cosiddetto **costruttore standard**.



Andiamo al codice...



I costruttori possono essere utilizzati come:


- *convertitori di tipo*

```
1  orario::orario(int o)
2  {
3      if(o<0 || o>23) sec=0;
4      else sec=o*3600;
5  }
```

- assegnare valori a seconda del numero di parametri (**overloading**)


```
1  orario(int);
2  orario(int, int);
3  orario(int, int, int);
```

La creazione di oggetti può essere statica e prevedere la creazione di un cosiddetto oggetto anonimo, creato solo per assegnarlo ad un oggetto specifico



```
1  orario o;  
2  o = orario(12,33,25);
```

Può altrimenti essere dinamica, tramite l'operatore new.



```
1  orario* ptr = new orario;  
2  orario* ptr1 = new orario(14,25);  
3  cout << ptr->Ore(); << endl; //stampa 0  
4  cout << ptr1->Ore(); << endl; //stampa 14  
5  // ptr->Ore() è equivalente a (*ptr).Ore()
```

Possiamo facilmente assegnare valori tramite i costruttori:

```
1  orario adesso(11,55);    //costruttore ore-minuti
2
3  orario copia;            //costruttore di default
4
5  copia = adesso;          //assegnazione
6
7  orario copia1= adesso;    //costruttore di copia
8                             //analogo a: int x = y
9
10 orario copia2(adesso);    //costruttore di copia
11                             //analogo a: int x(y)
```

Si consideri che i costruttori hanno una struttura sequenziale (*che vedremo...*)

I costruttori vengono chiamati implicitamente; per evitare questo fatto, usiamo la keyword explicit.

```
class orario
{
    public:
    explicit orario(int);
    //costruttore esplicito per evitare conversioni implicite

    orario x = 8;
    // Errore: non vi è invocazione implicita del costruttore orario(8);
```

Sarebbe anche possibile definire degli operatori espliciti di conversione

```
1 class orario
2 {
3     public:
4         operator int() {return sec;}
5
6     orario o(14,37);
7     int x = o;
8     //OK: viene richiamato implicitamente l'operatore int()
9     //sull'oggetto "o"
```

# La keyword «const»

- Vogliamo ottenere **immutabilità**
  - Se nelle funzioni → ogni invocazione non farà modifiche sul parametro

```
1 class orario
2 {
3     public:
4         void StampaSecondi() const;
5         int Ore() const;
6         int Minuti() const;
7         int Secondi() const;
```

```
1 void orario::AvanzaUnOra() {sec=(sec+3600)%86400;}
2 int orario::Ore() const {return sec/3600;}
3 int orario::Minuti() const {return (sec/60)%60;}
4 int orario::Secondi() const {return sec%60;}
```

- Se sui parametri → abbiamo un valore che non può essere modificato

```
1 const orario LE_DUE(14);
2 const orario LE_TRE(15);
3 //LE_DUE = LE_TRE // Illegale, non compila
```

# Const correctness



In a nutshell, using `const` is good practice because...

0. It protects you from accidentally changing variables that aren't intended be changed,
1. It protects you from making accidental variable assignments. For instance, you are protected from

```
if( x = y ) // whoops, meant if( x == y ).
```

2. The compiler can optimize it.



```
1  class C{
2      int a[1000];
3  }
4
5  bool byValue(C x) {return true;}
6  bool byConstRef(const C& x) {return true;}
7
8  int main(){
9      C obj;
10     for (int i = 0; i < 10000000; ++i) {
11         byValue(obj); // takes between 0.5 and 1.5 seconds
12         byConstRef(obj); // takes between 0.1 and 0.2 seconds
13     }
14 }
```

# Esercizio 1: Cosa stampa (1)

```
1 using namespace std;
2
3 class C{
4     private:
5         int x;
6
7     public:
8         C(int n = 0) {x = n;}
9         C F(C obj) {C r; r.x = obj.x + x; return r;}
10        C G(C obj) const {C r; r.x = obj.x + x; return r;}
11        C H(C& obj) {obj.x += x; return obj;}
12        C I(const C& obj) {C r; r.x = obj.x + x; return r;}
13        C J(const C& obj) const {C r; r.x = obj.x + x; return r;}
```

```
1  int main() {
2      C x, y(1) , z(2) ; const C v(2);
3
4      z=x.F(y); // (1)
5      v.F(y); // (2)
6      v.G(y); // (3)
7      (v.G(y)).F(x); // (4)
8      (v.G(y)).G(x); // (5)
9      x.H(v) ; // (6)
10     x.H(z.G(y)); // (7)
11     x.I(z.G(y)); // (8)
12     x.J(z.G(y)); // (9)
13     v.J(z.G(y)); // (10)
14
15     return 0;
16 }
```



# Esercizio 1: Soluzione



```
1  int main() {
2      C x, y(1), z(2); const C v(2);
3
4      z=x.F(y); // (1) - compila
5      v.F(y); // (2) - error: passing 'const C' as 'this' argument discards qualifiers [-fpermissive]
6      v.G(y); // (3) - compila
7      (v.G(y)).F(x); // (4) - compila
8      (v.G(y)).G(x); // (5) - compila
9      x.H(v) ; // (6) - error: binding reference of type 'C&' to 'const C' discards qualifiers
10     x.H(z.G(y)); // (7) - error: cannot bind non-const lvalue reference of type 'C&' to an rvalue of type 'C'
11     x.I(z.G(y)); // (8) - compila
12     x.J(z.G(y)); // (9) - compila
13     v.J(z.G(y)); // (10) - compila
14
15     return 0;
16 }
```



# Esercizio 2: Cosa stampa (2)



```
1  C f(C& x) {return x;}
2  C% g() const {return *this;}
3  C h() const {return *this;}
4  C* m() const {return this;}
5  C* n() const {return this;}
6
7  void p(){}
8  void q() const {p();}
9
10 void p() {}
11 static void x(C* const x) {x->p();}
12
13 void s(C* const x) {*this=*x;}
14
15 static C& t() {return C();}
16
17 static C* const u(C& x) {return &x;}
```



# Esercizio 2: Soluzione



```
1  C f(C& x) {return x;}           //OK
2  C% g() const {return *this;}    //NC
3  C h() const {return *this;}     //OK
4  C* m() const {return this;}     //OK
5  C* n() const {return this;}     //NC
6
7  void p(){}
8  void q() const {p();}           //NC
9
10 void p() {}
11 static void x(C* const x) {x->p();} //OK
12
13 void s(C* const x) {*this=*x;}   //NC
14
15 static C& t() {return C();}      //NC
16
17 static C* const u(C& x) {return &x;} //OK
```



# La keyword «static»

- *Variabili*: istanza condivisa da tutte le istanze della classe invece di essere specifica di ogni istanza
- *Funzioni*: possono essere richiamate direttamente senza creare oggetti

```
1 class orario
2 {
3     public:
4     static orario OraDiPranzo();
```

```
1 orario orario::OraDiPranzo() {return orario(13, 30, 0);}
```

# La keyword «static»

- Un esempio di come un campo dati statico possa essere utilizzato per contare il numero di istanze di una classe costruite da un programma.

```
1  class C{
2      int dato;
3
4      public:
5          C(int); //costruttore ad un parametro
6          static int cont; //campo dati statico pubblico
7  };
8
9  int C::cont = 0;    //inizializzazione campo dati statico
10
11 C::C(int n){ cont++; dato=n; } //definizione costruttore
12
13 int main(){
14     C c1(1), c2(2);
15     cout << C::cont; //stampa: 2
16 }
```

# Costruttore di copia

- Casi di invocazione:

1) Quando un oggetto viene dichiarato ed inizializzato on un altro oggetto della stessa classe:

```
1  orario adesso(14,30);
2  orario copia = adesso; //inizializza copia
3  orario copia1(adesso); //inizializza copia1
```

2) Quando un oggetto viene passato *per valore* come parametro di una funzione come:

```
1  ora = ora.Somma(Ora_di_pranzo);
2
3  orario orario::Somma(orario o)
4  {
5      orario aux;
6      aux.sec=(sec+o*3600+m*60+s)%86400;
7      return aux;
8  }
```

3) Quando una funzione ritorna per valore tramite *return* un oggetto

# Operatore di assegnazione



- Assegnazione di ogni campo dati membro a membro *right to left*
- Questo il comportamento *standard*, qualora il programmatore non ridefinisca esplicitamente il suo comportamento

Lo analizzeremo più a fondo successivamente.

```
1 //Tutti gli operatori hanno una struttura
2
3 C& operator=(const C& c) {
4     if (this != &c) {
5         delete[] m_pszData;
6         m_nLength = c.m_nLength;
7         m_pszData = new char[m_nLength + 1];
8         strcpy(m_pszData, c.m_pszData);
9     }
10    return *this;
11 }
```



# Differenza tra copia e assegnazione

- Costruttore di copia: (copia) uno spazio di memoria della stessa classe per un oggetto **che non esiste già**
- Operatore di assegnazione: (assegna) uno spazio di memoria per due oggetti **già esistenti** della stessa classe

A **copy constructor** is used to initialize a **previously uninitialized** object from some other object's data.

```
A(const A& rhs) : data_(rhs.data_) {}
```

For example:

```
A aa;  
A a = aa; //copy constructor
```

An **assignment operator** is used to replace the data of a **previously initialized** object with some other object's data.

```
A& operator=(const A& rhs) {data_ = rhs.data_; return *this;}
```



## *g++ -fno-elide-constructors*

Evita la creazione di un oggetto temporaneo inutile per costruire oggetti dello stesso tipo; a fini didattici spesso non consideriamo questa (e lo usiamo nelle stampe per vedere «tutto» quello che avviene in memoria)

```
1  C fun(C a) {return a;}
2
3  int main(){
4      C c;
5      fun(c); //2 invocazioni del costr. di copia
6
7      C y = fun(c); //2 invocazioni del costr. di copia e non 3
8
9      C z; z = fun(c); //2 invocazioni del costr. di copia
10
11     fun(fun(c)); //3 invocazioni del costr. di copia e non 4
12 }
```

# Esempio 2: Cosa stampa (3)



```
1  #include <iostream>
2  using namespace std;
3  class Puntatore {
4  public:
5      int* punt;
6  };
7  int main() {
8      Puntatore x, y;
9      x.punt = new int(8);
10     y = x;
11     cout << "*(y.punt) = " << *(y.punt) << endl;
12     *(y.punt) = 3;
13     cout << "*(x.punt) = " << *(x.punt) << endl;
14 }
15
```



# Esempio 2: Soluzione

```
1  #include <iostream>
2  using namespace std;
3  class Puntatore {
4      public:
5          int* punt; // CLASSE PUNT A INT, NO COSTRUTTORI, NO OPERATORI ASS.
6                      // (METODI DEFAULT ES. COSTRUTTORE DEFAULT)
7  };
8  int main() {
9      Puntatore x, y;
10     x.punt = new int(8);
11     y = x;
12     cout << "*(y.punt) = " << *(y.punt) << endl;
13     *(y.punt) = 3;
14     // DATO CHE Y.PUNT PUNTA ALLA STESSA AREA DI MEM X.PUNT,
15     // ENTRAMBI AVRANNO LO STESSO VALORE
16     cout << "*(x.punt) = " << *(x.punt) << endl;
17 }
18
```

# Esercizio 3: Cosa stampa (4)

```
1  #include <iostream>
2  using namespace std;
3
4  class D {
5      private:
6          int z;
7      public:
8          D(int x=0): z(x) { cout << "D01 "; }
9          D(const D& a): z(a.z) { cout << "Dc "; }
10 };
11
12 class C {
13     private:
14         D d;
15     public:
16         C() : d(D(5)) { cout << "C0 "; }
17         C(int a) : d(5) { cout << "C1 "; }
18         C(const C& c) : d(c.d) { cout << "Cc "; }
19 };
```

```
1  int main() {
2      C c1;
3      cout << "UNO" << endl;
4      C c2(3);
5      cout << "DUE" << endl;
6      C c3(c2);
7      cout << "TRE" << endl;
8
9      return 0;
10 }
```

# Esercizio 3: Soluzione



```
1  int main() {
2      C c1;
3      cout << "UNO" << endl; //D01 Dc C0 UNO
4      C c2(3);
5      cout << "DUE" << endl; // D01 C1 DUE
6      C c3(c2);
7      cout << "TRE" << endl; // Dc Cc TRE
8
9      return 0;
10 }
```



# Overloading degli operatori



- Ridefinizione di funzionalità per il contesto di una classe
  - (e.g. somma di orari in Orario)
- Tali funzionalità vengono definite mediante l'uso di operatori

+ - \* / % == != < <= > >= ++ --  
<< >> = -> [] () & new delete

- Caso importante:
  - Assegnazione standard

```
1 C& operator=(const C&);
```

- Simile all'assegnazione standard, vi è il costruttore di copia

```
1 C(const C&);
```



# Overloading degli operatori



- 1) Non si possono cambiare
    - a. Posizione
    - b. Numero operandi
    - c. Precedenze e associatività
  - 2) Tra gli argomenti almeno uno *user-defined type*
  - 3) Gli operatori `=`, `[]`, `→` si possono sovraccaricare solo come metodi interni
  - 4) Non si possono sovraccaricare operatori come `.", " :: ", "sizeof", "typeid"`
- Operatori particolari
    - Operatore ternario

```
1 // Operatore ternario
2 // (condizione) ? (espressione1) : (espressione2)
3 orario = (orario < 12) ? orario : orario - 12;
```



# Overloading degli operatori



- Si possono sovraccaricare degli operatori come funzioni esterne
  - Operatore di stampa

```
1 ostream& operator<<(ostream& os, const orario& o){
2     return os << o.Ore() << ":" << o.Minuti() << ":" << o.Secondi();
3 }
```

- Altri esempi di overloading
  - operator + (somma)
  - operator == (confronto)
- Altri operatori che ci saranno utili successivamente:
  - operator \* (referenziazione)
  - operator & (dereferenziazione)
  - operator -> (selezione a membro)
  - operator [] (indicizzazione)
  - operator ++ (somma
    - Incremento prefisso
    - Incremento postfisso

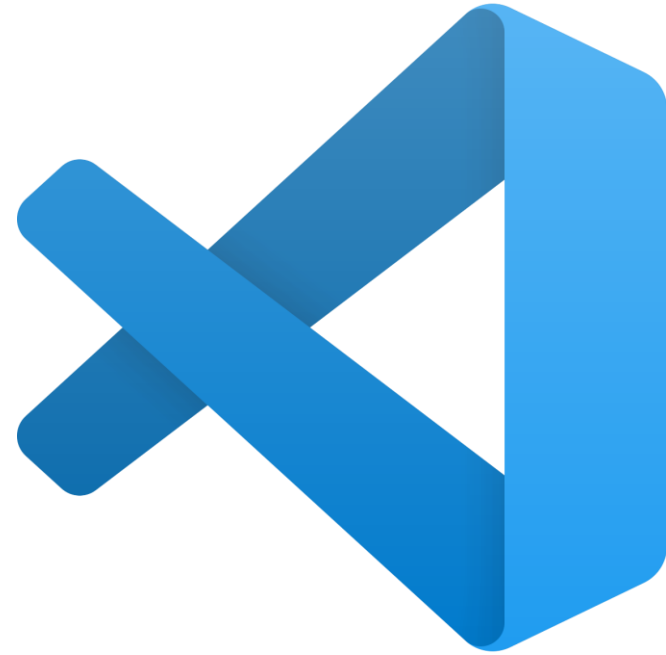




# Overloading degli operatori



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA



Estendiamo la nostra classe «orario» e vediamo alcuni esempi...



# Esercizio 4: Cosa stampa (4)

```
1  #include<iostream>
2  using std::cout;
3  class A {
4  private:
5      int x;
6  public:
7      A(int k = 5): x(k) {cout << k << " A01 ";}
8      A(const A& a): x(a.x) {cout << "Ac ";}
9      A g() const {return *this;}
10 };
11
12 class B {
13 private:
14     A ar[2];
15     static A a;
16 public:
17     B() {ar[1] = A(7); cout << "B0 ";}
18     B(const B& b) {cout << "Bc ";}
19 };
20 A B::a = A(9);
```

```
1  A Fun(A* p, const A& a, B b) {
2      *p = a;
3      a.g();
4      return *p;
5  };
6  int main() {
7      cout << "ZERO\n";
8      A a1; cout << "UNO\n";
9      A a2(3); cout << "DUE\n";
10     A* p = &a1; cout << "TRE\n";
11     B b; cout << "QUATTRO\n";
12     a1 = Fun(p,a2,b); cout << "CINQUE\n";
13     A a3 = Fun(&a1,*p,b); cout << "SEI";
14 }
```

# Esercizio 4: Soluzione

```
1  A Fun(A* p, const A& a, B b) {
2      *p = a;
3      a.g();
4      return *p;
5  };
6  int main() {
7      cout << "ZERO\n";
8      A a1; cout << "UNO\n";
9      A a2(3); cout << "DUE\n";
10     A* p = &a1; cout << "TRE\n";
11     B b; cout << "QUATTRO\n";
12     a1 = Fun(p,a2,b); cout << "CINQUE\n";
13     A a3 = Fun(&a1,*p,b); cout << "SEI";
14 }
```

```
1  /*
2      9 A01 ZERO
3      5 A01 UNO
4      3 A01 DUE
5      TRE
6      5 A01 5 A01 7 A01 B0 QUATTRO
7      5 A01 5 A01 Bc Ac Ac CINQUE
8      5 A01 5 A01 Bc Ac Ac SEI
9  */
```

# Accesso al materiale di tutorato



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA



DIPARTIMENTO  
**MATEMATICA**