

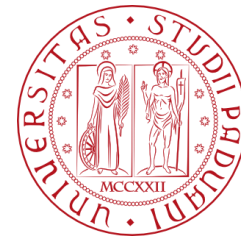
Tutorato 8

10/01/2024

Programmazione ad Oggetti – 2023-2024

Gabriel Rovesti

2103389 – LM Computer Science



UNIVERSITÀ
DEGLI STUDI
DI PADOVA



First thing first...



UNIVERSITÀ
DEGLI STUDI
DI PADOVA



Registrazione presenza Tutorato PaO

Login con SSO su Google Form inserendo i propri dati



Novità C++11: auto



- Si usa per l'inferenza automatica di tipo (comprende dal contesto quale classe diventare), evitando lo *strong typing* (quindi, dover descrivere il tipo in maniera esplicita e rigorosa)



```
1  vector<vector<int>>::const_iterator cit = v.begin();  
2  
3  auto cit = v.begin()
```

Novità C++11: default



- Per ogni classe, sono disponibili le versioni standard di:
 - Costruttore di default
 - Costruttore di copia
 - Assegnazione
 - Distruttore
- In C++11, tali funzioni si possono rendere esplicitamente di default
 - Significa quindi che vogliamo usare la versione generata dal compilatore per quella funzione, senza specificare un corpo

```
1  class A{
2      public:
3          A(int); //costruttore ad 1 parametro
4          A() = default; //costruttore altrimenti non definito
5          virtual ~A() = default; //distruttore altrimenti non definito
6  };
```



Novità C++11: delete



- In C++11, tali funzioni si possono rendere esplicitamente non disponibili
 - In questo modo, specifichiamo al compilatore *che non vogliamo* generare quella funzione automaticamente

```
1  class A{
2      public:
3          A& operator=(const A& a) = delete;
4          A(const A& a) = delete;
5  };

```

Novità C++11: override



- Lo abbiamo già visto una delle puntate precedenti, ma è una novità di C++11
- Serve a dichiarare esplicitamente quando si definisce un overriding di un metodo virtuale
- In questo modo, serve per evitare di definire o di dimenticare degli overriding

```
1  class B{
2      public:
3      virtual void m(double){}
4      virtual void f(int){}
5  };
6
7  class D: public B{
8      public:
9      virtual void m(int) override{} //illegale
10     virtual void f(int) override{} //ok
11 }
```



Novità C++11: final



- Questa keyword proibisce alle classi derivate di effettuare overriding
 - In questo modo, le funzioni virtuali non possono avere overriding in una classe derivata
- Può essere usato in un senso di ottimizzazione di compilazione (perché evita chiamate indirette ad altre sottoclassi permettendo di gestire le chiamate virtuali e risparmiare memoria – «devirtualization»)

```
3  class B{
4  public:
5      virtual void f() final;
6  };
7
8  class D : public B{
9  public:
10     void f() override; // errore: B::f() è final
11 };
12
```



Novità C++11: nullptr



- Esso ha come tipo `std::nullptr_t` e sostituisce il valore `NULL` ed il valore `0`, definendo effettivamente il caso «puntatore nullo»
- Esso è implicitamente convertibile a qualsiasi tipo puntatore ed a `bool`

```
1  // Esempio uso nullptr
2  using namespace std;
3  int main()
4  {
5      int *p = nullptr;
6      //controllo se esiste il puntatore
7      if (p == nullptr)
8          ...
9      else
10         ...
11 }
```



Funtore = Oggetto di una classe che può essere trattato come fosse una funzione

- Si può fare mediante l'overloading di operator() per avere un qualsiasi numero di parametri e ritornare qualsiasi tipo

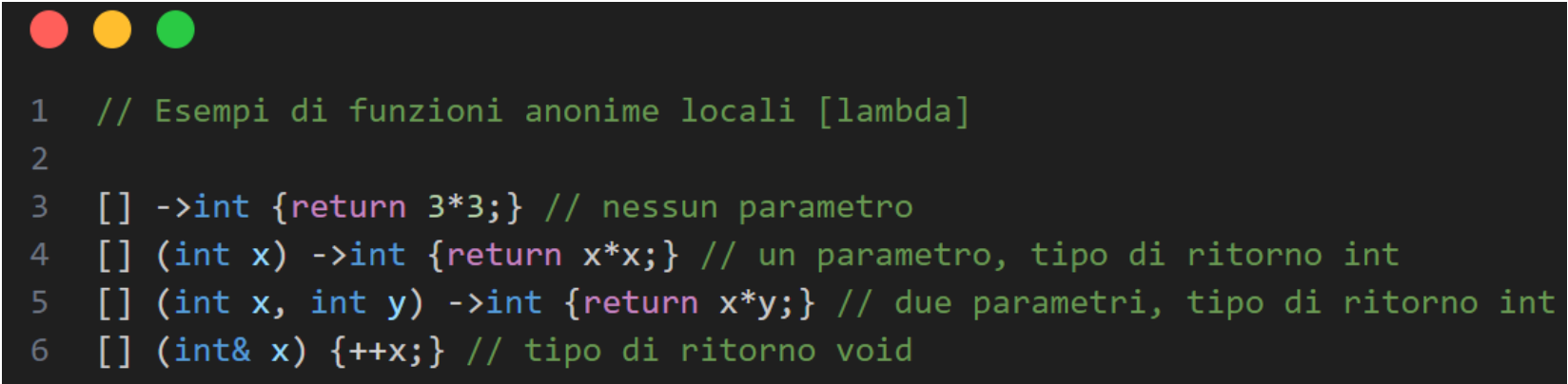
```
1  class Functor {
2  public:
3      int operator()(int x) const {
4          return x;
5      }
6  };
7
8  int main() {
9      Functor yes;
10     cout << yes(5) << endl;
11     return 0;
12 }
```

Lambda expressions

Lambda = Cosiddetti *funtori anonimi*, capiscono l'espressione in base al contesto, prendono una serie di parametri e poi ritornano un tipo.

- Struttura:

`[closure] (lista parametri) -> tipo di ritorno {corpo}`



```
1 // Esempi di funzioni anonime locali [lambda]
2
3 [] ->int {return 3*3;} // nessun parametro
4 [] (int x) ->int {return x*x;} // un parametro, tipo di ritorno int
5 [] (int x, int y) ->int {return x*y;} // due parametri, tipo di ritorno int
6 [] (int& x) {++x;} // tipo di ritorno void
```

Esempio utile: costruttore



```
class Z {  
public:  
    Z() {cout << "Z() ";}  
    Z(const Z& x) {cout << "Zc ";}  
};
```

```
class A {  
private:  
    Z w;  
public:  
    A() {cout << "A() ";}  
    A(const A& x) {cout << "Ac ";}  
};
```

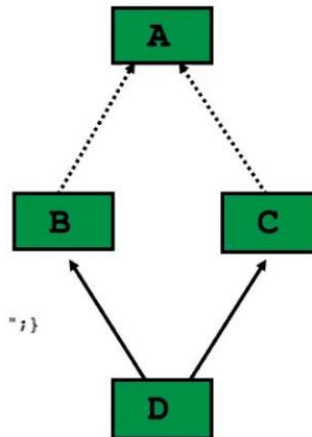
```
class B: virtual public A {  
private:  
    Z z;  
public:  
    B() {cout << "B() ";}  
    B(const B& x) {cout << "Bc ";}  
};
```

```
class C: virtual public A {  
private:  
    Z z;  
public:  
    C() {cout << "C() ";}  
};
```

```
class D: public B, public C {  
public:  
    D() {cout << "D() ";}  
    D(const D& x): C(x) {cout << "Dc ";}  
};
```

Cosa stampa?

```
D d2=d1;  
Z() A() Z() B() Zc Dc
```



Esercizio 1: Cosa Stampa

```
class A {
protected:
    virtual void j() { cout<<" A::j "; }
public:
    virtual void g() const { cout <<" A::g "; }
    virtual void f() { cout <<" A::f "; g(); j(); }
    void m() { cout <<" A::m "; g(); j(); }
    virtual void k() { cout <<" A::k "; j(); m(); }
    virtual A* n() { cout <<" A::n "; return this; }
};
```

```
class C: public A {
private:
    void j() { cout <<" C::j "; }
public:
    virtual void g() { cout <<" C::g "; }
    void m() { cout <<" C::m "; g(); j(); }
    void k() const { cout <<" C::k "; k(); }
};
```

```
class B: public A {
public:
    virtual void g() const override { cout <<" B::g "; }
    virtual void m() { cout <<" B::m "; g(); j(); }
    void k() { cout <<" B::k "; A::n(); }
    A* n() override { cout <<" B::n "; return this; }
};
```

```
class D: public B {
protected:
    void j() { cout <<" D::j "; }
public:
    B* n() final { cout <<" D::n "; return this; }
    void m() { cout <<" D::m "; g(); j(); }
};
```

```
A* p1 = new D(); A* p2 = new B(); A* p3 = new C(); B* p4 = new D(); const A* p5 = new C();
```

Le precedenti definizioni compilano correttamente. Per ognuna delle seguenti istruzioni scrivere nell'apposito spazio:

- **NON COMPILA** se la compilazione dell'istruzione provoca un errore;
- **ERRORE RUN-TIME** se l'istruzione compila correttamente ma la sua esecuzione provoca un errore a run-time;
- se l'istruzione compila correttamente e non provoca errori a run-time allora si scriva la stampa che l'esecuzione produce in output su `cout`; se non provoca alcuna stampa allora si scriva **NESSUNA STAMPA**.

Esercizio 1: Cosa Stampa

```
p1->g(); .....
```

```
p1->k(); .....
```

```
p2->f(); .....
```

```
p2->m(); .....
```

```
p3->k(); .....
```

```
p3->f(); .....
```

```
p4->m(); .....
```

```
p4->k(); .....
```

```
p5->g(); .....
```

```
(p3->n())->m();
```

```
(p3->n())->n()->g(); .....
```

```
(p4->n())->m(); .....
```

```
(p5->n())->g(); .....
```

```
(dynamic_cast<B*>(p1))->m(); .....
```

```
(static_cast<C*>(p2))->k(); .....
```

```
(static_cast<B*>(p3->n()))->g(); ..
```

Esercizio 1: Soluzione

```
1  p1->g(); // B::g
2  p1->k(); // B::k  A::n
3  p2->f(); // A::f  B::g  A::j
4  p2->m(); // A::m  B::g  A::j
5  p3->k(); // A::k  C::j  A::m  A::g  C::j
6  p3->f(); // A::f  A::g  C::j
7  p4->m(); // D::m  B::g  D::j
8  p4->k(); // B::k  A::n
9  p5->g(); // A::g
10 (p3->n())->m(); // A::n  A::m  A::g  C::j
11 (p3->n())->n()->g(); // A::n  A::n  A::g
12 (p4->n())->m(); // D::n  A::m  B::g  D::j
13 //(p5->n())->g(); // non compila (dovuto al const su this)
14 (dynamic_cast<B*>(p1))->m(); // D::m  B::g  D::j
15 //(static_cast<C*>(p2))->k(); // errore run time (chiamata infinita di k() in c)
16 (static_cast<B*>(p3->n()))->g(); cout << endl; // A::n A::g
```

Esercizio 2: Tipi

Esercizio Tipi

Siano A, B, C e D classi polimorfe distinte. Si considerino le seguenti definizioni.

```
template <class X, class Y>
X* fun(X* p) { return dynamic_cast<Y*>(p); }

int main() {
    D d; fun<A,B>(&d);
    if( fun<A,B>(new D()) == 0 ) cout << "Bjarne ";
    if( dynamic_cast<D*>(new B()) == 0 ) cout << "Stroustrup";
    A* p = fun<C,B>(new C());
}
```

Si supponga che:

1. il `main()` compili correttamente ed esegua senza provocare undefined behaviour;
2. l'esecuzione del `main()` provochi in output su `cout` la stampa Bjarne Stroustrup.

In tali ipotesi, per ognuna delle relazioni di sottotipo $T1 \leq T2$ nelle seguenti tabelle segnare con una croce l'entrata

- (a) "Vero" per indicare che $T1$ **sicuramente** è sottotipo di $T2$;
- (b) "Falso" per indicare che $T1$ **sicuramente non** è sottotipo di $T2$;
- (c) "Possibile" **altrimenti**, ovvero se non valgono né (a) né (b).

Esercizio 2: Soluzione



1 $A \leq B$ - Falso

2 $A \leq C$ - Falso

3 $A \leq D$ - Falso

4 $B \leq A$ - Vero

5 $B \leq C$ - Falso

6 $B \leq D$ - Falso



1 $C \leq A$ - Vero

2 $C \leq B$ - Possibile

3 $C \leq D$ - Falso

4 $D \leq A$ - Vero

5 $D \leq B$ - Falso

6 $D \leq C$ - Possibile



Esercizio 3: Funzione

Esercizio Funzione

```
class A {
public:
    virtual A* f() const =0;
};

class D: public B {};

class B: public A {};

class E: public B {
public:
    A* f() const {return new E();}
};

class C: public B {
public:
    B* f() const {return new C();}
};

class F: public C, public D, public E {
public:
    D* f() const {return new F();}
};
```

Queste definizioni compilano correttamente. Definire una funzione

```
list<const D *const> fun(const vector<const B*>&)
```

con il seguente comportamento: in ogni invocazione `fun(v)`, per tutti i puntatori `q` contenuti nel vector `v`:

- (A) se `q` non è nullo ed ha un tipo dinamico esattamente uguale a `C` allora `q` deve essere rimosso da `v`;
 - (A₁) se il numero N di puntatori rimossi dal vector `v` è maggiore di 2 allora viene sollevata una eccezione di tipo `C`.
- (B) sul puntatore `q` non nullo deve essere invocata la funzione virtuale pura `A* A::f()` che ritorna un puntatore che indichiamo qui con `ptr`;
 - (B₁) se `ptr` è nullo allora viene sollevata una eccezione `std::string("nullptr")`;
 - (B₂) `fun` ritorna la lista di tutti e soli questi puntatori `ptr` che: non sono nulli ed hanno un tipo dinamico che è sottotipo di `D*` e non è un sottotipo di `E*`.

Esercizio 3: Soluzione

```
1  list<const D *const> fun(const vector<const B*>&v){
2      list<const D *const> result;
3      int count = 0;
4      for (vector<const B*>::const_iterator q = v.begin(); q != v.end(); ++q) {
5          if(*q != nullptr && typeid(**q) == typeid(C)){
6              count++;
7              if(count > 2){
8                  throw C();
9              }
10             B* ptr = const_cast<B*>(*q);
11             delete ptr;
12
13         }else{
14             A* ptr = (*q)->f();
15             if(ptr == nullptr){
16                 throw std::string("nullptr");
17             }
18             //if(dynamic_cast<D*>(ptr) != nullptr && dynamic_cast<E*>(ptr) == nullptr){ //Completamente equivalente; per const correctness, viene mostrata
19             //result.push_back(dynamic_cast<D*>(ptr));
20             //}
21
22             if(dynamic_cast<const D *const>(ptr) != nullptr && dynamic_cast<const E *const>(ptr) == nullptr){
23                 result.push_back(dynamic_cast<const D *const>(ptr));
24             }
25         }
26     }
27     return result;
28 }
```

Esercizio 4: Funzione

Si assumano le seguenti specifiche riguardanti la libreria Qt (**attenzione:** non si tratta di codice da definire!).

- `QWidget` è la classe base di tutte le classi Gui della libreria Qt. La classe `QWidget` ha il distruttore virtuale. La classe `QWidget` rende disponibile un metodo `QSize size() const` con il seguente comportamento: `w.size()` ritorna un oggetto di tipo `QSize` che rappresenta la dimensione del widget `w`. Inoltre, la classe `QWidget` rende disponibile un metodo `void resize(const QSize&)` con il seguente comportamento: `w.resize(qs)` imposta la dimensione del widget `w` a `qs`. Qt rende disponibili gli operatori esterni di uguaglianza `bool operator==(const QSize&, const QSize&)` e disuguaglianza `bool operator!=(const QSize&, const QSize&)` tra oggetti di `QSize`.
- `QAbstractButton` è una classe astratta che deriva direttamente e pubblicamente da `QWidget` ed è la classe base astratta di tutti i pulsanti astratti (button widgets). La classe `QAbstractButton` rende disponibile un metodo `bool isDown() const` con il seguente comportamento: `ab.isDown()` ritorna `true` se il button `ab` è nello stato “down”, altrimenti ritorna `false`.
- `QCheckBox` è una classe concreta che deriva direttamente e pubblicamente da `QAbstractButton` e rappresenta un pulsante checkbox.
- In Qt esistono altre classi concrete che derivano direttamente e pubblicamente da `QAbstractButton` (ad esempio `QRadioButton`).
- La classe `QAbstractSlider` è una classe astratta che deriva direttamente e pubblicamente da `QWidget` ed è la classe base astratta di tutti i cursori astratti (slider widgets). La classe `QAbstractSlider` rende disponibile un metodo `void setSliderDown(bool x)` che permette di impostare lo stato “down” dello slider quando è invocato con parametro attuale `true`.
- `QSlider` è una classe concreta che deriva direttamente e pubblicamente da `QAbstractSlider`. `QSlider` è dotato di un costruttore di default.
- In Qt esistono altre classi concrete che derivano direttamente e pubblicamente da `QAbstractSlider` (ad esempio `QScrollBar`).

Esercizio 4: Funzione

Si assumano le seguenti specifiche riguardanti la libreria Qt (**attenzione:** non si tratta di codice da definire!).

- `QWidget` è la classe base di tutte le classi Gui della libreria Qt. La classe `QWidget` ha il distruttore virtuale. La classe `QWidget` rende disponibile un metodo `QSize size() const` con il seguente comportamento: `w.size()` ritorna un oggetto di tipo `QSize` che rappresenta la dimensione del widget `w`. Inoltre, la classe `QWidget` rende disponibile un metodo `void resize(const QSize&)` con il seguente comportamento: `w.resize(qs)` imposta la dimensione del widget `w` a `qs`. Qt rende disponibili gli operatori esterni di uguaglianza `bool operator==(const QSize&, const QSize&)` e disuguaglianza `bool operator!=(const QSize&, const QSize&)` tra oggetti di `QSize`.
- `QAbstractButton` è una classe astratta che deriva direttamente e pubblicamente da `QWidget` ed è la classe base astratta di tutti i pulsanti astratti (button widgets). La classe `QAbstractButton` rende disponibile un metodo `bool isDown() const` con il seguente comportamento: `ab.isDown()` ritorna `true` se il button `ab` è nello stato “down”, altrimenti ritorna `false`.
- `QCheckBox` è una classe concreta che deriva direttamente e pubblicamente da `QAbstractButton` e rappresenta un pulsante checkbox.
- In Qt esistono altre classi concrete che derivano direttamente e pubblicamente da `QAbstractButton` (ad esempio `QRadioButton`).
- La classe `QAbstractSlider` è una classe astratta che deriva direttamente e pubblicamente da `QWidget` ed è la classe base astratta di tutti i cursori astratti (slider widgets). La classe `QAbstractSlider` rende disponibile un metodo `void setSliderDown(bool x)` che permette di impostare lo stato “down” dello slider quando è invocato con parametro attuale `true`.
- `QSlider` è una classe concreta che deriva direttamente e pubblicamente da `QAbstractSlider`. `QSlider` è dotato di un costruttore di default.
- In Qt esistono altre classi concrete che derivano direttamente e pubblicamente da `QAbstractSlider` (ad esempio `QScrollBar`).

Esercizio 4: Funzione



Si assuma una situazione in cui non vi è alcuna condivisione di memoria. Definire una funzione:

```
list<QCheckBox> Fun(vector<const QWidget*>&, const QSize&)
```

con il seguente comportamento: in ogni invocazione `Fun(vec, sz)`, per ogni puntatore `p` appartenente al vector `vec`:

1. se `*p` è un qualsiasi cursore astratto (slider widget) allora:
 - se `*p` non è un `QSlider` allora viene sostituito da uno `QSlider` di default con dimensione impostata a `sz`;
 - se `*p` è invece un `QSlider` di dimensione diversa da `sz` allora la sua dimensione viene impostata a `sz`; inoltre `*p` viene impostato allo stato "down"
2. se invece `*p` non è un cursore astratto (slider widget) ma è un qualsiasi pulsante astratto (button widget) in stato "down", allora il puntatore `p` viene rimosso dal vector `vec`;
3. `Fun(vec, sz)` deve ritornare una lista di `QCheckBox` contenente una copia di tutti gli oggetti `QCheckBox` puntati da un puntatore rimosso dal vector `vec` nel precedente punto 2.



Esercizio 4: Funzione

```
1  list<QCheckBox> Fun(vector<const QWidget*>& v, const QSize& sz){
2      list<QCheckBox> res;
3      for (auto it = v.begin(); it != v.end()){
4          QAbstractSlider* s = dynamic_cast<QAbstractSlider*>(const_cast<QWidget*>(*it));
5
6          if (s){
7              if (!dynamic_cast<QSlider*>(s)){
8                  delete s;
9                  s = new QSlider();
10                 s->resize(sz);
11             } else {
12                 if (s->size() != sz){
13                     s->resize(sz);
14                     s->setSliderDown(true);
15                 }
16             }
17             it++;
18         } else {
```

Esercizio 4: Funzione



```
        QAbstractButton* b = dynamic_cast<QAbstractButton*>(const_cast<QWidget*>(*it));

        if (b && b->isDown()){
            QCheckBox* cb = dynamic_cast<QCheckBox*>(b);
            if (cb) res.push_back(*cb);
            delete b;
            it = v.erase(it);
        } else it++;
    }

    return res;
}
```

