

Di C++11 non consideriamo queste major changes:

- puntatori smart `unique_ptr<T>` `shared_ptr<T>`
- concorrenza (multithreading)
- lambda espressioni (funzioni locali, a.k.a. closures)
- rvalue reference type **T&&** (temporanei modificabili)

Compilazione C++11 (dalla versione 4.7)

g++ -std=c++11

Specifica delle eccezioni deprecata

Problemi nella specifica delle eccezioni

- **Run-time checking:** il test di conformità delle eccezioni avviene a run-time e non a compile-time, quindi non vi è una garanzia statica di conformità.
- **Run-time overhead:** Run-time checking richiede al compilatore del codice aggiuntivo che può inficiare alcune ottimizzazioni.
- **Inutilizzabile con i template:** in generale i parametri di tipo dei template non permettono di specificare le eccezioni.

Inferenza automatica di tipo

Seccature dello strong typing...

```
vector< vector<int> >::const_iterator cit=v.begin();
```



Keyword: **auto**

Dichiarazioni di variabili senza specifica del loro tipo.

```
auto x = 0;    // x ha tipo int perché 0 è un litterale di tipo int
auto c = 'f';  // char
auto d = 0.7;  // double
auto debito_nazionale = 25000000000000L; // long int
auto y = qt_obj.qt_fun(); // y ha il tipo di ritorno di qt_fun
```

Permette di evitare alcune verbosità dello strong typing, specialmente per i template.

```
void fun(const vector<int> &vi){  
    vector<int>::const_iterator ci=vi.begin();  
    ...  
}
```

// posso rimpiazzarlo con

```
void fun(const vector<int> &vi){  
    auto ci=vi.begin();  
    ...  
}
```

```
void fun(vector<int> &vi){  
    auto ci=vi.begin(); // tipo: vector<int>::iterator  
    ...  
}
```

Keyword: **decltype**

Determina staticamente il tipo di espressioni.

```
int x = 3;  
decltype(x) y = 4;
```

```
std::vector<int> v(1);  
auto a = v[0];           // a ha tipo int  
decltype(v[1]) b = 1;    // b ha tipo int  
auto c = 0;              // c ha tipo int  
auto d = c;              // d ha tipo int  
decltype(c) e;           // e ha tipo int  
decltype(0) f;           // f ha tipo int
```


Inizializzazione uniforme aggregata con { }

Inizializzazione uniforme per **array**

```
// inizializzazione di array dinamico
int* a = new int[3] {1,2,0};

class X {
    int a[4];
public:
    X() : a{1,2,3,4} {} // inizializzazione di campo dati array
};
```

Inizializzazione uniforme per **contenitori STL**

```
// inizializzazione di contenitori in C++11
std::vector<string> vs = {"first", "second", "third"};

std::map<string,string> singerPhones =
    { {"SferaEbbasta", "347 0123456"},
      {"RogerWaters", "348 9876543"} };

void fun(std::list<double> l);
fun({0.34, -3.2, 5, 4.0});
```

keywords default e delete


Per ogni classe sono disponibili le versioni **standard** di:

- 1) costruttore di default
- 2) costruttore di copia
- 3) assegnazione
- 4) distruttore

In C++11 tali funzioni standard si possono rendere esplicitamente di default oppure non disponibili.

```
class A {  
public:  
    A(int) {} // costruttore ad 1 argomento  
    A() = default; // costruttore altrimenti non disponibile  
    virtual ~A() = default; // distruttore virtuale standard  
};
```

```
class NoCopy {  
public:  
    NoCopy& operator=(const NoCopy& ) = delete;  
    NoCopy (const NoCopy&) = delete;  
};
```



```
int main() {  
    NoCopy a,b;  
    NoCopy b(a); // errore in compilazione  
    b=a; // errore in compilazione  
}
```

```
class OnlyDouble {
public:
    static void fun(double) {}
    template <class T> static void fun(T) = delete;
    // NESSUNA CONVERSIONE A DOUBLE PERMESSA
};

int main() {
    int a=5; float f=3.1;
    OnlyDouble::fun(a); // ILLEGALE: use of deleted function with T=int
    OnlyDouble::fun(f); // ILLEGALE: use of deleted function with T=float
}
```

Overriding esplicito

keyword: **override**


Per dichiarare esplicitamente quando si definisce un overriding di un metodo virtuale

```
class B {  
public:  
    virtual void m(double) {}  
    virtual void f(int) {}  
};  
  
class D: public B {  
public:  
    virtual void m(int) override {} // ILLEGALE  
    virtual void f(int) override {} // OK  
};
```

Serve per evitare di definire, o di dimenticare, inavvertitamente degli overriding

keyword: **final**

Un metodo virtuale **final** proibisce alle classi derivate di effettuare overriding

```
class B {  
public:  
    virtual void m(int) {}  
};  
  
class C: public B {  
public:  
    virtual void m(int) final {} // final override  
};  
  
class D: public C {  
public:  
    virtual void m(int) {}; // ILLEGALE   
};
```

"keyword" **override** **final**

Note that neither **override** nor **final** are language keywords. They are technically identifiers; **they only gain special meaning when used in those specific contexts. In any other location, they can be valid identifiers.**

final può permettere al compilatore una ottimizzazione di de-virtualizzazione.

Esercizio: verificare su g++/clang

Puntatori nulli

```
void f(int);  
void f(char*);  
  
int main() {  
    f(0);           // quale f invoca? invoca f(int)  
}
```

keyword: **nullptr**

Può sostituire la macro **NULL** ed il valore 0.

nullptr ha come tipo **std::nullptr_t** che è convertibile implicitamente a **qualsiasi tipo puntatore ed a bool**, mentre non è convertibile implicitamente ai tipi primitivi integrali

```
void f(int);  
void f(char*);  
  
int main() {  
    f(nullptr);     // quale f invoca? invoca f(char*)  
}
```

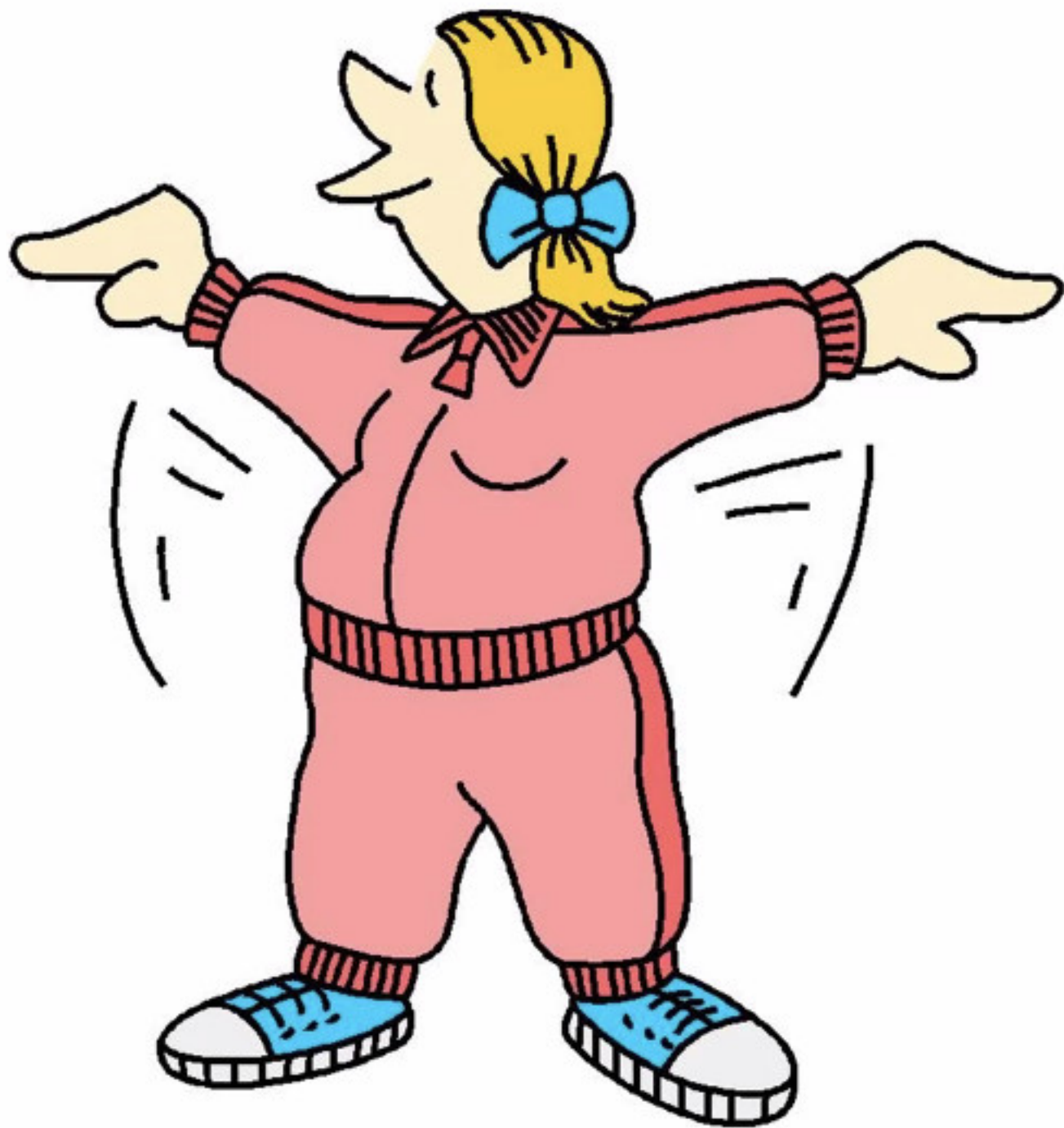
```
const char* pc = str.c_str();  
if (pc != nullptr) std::cout << pc << endl;
```

Chiamate di costruttori

Un costruttore nella sua lista di inizializzazione può invocare un **altro costruttore della stessa classe**, un meccanismo noto come **delegation** e disponibile in linguaggi come Java

```
class C {  
    int x, y;  
    char* p;  
public:  
    C(int v, int w) : x(v), y(w), p(new char [5]) {}  
    C(): C(0,0) {}  
    C(int v): C(v,0) {}  
};
```

È una alternativa al meccanismo degli argomenti di default dei costruttori; alternativa considerata **preferibile** da alcuni esperti di programmazione



Definire un template di funzione `Fun(T1*, T2&)` che ritorna un booleano con il seguente comportamento. Consideriamo una istanziamento implicita `Fun(p, r)` dove supponiamo che i parametri di tipo `T1` e `T2` siano istanziati a tipi polimorfi (cioè che contengono almeno un metodo virtuale). Allora `Fun(p, r)` ritorna `true` se e soltanto se valgono le seguenti condizioni:

1. i parametri di tipo `T1` e `T2` sono istanziati allo stesso tipo;
2. siano `D1*` il tipo dinamico di `p` e `D2&` il tipo dinamico di `r`. Allora (i) `D1` e `D2` sono lo stesso tipo e (ii) questo tipo è un sottotipo proprio della classe `ios` della gerarchia di classi di I/O (si ricordi che `ios` è la classe base astratta della gerarchia).

Ad esempio, il seguente `main()` deve compilare e provocare le stampe indicate:

```
#include<iostream>
#include<fstream>
#include<typeinfo>
using namespace std;

class C { public: virtual ~C() {} };

main() {
    ifstream f("pippo"); fstream g("pluto"), h("zagor"); iostream* p = &h;
    C c1,c2;
    cout << Fun(&cout,cin) << endl; // stampa: 0
    cout << Fun(&cout,cerr) << endl; // stampa: 1
    cout << Fun(p,h) << endl; // stampa: 0
    cout << Fun(&f,*p) << endl; // stampa: 0
    cout << Fun(&g,h) << endl; // stampa: 1
    cout << Fun(&c1,c2) << endl; // stampa: 0
}
```

/*

Definire un template di funzione Fun(T1*, T2&) che ritorna un booleano con il seguente comportamento. Consideriamo una istanziatura implicita Fun(p,r) dove supponiamo che i parametri di tipo T1 e T2 siano istanziati a tipi polimorfi (cioe` che contengono almeno un metodo virtuale). Allora Fun(p,r) ritorna true se e soltanto se valgono le seguenti condizioni:

1. i parametri di tipo T1 e T2 sono istanziati allo stesso tipo;

2. siano D1* il tipo dinamico di p e D2& il tipo dinamico di r. Allora:

(i) D1 e D2 sono lo stesso tipo

(ii) questo tipo e` un sottotipo proprio della classe ios della gerarchia di classi di I/O (si ricordi che ios e` la classe base astratta della gerarchia).

*/

```
template <class T1, class T2> bool Fun(T1* p, T2& r) {
    return typeid(p)==typeid(&r) && typeid(*p)==typeid(r) && dynamic_cast<std::ios*>(p);
}
```

```
class Z {
public: Z(int x) {}
};
```

```
class B: virtual public A {
public:
    void f(const bool&) {cout<< "B::f(const bool&) ";}
    void f(const int&) {cout<< "B::f(const int&) ";}
    virtual B* f(Z) {cout <<"B::f(Z) "; return this;}
    virtual ~B() {cout << "~B ";}
    B() {cout <<"B() "; }
};
```

```
class D: virtual public A {
public:
    virtual void f(bool) const {cout <<"D::f(bool) ";}
    A* f(Z) {cout << "D::f(Z) "; return this;}
    ~D() {cout <<"~D ";}
    D() {cout <<"D() ";}
};
```

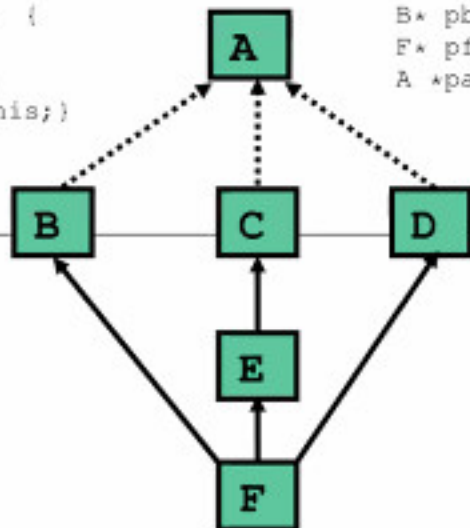
```
class F: public B, public E, public D {
public:
    void f(bool){cout<< "F::f(bool) ";}
    F* f(Z){cout <<"F::f(Z) "; return this;}
    F() {cout <<"F() "; }
    ~F() {cout <<"~F ";}
};
```

```
class A {
public:
    void f(int) {cout << "A::f(int) "; f(true);}
    virtual void f(bool) {cout <<"A::f(bool) ";}
    virtual A* f(Z) {cout <<"A::f(Z) "; f(2); return this;}
    A() {cout <<"A() "; }
};
```

```
class C: virtual public A {
public:
    C* f(Z){cout <<"C::f(Z) "; return this;}
    C() {cout <<"C() "; }
};
```

```
class E: public C {
public:
    C* f(Z){cout <<"E::f(Z) "; return this;}
    ~E() {cout <<"~E ";}
    E() {cout <<"E() ";}
};
```

```
B* pb=new B; C* pc = new C; D* pd = new D; E* pe = new E;
F* pf = new F; B *pb1= new F;
A *pa1=pb, *pa2=pc, *pa3=pd, *pa4=pe, *pa5=pf;
```



pa3->f(3);
pa5->f(3);
pb1->f(true);
pa4->f(true);
pa2->f(Z(2));
pa5->f(Z(2));
(dynamic_cast<E*>(pa4))>f(Z(2));
(dynamic_cast<C*>(pa5))>f(Z(2));
pb->f(3);
pc->f(3);
(pa4->f(Z(3)))>f(4);
(pc->f(Z(3)))>f(4);
E* puntE = new F;
A* puntA = new F;
delete pa5;
delete pb1;

cosa
stampa?

};

```
B* pb=new B; C* pc=new C; D* pd=new D;E* pe=new E; F* pf = new F; B *pbl= new F;
A *pa1=pb, *pa2=pc, *pa3=pd, *pa4=pe, *pa5=pf;
```

```
int main() {
```

```
    A* puntA = new F; // A() B() C() E() D() F()
```

```
    A* pa4 = new E;
```

```
    pa4->f(true); // A::f(bool)
```

```
    (pa4->f(Z(3)))->f(4); // E::f(Z) A::f(int) A::f(bool)
```

```
    (pa4->f(Z(3)))->f(); // E::f(Z) NON COMPILA
```

```
    (dynamic_cast<E*>(pa4))->f(Z(2)); // E::f(Z)
```

```
    pa2->f(Z(2)); // C::f(Z)
```

```
    pa5->f(Z(2)); // F::f(Z)
```

```
    (dynamic_cast<C*>(pa5))->f(Z(2)); // F::f(Z)
```

```
    (pc->f(Z(3)))->f(4); // C::f(Z) C::f(Z)
```

```
    delete pbl; // F~ B~
```



```
(pa4->f(Z(3)))->f(4); // E::f(Z) A::f(int) A::f(bool)

(pa4->f(Z(3)))->f(); // E::f(Z) NON COMPILA

(dynamic_cast<E*>(pa4))->f(Z(2)); // E::f(Z)

pa2->f(Z(2)); // C::f(Z)

pa5->f(Z(2)); // F::f(Z)

(dynamic_cast<C*>(pa5))->f(Z(2)); // F::f(Z)

(pc->f(Z(3)))->f(4); // C::f(Z) C::f(Z)

delete pb1; // F~ B~

delete pa5; // NESSUNA STAMPA !!!
```

```
pa3->f(3);

pa5->f(3);

pb1->f(true);
```