APC Cheat Sheet

Samuele Milanesi

2019/12/28

Intro

Namespaces

servono ad evitare conflitti di funzioni con lo stesso nome in librerie diverse

```
1 namespace myLibrary{
2     /* Tutto quello che sta qui dentro viene invocato
         dall'esterno tramite myLibrary::nomeMembro*/
3 }
```

Arrays

```
1 const int size(5);
2 int arr1[size]={}; // tutti gli elementi sono
     inizializzati a 0
3 int arr2[size]={2,3,1,4,5} // elementi inizializzati
     tramite lista
4 int arr3={2,3,1,4} // array inizializzato con lista,
     dimensione dedotta dalla lungezza della lista
5 int arr4[size]={1,2} // array [1,2,0,0,0]
```

Rmk. non si può inizializzare un array con un altro array: le copie vanno fatte elemento per elemento.

Dichiarazioni e definizioni

```
1 int a; int foo(double); // dichiarazioni: creano
    spazio in memoria per un oggetto non ancora
    definito
2 int a(3); int foo(double x){ return ++x;} //
    definizioni: creano e allocano un oggetto avente un
     valore
```

Rmk.

```
1 int a=5; // avviene la dichiarazione dell'int a e poi
    l'assegnamento del valore 5 ad a (costruzione
    tramite copia)
2 int a(5) // avviene la definizione di un intero di
    valore 5
```

Header files

Per chiarezza e praticità nei casi reali si usa dividere le *dichiarazioni* all'interno di un file separato detto *header* (file `.hpp`). Questo permette di vedere tutto ciò che è presente all'interno della libreria senza vederne l'implementazione. Il file `.hpp` viene poi incluso all'interno di un file `.cpp` dove vengono *implementate* tutte le funzioni dichiarate nell'header.

```
1 // file myLibrary.hpp
2 #ifndef MY_LIBRARY.h // se non è già stata definita
3 #define MY_LIBRARY.h // definisci l'header
4
5     // qui vanno le varie dichiarazioni della libreria
6
7 #endif // fine della libreria MY_LIBRARY
```

```
1 // file myLibrary.cpp
2 #include "myLibrary.hpp"
3
4 // qui vanno le varie implementazioni della libreria
```

Rinominare types

Utile quando il nome di alcuni types è complesso / non si capisce a cosa si riferisca

```
1 typedef std::vector<std::string> vet_string, *
    vet_string)_ptr; // ora posso usare il type
    vet_string per definire vettori di stringhe e
    vet_string_ptr per definire puntatori a vettori di
    stringhe
2 using vet_string =std::vector<std::string>; // metodo
    alternativo usabile da C++11 in avanti
```

Pointers & references

Basics ptrs & refs

Un *pointer* è un oggetto il cui valore è un indirizzo alla cella di memoria dell'oggetto puntato. Una *reference* è un nome con cui ci si riferisce al contenuto di una certa cella di memoria.

```
1 int x(5);
2 &x // `&` è usato come operatore di indirizzo:
    applicato alla variabile x restituisce l'indirizzo
    della cella di memoria di x.
3 int* ptr(&x); // creo un puntatore alla cella di
    memoria di x
4 *ptr // `*` è usato come operatore di
    dereferenziazione: *ptr restituisce il contenuto
    della cella di memoria puntata da ptr
5 *ptr=6; // adesso x=6
6 int& y(x) /* viene creata una reference y al contenuto
     della variabile x. Ora quel contenuto può essere
    maneggiato con due nomi diversi: `x` e `y` (oltre
    che da `*ptr`)*/
```

Rmk. mentre un pointer è un oggetto che vive di vita propria (può essere copiato, assegnato, riassegnato, ...) una reference è soltanto un *nome* dato ad un oggetto. Questo implica che 1. Le reference devono essere sempre inizializzate con il l'oggetto a cui vogliamo si riferiscano 2. Le reference non possono essere riassegnate perchè quando si usano dopo l'inizializzazione si sta usando l'oggetto

a cui si riferiscono 3. Si possono avere puntatori a puntatori ma non reference a reference 4. Si possono avere reference a puntatori

```
// continuo codice sopra
int z(9);
ptr=&z; // ok, ptr adesso punta alla cella di memoria di z
int** ptr2(ptr); // puntatore a puntatore a int
int*& ptr_newname(ptr); // il puntatore ptr adesso può essere chiamato anche con ptr_newname
y=z // il contenuto di z viene copiato in y (dunque x=y=9)
y=&z // errore si sta assegnando un int a un tipo `cella di memoria`
```

Const references

Oggetti il cui valore deve rimanere costante è bene siano dichiarati con l'attributo `const`: questo assicura che non possano essere modificati successivamente nel codice e permette al compilatore di assegnare un valore alle variabili in question at-compile-time aumentando la prestazioni del programma

```
const double PI(3.14159265);
```

Rmk. quando si usa l'attributo `const` è obbligatorio inizializzare l'oggetto costante. Se voglio creare una *reference* ad un oggetto `const` devo necessariamente creare una `const` reference

```
const double& PIGRECO(PI); //ok
double& PIGRECA(PI) // Errore! Si cerca di assegnare una reference non costante a un oggetto costante
```

D'altra parte posso assegnare ad un oggetto *non costante* una reference `const`. Questo si usa ad esempio per passare grossi oggetti a funzioni tramite reference ma assicurandosi che le operazioni fatte sull'oggetto non ne modifichino il contenuto.

```
int x(5);
const int& x_costante(x); // ok
x=6; // ok, ora x=6 e x_const=6
x_const=7; // Errore! Si cerca di usare una ref costante per modificare il contenuto
```

Gestione della memoria

La memoria si compone di:

- codice: dove c'è il file compilato con le operazioni che il programma esegue
- static data: dove sono salvati i valori delle variabili
- free store o *heap* : memoria dinamica che viene allocata su specifica richiesta del programma tramite apposita funzione (`new`) e non viene disallocata se non su specifica richiesta tramite funzione `delete`
- stack: memoria che gestisce il progresso dell'esecuzione del programma. Quando il programma chiama una funzione `f1`, lo stato del programma in quel momento viene salvato in un pacchetto che viene aggiunto alla stack, poi si passa all'esecuzione di `f1`: se questa chiama un'altra funzione `f2` lo stato di `f1` viene salvato in un pacchetto nella stack e si passa all'esecuzione di `f2` e così via. I vari pacchetti sono disallocati nel momento in cui la funzione corrisponedente termina il proprio scopo. (è il principio con cui si creano funzioni ricorsive)

Gestione dinamica della memoria

Si parla di gestione dinamica della memoria quando si creano variabili il cui contenuto è salvato nell'heap. Questo si può fare solo creando *puntatori* alla memoria in questione (non si può creare una variabile nell'heap solamente dandole un nome).

```
1 int* foo(int x)
2 {
3     int z(4);
4     int* ptr = new int(x); // creo un puntatore `ptr`
          a una cella di memoria allocata nell'heap il
          cui contenuto ha valore pari a quello di `x`.
          Si accede al contenuto tramite `*ptr`.
5     int* ptr2 = new int(x+25) // ora posso accedere
          anche tramite `*ptr2`
6     ptr=&z // `prt` ora punta al contenuto di `z`
7     return ptr2
8 }
9 /* fuori dallo scopo di `foo` le variabili `x` e `z`
     sono state cancellate dalla memoria (erano salvate
     sulla stack). Rimangono salvate sulla heap una
     variabile di valore pari a `x+25` accessibile dal
```

```
   puntatore restituito da `foo` e una variabile di
   valore pari a quello di `x` che però non è
   accessibile nè cancellabile in alcun modo perché il
    puntatore che puntava ad essa viveva sulla stack
   nello scope di `foo` e ora non esiste più. Si parla
    in questo caso di memory leak*/
```

Aritmetica dei pointers e arrays

In c++ un array non è altro che una sequenza di elementi in celle di memoria successive. Il nome con cui ci si riferisce quando si crea l'array è in realtà un puntatore al primo elemento dell'array.

Sui pointers possono essere applicati operatori aritmetici per ottenere nuovi pointers.

```
1 int x[4]={0,1,2,3}; // x punta al primo elemento di
     {0,1,2,3}
2 int*  ptr(x); // ptr punta al primo elemento di
     {0,1,2,3}
3 int* ptr2;
4 ptr2=ptr+1; // ptr2 punta alla cella di memoria subito
     dopo a quella puntata da ptr ovvero punta a dove è
     contenuto il dato `1`
5
6 // rmk: dati due ptrs ad elementi diversi di uno
     stesso array, la loro differenza dà la distanza tra
     i due elementi
```

Subscription operator

Se un puntatore punta ad un elemento di un'array si può usare l'operatore di subscription per dereferenziare gli elementi.

```
1 int x[3]={1,2,3}
2 int* ptr(x);
3 x[2] == *(ptr+2); // true
4 ptr[2] == *(ptr]2); // true
5 int* pp = &x[2]; // pp punta alla cella di memoria
     contente 3
6 pp[-1]==x[1]; // true
```

Classi

Sono user-defined types che specificano al proprio interno dei *membri* che si dividono in *metodi* (funzioni) e *dati* (variabili).

Ogni classe fornisce un'API (application program interface) ovvero un pacchetto di metodi che possono essere invocati al di fuori della classe per sfruttare le operazioni definite all'interno della classe. Si usano le API al posto che esporre tutti i membri della classe per robustezza del codice: una classe chiamata dall'esterno può fare solo quello che l'API permette di fare con una sorta di modello black-box: l'utilizzatore chiama metodi nell'API e sa che riceverà dei risultati.

Rmk. l'uso di API è utile e comune in tutte le aziende software che spesso vendono l'accesso a API del proprio software: e.g. Amazon ha delle librerie di machine learning: chi le vuole usare non otterrà tutto il codice della libreria ma solo l'API con cui richiamare dei metodi che dati degli input restituiranno degli output.

```
1 class NomeClasse{
2 private:
3     // membri privati accessibili solo da membri della
          classe stessa
4     // rmk: by default i membri sono privati
5 public:
6     // API: tutti i membri pubblici sono accessibili
          da oggetti esterni alla classe
7 }
```

Rmk. per *ogni* classe si usa fare un header diverso `NomeClasse.hpp` con un proprio file di implementazione `NomeClasse.cpp`.

L'oggetto `this`

Definendo una classe `Foo` si definisce un user-defined-type `Foo`. Gli oggetti di tipo `Foo` sono *istanze* della classe. All'interno della classe si posso creare metodi che agiscono sull'oggetto (istanza) che le invoca. In `c++` esiste un puntatore implicito che punta all'oggetto che invoca i metodi, questo puntatore si indica con `this`.

```
class Foo{
private:
    int m;
public:
    void increase_m(int m)
    {
        (this->m+=m);
    }
/* In questo codice (che non scriverei neanche se
   fosse destinato al mio peggior nemico) nella
   funzione increase_m esistono due variabili `m`: la
   prima è quella definita come parametro che vive
   solo nello scope della funzione e in tale scope è
   chiamata con `m`; la seconda è la `m` membro
   privato della classe: essa viene raggiunta dall'
   istanza che invoca increase_m tramite il puntatore
   `this`*/
}
```

Metodi costanti

In una classe ci sono funzioni che non modificano lo stato dell'istanza che le invoca: questi metodi è bene siano evidenziati con l'attributo `const`

```
class Foo{
private:
    int m;
    //...
public:
    void bar(){ ++m }; // metodo NON costante,
       modifica m
    void print(){ cout<<m<<endl } const; //metodo
       costante, non modifica lo stato dell'istanza
}
```

Friends

I membri privati di una classe sono accessibili solo da metodi della classe stessa invocati dall'istanza di cui fanno parte. Per chiarezza e minimalità del codice,

quando si definisce una classe è utile utilizzare delle funzioni *helper* esterne alla classe stessa ma che agiscono su istanze della classe. Spesso queste funzioni helper devono accedere a membri privati della classe. Questo è possibile solo definendo la funzione helper come `friend` nella classe.

```
class Foo{
private:
    int m;
    //...
public:
    friend bool bar(Foo&);
}

bool bar(Foo& obj){
    obj.m<10 ? ++(obj.m) : --(obj.m); // la funzione
       bar accede ad un membro privato dell'istanza
       obj, cosa che può fare solo se bar viene
       definita come `friend` per Foo
    return obj.m<10;
}
```

Operatori

Quando si definisce una classe si possono overloaddare operatori come `==`, `<`, `()`, ... affinché abbiano senso con lo user-defined-type creato con la classe.

Posso overloaddare un operatore in due modi: come membro della classe o come funzione helper esterna alla classe.

```
class Foo{
private:
    int m;
    //...
public:
    friend bool operator <(const Foo&, const Foo&); //
        dichiaro l'operatore come friend e lo
       definisco esternamente alla classe

    bool operator >(const Foo& rhs){
        return (this->m)> rhs.m;
    } // definisco l'operatore come membro della
       classe: la lhs in questo caso è *this
}

```

```cpp
bool operator <(const Foo& lhs, const Foo& rhs){
    return lhs.m < rhs.m;
}
```

- Operatori che **devono** essere membri: `=`, `[ ]`, `()`,`->`
- Operatori che è meglio siano membri: `+=`, `-=`, `/=`, `++`, `--`, ...
- Operatori che è meglio siano esterni: `+`, `-`, `/`, `%`, `<`, `>`, `==`, `!=`, ...
- Operatori che **non devono essere overloaddati**: `*`, `&&`, `||`, `!`

Membri statici

Ogni oggetto di una stessa classe condivide la struttura dei dati definita dalla classe stessa, ma non condivide il *valore* che questi dati hanno (sono infatti allocati in celle di memoria diversi i dati). In qualche caso può essere utile avere un membro di una classe il cui valore sia *condiviso* da tutte le istanze della classe stessa e che, quando un'istanza modifica tale valore, esso venga modificato in tutte le istanze allo stesso modo.

Questo tipo di membri sono definiti attraverso l'attributo `static`.

```cpp
class Foo{
private:
    static int cnt; // dato statico condiviso da tutte
        le istanze (può essere pubblico o privato) e
       viene solo dichiarato dentro alla classe
public:
    static void increase_cnt(){++cnt;} // metodo
       statico ogni volta che viene chiamato, aumenta
       il cnt in tutte le istanze
}
int Foo::cnt=0; // il membro statico viene
   inizializzato fuori dalla classe (una volta sola e
   di norma nel file .cpp) e non viene riportato l'
   attributo static

Foo a();
Foo b();
a.increase_cnt(); // a.cnt==b.cnt==1
Foo::increase_cnt(); // a.cnt==b.cnt==2 si possono
   chiamare i metodi statici direttamente con l'
   operatore di scope Foo:: (non essendo legati ad
   alcuna istanza in particolare)
```

Rmk. metodi statici possono modificare e maneggiare solamente dati statici: non sono legati ad un'istanza e quindi non ha senso avere riferimenti ad oggetti che sono specificati su singole istanze. Nonostante ciò questi metodi possono essere invocati attraverso istanze (un po' fuorviante ma così è...).

Constructor e destructor

Constructors

In una classe c'è un metodo pubblico speciale che prende il nome della classe stessa, non restituisce alcun type e viene chiamato ogni volta che un oggetto di quella classe è istanziato. Questo metodo prende il nome di *constructor*.

Il constructor può ricereve in input dei parametri che vengono usati per inizializzare i membri della classe.

Si può overloaddare i constructors e avere più costruttori per una stessa classe.

```
1  class Foo{
2  private:
3      int m_dato1;
4      int m_dato2=3; // in-class initializer: valore di
           default se non viene passato altro.
5      std::string m_str;
6      //...
7  public:
8      Foo(int d1, int d2, const std::string& str) //
           lista dei parametri in input
9          :m_dato1(d1),m_dato2(d2), m_str(str) //
               initializer list viene eseguita per
               primissima e definisce i dati con i valori
               in input
10         {
11             // corpo del constructor, viene eseguito
                   dopo l'initializer list
12         };
13 }
```

Rmk. l'initializer list esiste durante la costruzione dell'oggetto e quindi può inizializzare anche i membri costanti, cosa che non possono fare i normali metodi.

Default constructor e synthesized default constructor (SDC)

Ogni classe ha un constructor di default caratterizzato da

- lista di input vuota
- inizializza i membri con gli *in-class initlializers* (se presenti) o valore di default dei relativi types (e.g. una stringa viene inizializzata con una stringa vuota in assenza di in-class initializer)

Il default constructor può essere definito da noi a mano oppure può essere lasciato creare in automatico dal compilatore: in questo caso si chiama di synthesized default constructor (SDC).

Il problema è che l'SDC può dare dei problemi nei casi non banali, ad esempio:

```
1 class Foo{
2     public:
3         Foo()=default; //usare il constructor di
              default in modo esplicito (non necessario
              ma più chiaro)
4         Foo obj;
5 }
6
7 Foo() // Errore nella sintesi di un SDC: nella classe
     c'è un membro che il compilatore non sa
     inizializzare perchè non esiste nessun constructor
     di default per la classe Foo (i.e. cat that si
     mangia its own coda)
```

Delegating constructors

Quando una classe ha più di un constructor è possibile per un constructor "delegare" l'inizializzazione di alcuni parametri a constructors già esistenti.

```
1  class Foo{
2      private:
3          int m_m;
4          int m_n;
5      public:
6          Foo(int m, int n) // Constructor 1
7              : m_m(m), m_n(n){};
8
9          Foo(int m) // Constructor 2: delega al
               constructor 1 l'inizializzazione dei
               parametri richiamandolo nella sua
               initializing list
10             : Foo(m,0){};
11         Foo obj;
12 }
```

Conversioni implicite di type

Quando si definisce un costruttore *avente un unico parametro in ingeresso* si crea una relazione tra il type del parametro in ingresso e quello della classe. Salvo diversa indicazione quindi il compilatore creerà la possibilità di convertire automaticamente da type ingresso a type classe.

```
1  class Foo{
2      public:
3          int m_x
4          Foo(int x):m_x(x){}; // si crea una implicit
               conversion tra int e Foo
5
6          Foo operator +(const Foo& rhs)
7          { // definisco operatore di somma tra Foo
8              return Foo(rhs.x+this->x);
9          }
10 }
11 int main()
12 {
13 Foo a(1); Foo b(2); Foo c;
14 c= a+b; // ok, operatore di somma tra Foo ben definito
15 c= a+6; // ok, viene fatta conversione implicita di 6
       in Foo(6) e poi eseguita la somma
16 }
```

Questo tipo di operazione se ammessa non consapevolmente può portare ad errori difficili da debuggare. Per evitare ciò si può dare al costruttore che prende in ingresso un solo parametro l'attributo `explicit`: in questo modo non verrà fatta conversione implicita.

```
1  class Foo{
2      public:
3          int m_x
4          explicit Foo(int x):m_x(x){}; // no conversion
5  }
```

Copy constructor e copy-assignment operator

Copy initialization

Quando utilizziamo un oggetto per inizializzarne un altro stiamo creando una *copia* del valore dell'oggetto e allocando questa copia in memoria in un altro oggetto. Questa pratica si chiama *copy initialization*.

```
1 int x(4); // inizializzazione diretta
2 int y=x; // copy initialization
3 int z(x); // inizializzazione diretta
4 int w=4; // copy initialization
```

Quando si fa uso della *copy initialization* viene chiamato il **copy constructor** dell'oggetto.

Rmk. l'inizializzazione per copia avviene anche quando:

- passiamo un oggetto a una funzione non tramite ref.
- restituiamo un oggetto da una funzione non tramite ref.
- inizializziamo un array con una lista {el1, el2, ...}

Copy Constructor e Synthesized Copy Constructor

Un constructor è un *copy constructor* quando il suo primo parametro è una reference al type della classe e ogni altro parametro ha dei valori di default.

```
1 class Foo{
2 public:
3     Foo()=default;
4     Foo(const Foo&, int m=0); // copy constructor
5     // il type del primo parametro è una reference (
          solitamente costante)
6 }
```

Quando non viene esplicitamente definito un copy constructor per una classe, il compilatore prova a sintetizzarne uno: l'operazione ha successo se tutti i membri della classe sono copiabili, altrimenti il synthesized copy constructor non è disponibile.

Rmk. i problemi principali si hanno quando una classe ha tra i suoi membri dei puntatori: in genere si vuole che oggetti diversi assegnati tramite copia condividano il *valore* dei propri membri ma non le celle di memoria! Se ci si affida al copy constructor sintetizzato quando ci sono puntatori, questi verranno

copiati e quindi oggetti creati tramite copia condivideranno la memoria con gli oggetti sorgente usati per inizializzarli!

```
class Foo{
    public:
        int* ptr;
        Foo(int m){ *ptr = m};
        Foo(const Foo& f){
            // questo è quello che farebbe il
               synthetized copy constructor creando un
                problema dove l'oggetto creato tramite
                copia condivide la memoria con l'
               oggetto sorgente
            ptr=f.ptr;
        }
        Foo(const Foo& f): ptr(new int){
            // questa è una corretta implementazione
               di copia tramite copia dei valori
            *ptr=f->ptr;
        }
}
```

Copy-assignment operator

Per una classe si può definire un copy-assignment operator che viene usato per nei casi in cui si vuole copiare un oggetto in un altro oggetto esistente

```
Foo obj1, obj2(params);
obj1=obj2; // qui viene chiamato il copy-assignment
   operator
```

Come nel caso del copy constructor, se non viene definito dall'utente un copy-assignment operator il compilatore prova a sintetizzarne uno.

Definizione del copy-assignment operator:

```
class Foo{
    public:
        Foo& operator=(const Foo& rhs){/* ... */};
    // input: reference (generalmente costante) al
       tipo della classe (all'oggetto sorgente);
       output: reference al nuovo oggetto (*this)
}
```

Delete

Ci sono classi che non devono avere possibilità di assegnamento tramite copia. In questo caso si creano il copy constructor e il copy-assignment operator dando loro esplicitamente l'attributo `delete`

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

Destructor

Metodo speciale che viene chiamato nel momento in cui un oggetto viene eseguito per l'ultima volta. Controlla come deve essere disallocata la memoria allocata nella creazione dell'oggetto.

Sintassi:

```
1 class Foo{
2     public:
3         ~Foo(){/*corpo del destructor*/};
4 }
```

In generale è importante definire un destructor ogni qualvolta la nostra classe allochi dinamicamente della memoria: in questo caso è necessario disallocarla esplicitamente tramite `delete` nel destructor onde evitare il rischio di memory leaks (se l'oggetto alloca variabili nell'heap e viene distrutto senza disallocarle, queste non possono essere più disallocate).

Come gli altri metodi speciali, se non viene esplicitamente definito un destructor, il compilatore proverà a sintetizzarne uno.

Functions overload

Parametri di default

Nella dichiarazione di funzioni possono essere impostati dei valori di default per dei parametri: nel caso questi parametri non vengano passati dalla funzione chiamante essi assumeranno il valore di default.

```
1 void print(std::string s = "Default␣string"){
2     std::cout<< s << std::endl;
3 }
4 // chiamando print() viene stampato "Default string",
     chiamando print("ciao") viene stampato "ciao"
```

Overloading di funzioni

Ci sono casi in cui una stessa funzione vorremmo fosse ben definita su input diversi (ad esempio la funzione `somma(x,y)` vorremmo funzionasse sia nel caso in cui `x` e `y` fossero interi sia nel caso in cui fossero double).

Per farlo si possono definire più funzioni aventi *stesso nome e stesso output type* ma parametri in input diversi per type o anche per numero.

```
1 void f(int x){/*...*/};
2 void f(int x, int y){/*...*/};
3 void f(double x){/*...*/};
4 void f(){/*...*/};
5 /*quando viene chiamata `f` avviene quella che si
     chiama overload resolution: viene cercata il
     miglior match tra i parametri passati e le
     possibili dichiarazioni di `f`*/
6 f(5.6) // la funzione chiamata  è f(double) perché il
     match non richiede conversioni
```

Nella chiamata di funzioni overloaddate possono succedere tre casi:

- best match: il compilatore trova una e una sola soluzione all'overload resolution.
- no match: nessuna implementazione soddisfa la chiamata, errore.
- ambiguous match: ci sono diverse soluzioni valide all'overload resolution. Il compilatore fornisce un errore con warning al problema.

Overloading e const attribute

```
void f(int x){/*...*/};
void f(const int x){/*...*/};
/*Questo crea sempre un ambiguous match error perché `int` e `const int` non sono riconosciuti come diversi durante l'overload resolution*/
```

Quello che si può fare è overlaoddare `const ref` e `ref`

```
void f(const int& x){/*funzione che non modifica x*/};
void f(int& x){/*funzione che modifica x*/};

int z(5);
const int& z_constref(z);
f(z); // viene chiamata `f(int& x)` e `z` viene modificato da `f`
f(z_constref) // viene chiamata `f(const int& x)` e `z` non viene modificato da `f`
```

Overloading di metodi costanti

Quando l'overloading viene fatto su metodi di una classe si possono può fare overloading sul fatto che un metodo modifichi o meno lo stato dell'oggetto chiamante

```
class Bar
{
private: //...
public:
    //...
    void foo() const;
    void foo();
}

Bar a;
const Bar b;
a.foo(); // chiama la funzione non costante perché il metodo è invocato da un oggetto non costante (invocare la versione const sarebbe possibile ma non è il best match)
```

```
13 b.foo(); // chiama la funzione costante perché il
      metodo è invocato da un oggetto costante (invocare
      la versione non const non sarebbe è possibile perch
      è modificherebbe un oggetto const)
```