CONST CORRECTNESS

CONST-CORRECTNESS

- "Mi capita di imbattermi in programmatori che pensano che non valga la pena sbattersi di usare const. "E' un casino scrivere const ovunque... Se lo usassi in un posto, dovrei usarlo ovunque... Ad ogni modo, altri programmatori non lo usano e i loro programmi funzionano ugualmente... Persino alcune librerie che uso non aderiscono al paradigma del const-correctness. Quindi ne vale la pena?"
- Potremmo immaginare una scena simile, al poligono di tiro: "E' un casino essere prudenti con le armi... Ad ogni modo, conosco persone che non lo sono, e non si sono mai fatti saltare una rotula con la propia rivoltella"
- Gente non prudente con le armi non dovrebbe andare in giro a piede libero, così come muratori che non osservano le regole di sicurezza sul lavoro o elettricisti che lasciano i cavi scoperti. **Non ci sono giustificazioni per ignorare una misura di sicurezza se questa viene messa a disposizione da un prodotto, quindi non ci sono particolari scuse per i programmatori pigri che scrivono codice che non sia const-correct**.
- *Herb Sutter, in generale un tipo a posto*

CONST-CORRECTNESS

- Invece di chiederci se usare o meno const sia davvero importante, poniamoci un'altra domanda.
- Perchè usare variabili globali è considerato una cattiva pratica?

CONST-CORRECTNESS

- Invece di chiederci se usare o meno const sia davvero importante, poniamoci un'altra domanda.
- Perchè usare variabili globali è considerato una cattiva pratica?

- Le variabili **globali** possono essere lette e modificate da qualsiasi parte **del programma**, rendendo difficile ricordare o capire quale siano tutti i possibili usi
- Una variabile **globale** può essere letta e modificata da qualsiasi parte **del programma**, e qualsiasi regola riguardante il suo uso può essere violata o dimenticata

CONST-CORRECTNESS

- Invece di chiederci se usare o meno const sia davvero importante, poniamoci un'altra domanda.
- Perchè usare variabili globali è considerato una cattiva pratica?

- Le variabili **non-const** possono essere lette e modificate da qualsiasi parte **della funzione**, rendendo difficile ricordare o capire quale siano tutti i possibili usi
- Una variabile **non-const** può essere letta e modificata da qualsiasi parte **della funzione**, e qualsiasi regola riguardante il suo uso può essere violata o dimenticata

CONST-CORRECTNESS

- Gioca con noi a "Trova il bug in questa funzione":

- ```
  void f( int x, int y ) {
      if ((x==2 && y==3)||(x==1))
         std::cout << 'a' << std::endl;
      if ((y==x-1)&&(x==-1||y=-1))
         std::cout << 'b' << std::endl;
      if ((x==3)&&(y==2*x))
         std::cout << 'c' << std::endl;
  }
  ```
  ```

CONST-CORRECTNESS

- Gioca con noi a "Trova il bug in questa funzione":

```
void f( int x, int y ) {
    if ((x==2 && y==3)||(x==1))
       std::cout << 'a' << std::endl;
    if ((y==x-1)&&(x==-1||y=-1))
       std::cout << 'b' << std::endl;
    if ((x==3)&&(y==2*x))
       std::cout << 'c' << std::endl;
}
```

- Giusto per fare una piccola digressione: che razza di operatore è questo?

```
int x = 10;
x --> 1;
```

CONST-CORRECTNESS

- E se avessimo specificato const? In fondo, la funzione non modifica i valori originali che gli vengono passati...

- ```
  void f( const int x, const int y ) {
      if ((x==2 && y==3)||(x==1))
         std::cout << 'a' << std:: endl;
      if ((y==x-1)&&(x==-1||y=-1))
         std:: cout << 'b' << std:: endl;
      if ((x==3)&&(y==2*x))
          std:: cout << 'c' << std:: endl;
  }
  ```
  ```

CONST-CORRECTNESS

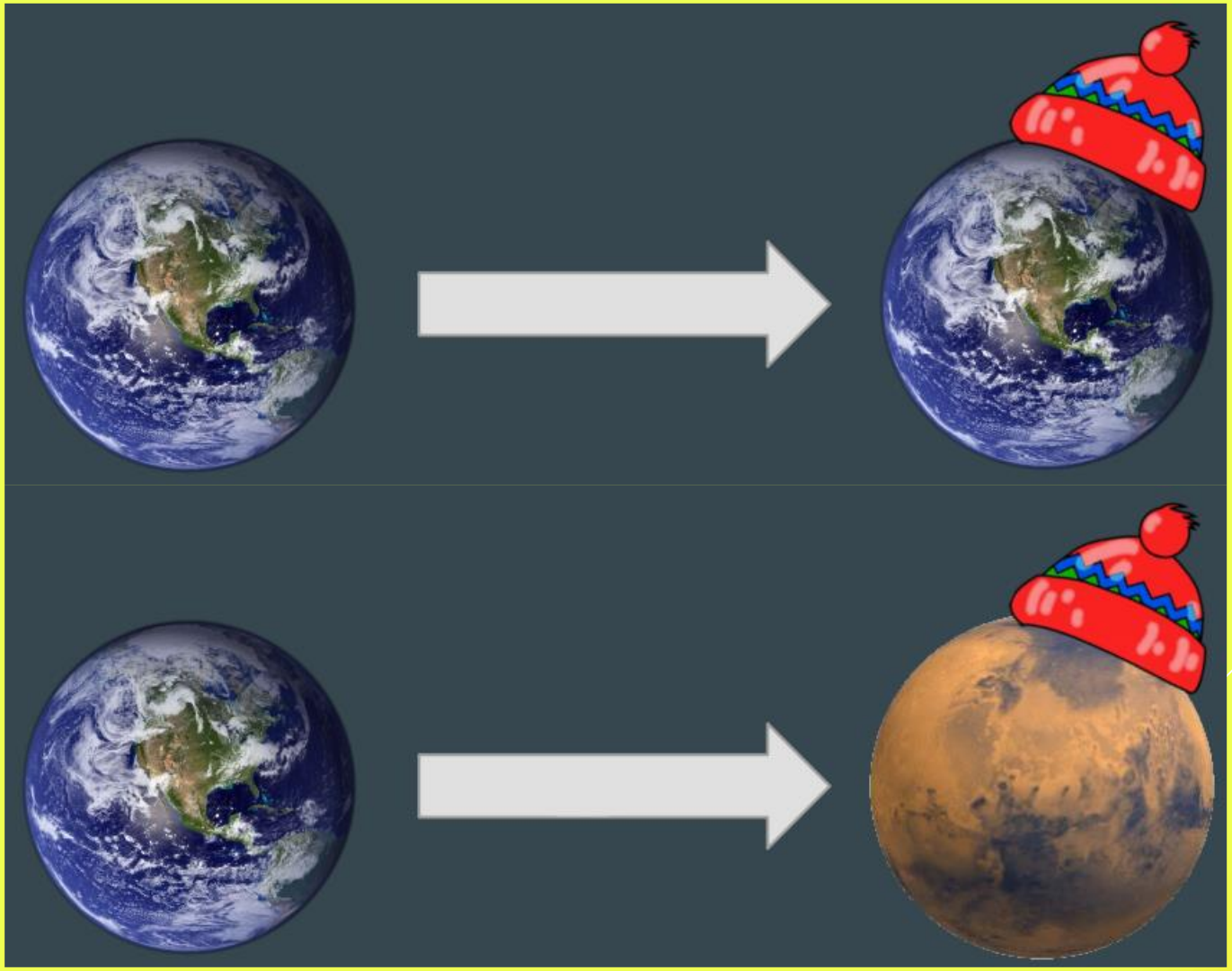
- E se avessimo specificato const? In fondo, la funzione non modifica i valori originali che gli vengono passati...

- ```
  void f(const int x, const int y) {
      if ((x==2 && y==3)||(x==1))
         std::cout << 'a' << std:: endl;
      if ((y==x-1)&&(x==-1||y=-1))
         std:: cout << 'b' << std:: endl;
      if ((x==3)&&(y==2*x))
          std:: cout << 'c' << std:: endl;
  }
  ```

- Il compilatore troverà il bug per noi!!!!
  ```

CONST-CORRECTNESS

- Questo è un caso banale, vediamo un esempio pià concreto. Usiamo una libreria che non conosciamo...

- `Pianeta terra;`

- Ho trovato una libreria per contare la popolazione dei pianeti!

CONST-CORRECTNESS

- Questo è un caso banale, vediamo un esempio pià concreto. Usiamo una libreria che non conosciamo...

- ```
  long int countPeople(Pianeta& p);
  long int popolazione = countPeople(terra);
  ```


  ```

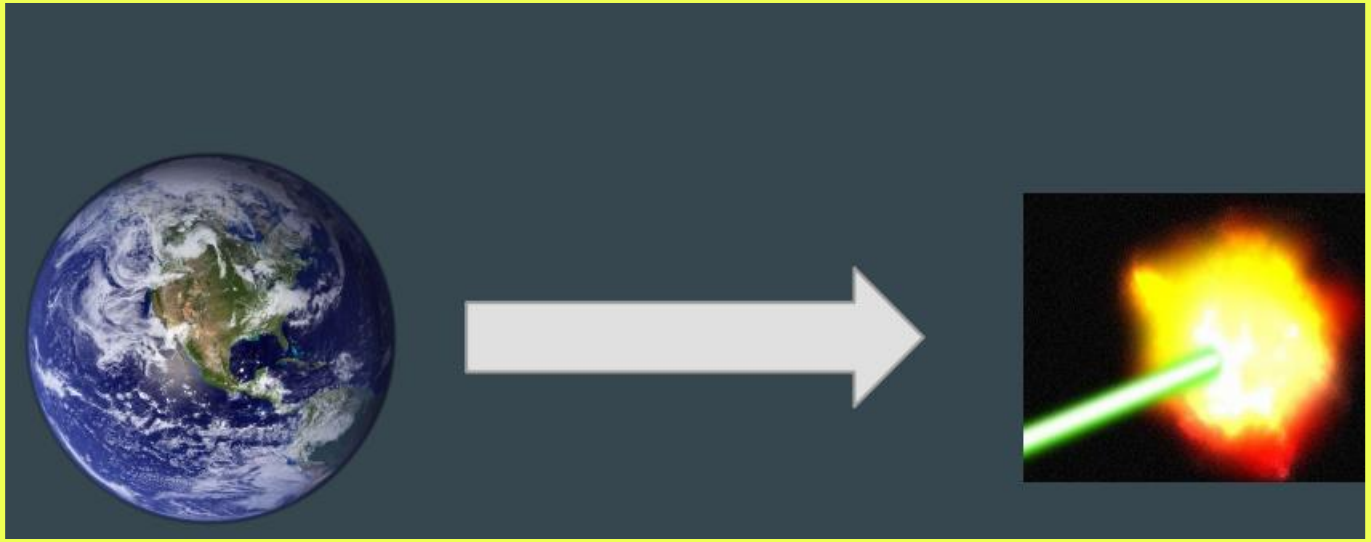
CONST-CORRECTNESS

- `long int popolazione = countPeople(terra);`



CONST-CORRECTNESS

- `long int popolazione = countPeople(terra);`



CONST-CORRECTNESS

- Come è potuto succedere?

- ```
  long int popolazione = countPeople(Pianeta& p) {
      mettiIlCappelloCheAlPoloNordFaFreddo(p);
      marsifica(p);
      raggioDellaMorte(p);
      return 0;
  }
  ```
  ```

CONST-CORRECTNESS

- E se la funzione avesse dichiarato un membro const?

- ```
  long int popolazione = countPeople(const Pianeta& p) {
      mettiIlCappelloCheAlPoloNordFaFreddo(p);
      marsifica(p);
      raggioDellaMorte(p);
      return 0;
  }
  ```
  ```

CONST-CORRECTNESS

- E se la funzione avesse dichiarato un membro const?

```
test.cpp: In function ‘long int countPopulation(const Planet&)’:

test.cpp:9:21: error: invalid initialization of reference of type
‘Planet&’ from expression of type ‘const Planet’
test.cpp:3:6: error: in passing argument 1 of ‘void addLittleHat
(Planet&)’

test.cpp:12:12: error: invalid initialization of reference of type
‘Planet&’ from expression of type ‘const Planet’
test.cpp:4:6: error: in passing argument 1 of ‘void marsify(Planet&)’

test.cpp:16:14: error: invalid initialization of reference of type
‘Planet&’ from expression of type ‘const Planet’
test.cpp:5:6: error: in passing argument 1 of ‘void deathStar(Planet&)’
```

- `const` aiuta a capire se una variabile possa essere modificata dalla funzione qualora questa venisse passata come riferimento o come puntatore

CONST-CORRECTNESS

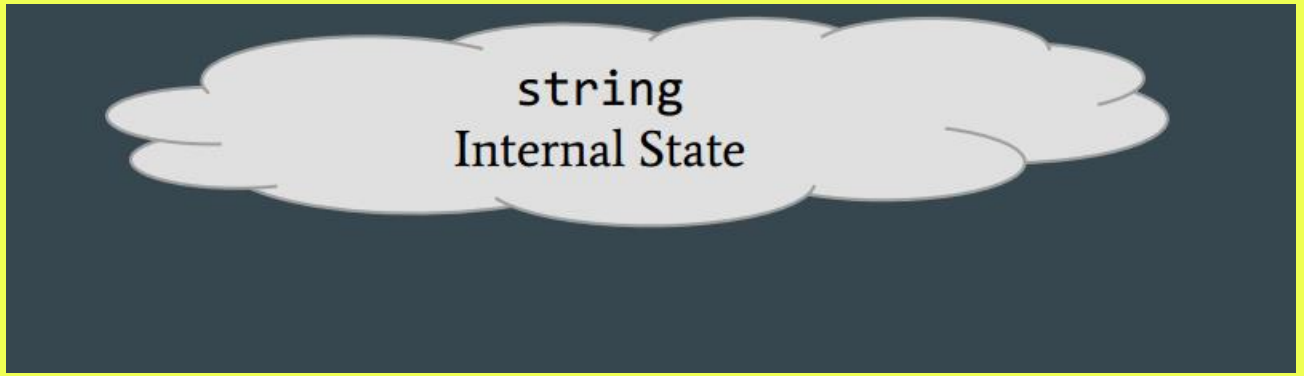
```
void f(int& x) {
    // il valore che x ha qui…
    unaFunzioneConst(x);
    unAltraFunzioneConst(x);
    // …è lo stesso che ha qui
}
```

```
void h( const int& y ) {
    // quel cavolo che voglio
}
```

```
void g() {
    int x = 42;
    h(x);
    // x è ancora 42
}
```

CONST-CORRECTNESS

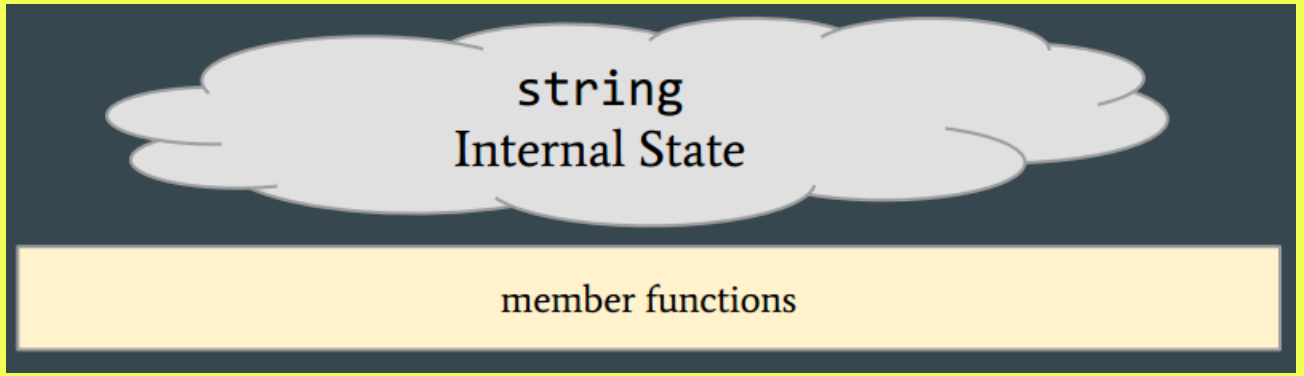
- Per gli int è da sballo, ma per gli oggetti? Come definiamo delle funzioni membro const?



- Questa è una istanza di `std::string` all'interno di un oggetto

CONST-CORRECTNESS

- Per gli int è da sballo, ma per gli oggetti? Come definiamo delle funzioni membro const?

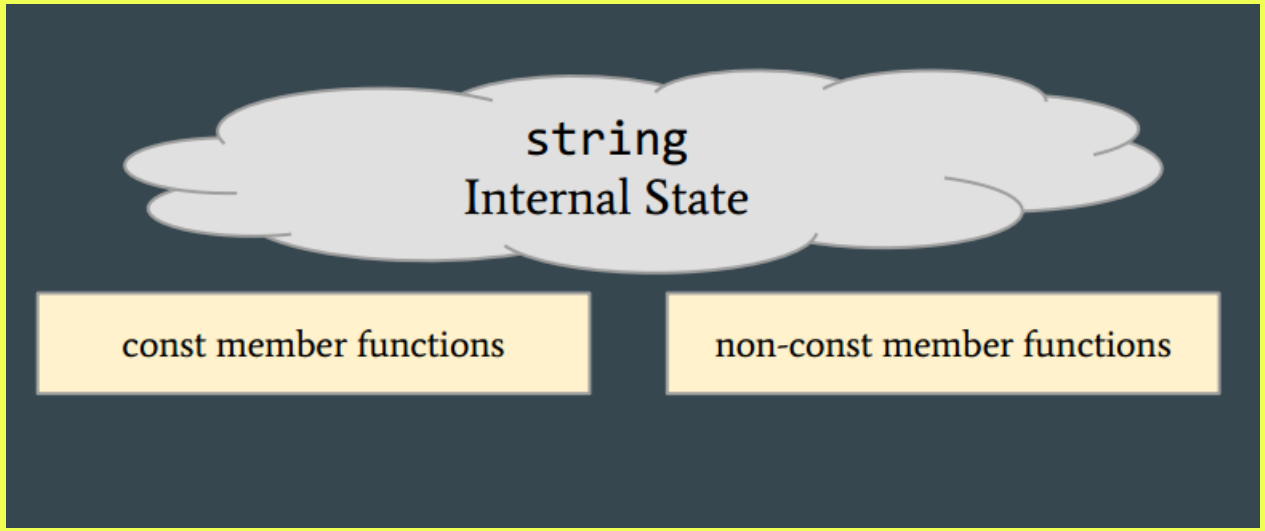


- Fino ad ora, abbiamo usato i membri funzione per interagire con l'istanza di un oggetto.

CONST-CORRECTNESS

- Per gli int è da sballo, ma per gli oggetti? Come definiamo delle funzioni membro const?

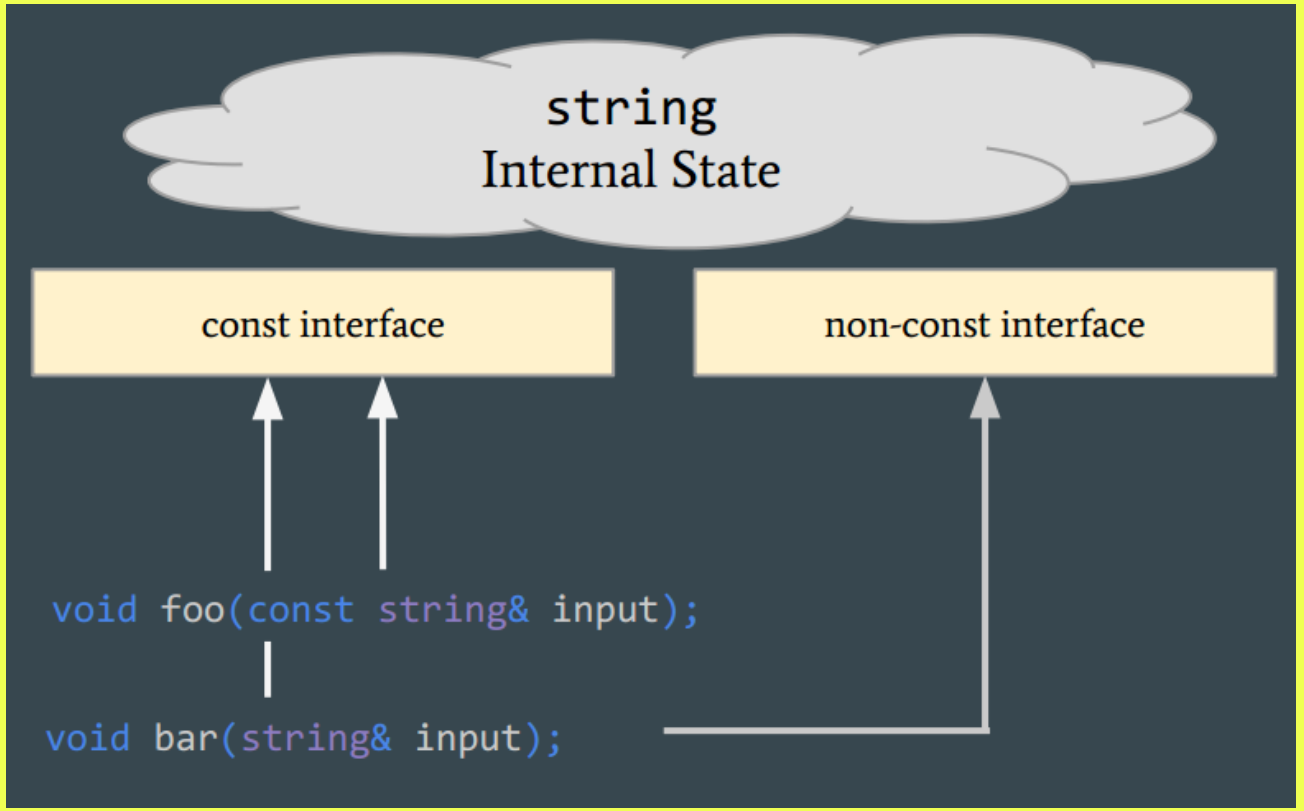
- ```
  class x {
  public:
      x(std::string s) : laMiaStringa(s) {};
      void funzione();
  private:
      std::string laMiaStringa;
  }
  ```

```
x istanz01("stringa in ingress0");
x.funzione();
```
```

CONST-CORRECTNESS

- Per gli int è da sballo, ma per gli oggetti? Come definiamo delle funzioni membro const?



- Ora vediamo che esistono membri funzione const e non-const; la particolarità è che oggetti di tipo const non possono usare funzioni membro di tipo non-const

CONST-CORRECTNESS

- Per gli int è da sballo, ma per gli oggetti? Come definiamo delle funzioni membro const?

- ```
  class x {
  public:
      x(std::string s) : laMiaStringa(s) {};
      void funzione();
      void funzione() const;
  private:
      std::string laMiaStringa;
  }

  x istanz01("stringa di input");
  istanz01.funzione();
  const x istanz02("sono una classe const!");
  istanz02.funzione();
  ```
  ```

CONST-CORRECTNESS

- Per gli int è da sballo, ma per gli oggetti? Come definiamo delle funzioni membro const?



CONST-CORRECTNESS

- Per gli int è da sballo, ma per gli oggetti? Come definiamo delle funzioni membro const?

```
// Defining const member functions
struct Planet {
    int countPopulation() const;
    void deathStar();
};

int Planet::countPopulation() const {
    return 42; // seems about right
}

void Planet::deathStar() {
    cout << "BOOM" << endl;
}
```

```
// using const member functions
struct Planet {
    int countPopulation() const;
    void deathStar();
};

void evil(const Planet &p) {
    // OK: countPopulation is const
    cout << p.countPopulation() << endl
    // NOT OK: deathStar isn't const
    p.deathStar();
}
```

CONST-CORRECTNESS

- Funzioni membro const possono essere invocate da oggetti const e non-const
- Funzioni membro non-const possono essere invocate solo da oggetti non-const

```
class C
{
    int i;
public:
    int Get() const // Note the "const" tag
      { return i; }
    void Set(int j) // Note the lack of "const"
      { i = j; }
};

void Foo(C& nonConstC, const C& constC)
{
    int y = nonConstC.Get(); // Ok
    int x = constC.Get();    // Ok: Get() is const

    nonConstC.Set(10); // Ok: nonConstC is modifiable
    constC.Set(10);    // Error! Set() is a non-const method and constC is a const-qualified object
}
```

CONST-CORRECTNESS

- Questo succede perché ogni volta che viene chiamata una funzione membro di un oggetto, implicitamente viene passato un puntatore costante all'oggetto stesso. Pensiamola così:
- `int MiaClasse::foo( int a )`

- Implicitamente equivale a:

CONST-CORRECTNESS

- Questo succede perché ogni volta che viene chiamata una funzione membro di un oggetto, implicitamente viene passato un puntatore costante all'oggetto stesso. Pensiamola così:
- `int MiaClasse::foo( int a )`

- Implicitamente equivale a:
- `int MiaClasse::foo( MiaClasse * const this, int a )`

CONST-CORRECTNESS

- Questo succede perché ogni volta che viene chiamata una funzione membro di un oggetto, implicitamente viene passato un puntatore costante all'oggetto stesso. Pensiamola così:
- `int MiaClasse::foo( int a )`

- Implicitamente equivale a:
- `int MiaClasse::foo( MiaClasse * const this, int a )`

- E metodi const?
- `int MiaClasse::foo( int a ) const`

CONST-CORRECTNESS

- Questo succede perché ogni volta che viene chiamata una funzione membro di un oggetto, implicitamente viene passato un puntatore costante all'oggetto stesso. Pensiamola così:
- `int MiaClasse::foo( int a )`

- Implicitamente equivale a:
- `int MiaClasse::foo( MiaClasse * const this, int a )`

- E metodi const?
- `int MiaClasse::foo( int a ) const`

- equivale a:
- `int MiaClasse::foo( MiaClasse const * const this, int a )`

CONST-CORRECTNESS

- Può accadere che in una classe siano definiti metodi const e non const aventi lo stesso nome e segnatura, per far fronte ad entrambi I casi (overload).
  Spesso hanno funzioni differenti!!!

```
class MyArray
{
    int data[100];
public:
    int &       Get(int i)       { return data[i]; }
    int const & Get(int i) const { return data[i]; }
};

void Foo( MyArray & array, MyArray const & constArray )
{
    // Get a reference to an array element
    // and modify its referenced value.

    array.Get( 5 )      = 42; // OK! (Calls: int & MyArray::Get(int))
    constArray.Get( 5 ) = 42; // Error! (Calls: int const & MyArray::Get(int) const)
}
```

CONST-CORRECTNESS

- Il discorso rimane valido anche per I puntatori (come visto nella lezione 0)

- `std::string * const sss;`
  puntatore costante ad una stringa non costante

- `const std::string * p;`
  `std::string const * p;`
  puntatore non costante ad una stringa costante

- `const std::string * const p;`
  `std::string const * const p;`
  puntatore costante ad una stringa costante

CONST-CORRECTNESS

- Ricordo che gli iteratori sono – in fondo – dei puntatori sotto steroidi e il discorso vale anche per loro, con qualche variante...
- `const std::vector<int>::iterator itr` si comporta come un `int* const itr` cioè un puntatore costante a vettori non costanti
- Per definire un nuovo iteratore read-only, uso `const_iterator`:

```
const std::vector<int>::iterator itr = v.begin();
*itr = 5;
++itr;
```

CONST-CORRECTNESS

- Ricordo che gli iteratori sono – in fondo – dei puntatori sotto steroidi e il discorso vale anche per loro, con qualche variante...
- `const std::vector<int>::iterator itr` si comporta come un `int* const itr` cioè un puntatore costante a vettori non costanti
- Per definire un nuovo iteratore read-only, uso `const_iterator`:

```
const std::vector<int>::iterator itr = v.begin();
*itr = 5;          // Ok, posso cambiare il valore a cui punta l'iteratore
++itr;             // Male: non posso modificare l'iteratore in sé
```

CONST-CORRECTNESS

- Ricordo che gli iteratori sono – in fondo – dei puntatori sotto steroidi e il discorso vale anche per loro, con qualche variante...
- `const std::vector<int>::iterator itr` si comporta come un `int* const itr` cioè un puntatore costante a vettori non costanti
- Per definire un nuovo iteratore read-only, uso `const_iterator`:

```
const std::vector<int>::iterator itr = v.begin();
*itr = 5;          // Ok, posso cambiare il valore a cui punta l'iteratore
++itr;             // Male: non posso modificare l'iteratore in sé
```

```
std::vector<int>::const_iterator citr = v.begin();
*itr = 5;
++itr;
int value = *itr;
```

CONST-CORRECTNESS

- Ricordo che gli iteratori sono – in fondo – dei puntatori sotto steroidi e il discorso vale anche per loro, con qualche variante...
- `const std::vector<int>::iterator itr` si comporta come un `int* const itr` cioè un puntatore costante a vettori non costanti
- Per definire un nuovo iteratore read-only, uso `const_iterator`:

```
const std::vector<int>::iterator itr = v.begin();
*itr = 5;          // Ok, posso cambiare il valore a cui punta l'iteratore
++itr;             // Male: non posso modificare l'iteratore in sé
```

```
std::vector<int>::const_iterator citr = v.begin();
*itr = 5;          // Male, non posso modificare il valore a cui punta
++itr;             // Ok, posso incrementare l'iteratore
int value = *itr; // Ok, posso leggere il valore a cui punta
```

CONST-CORRECTNESS

- Riassunto:
- `const` è un qualificatore che può essere applicato a qualsiasi tipo

```
const std::string& s = f();
```

- Può essere usato anche su funzioni:

```
size_t std::vector<int>::size() const;
```

- Buona regola qualificare con const ogni entità che non deve essere cambiata nel corso dell'esecuzione del programma.
- Passare tramite referenze const, quando possibile
- Usare `const_iterator` quando possibile
- Non distruggere il pianeta quando possibile

CONST-CORRECTNESS

- Riassunto:
- `const` sugli oggetti:
  garantisce che l'oggetto non cambia valore o aspetto, perché verranno invocati solo i membri funzioni qualificati come const e tutti i membri pubblici verranno trattati implicitamente come const.
  Questo permette di scrivere programmi più corretti e i compilatori possono produrre codice più ottimizzato
- `const` sulle funzioni
  garantisce che la funzione non cambierà qualsiasi membro non static e non mutable.

NULL E NULLPTR

- Totalmente non correlato, ma serve per arrivare a 110 slide:
- `NULL` vs. `nullptr`

- `NULL` è una macro che viene espansa in una espressione costante di tipo intero che viene valutata 0 (zero)
- `nullptr` è una keyword (da C++11). Viene implicitamente convertita in un puntatore di qualsiasi tipo a valore 0 (zero)

- Lo standard raccomanda di non usare `NULL` per indicare un puntatore nullo

NULL E NULLPTR

- Non usare NULL per indicare un puntatore nullo

- `time_t time( int seconds );`
- `time_t time( time_t* timer );`

- `[ . . . ]`

- `time( NULL );` `// Invoca la seconda funzione`
- `time( nullptr );` `// Invoca la seconda funzione`

NULL E NULLPTR

- Non usare NULL per indicare un puntatore nullo; usare nullptr

- `time_t time( int seconds );`
- `time_t time( time_t* timer );`

- `[ . . . ]`

- `time( NULL );` `// Oh, cacchio: invece invoca la prima funzione`
- `time( nullptr );` `// Invoca la seconda funzione`

- `NULL` si comporta come una costante di tipo int, mentre `nullptr` non può essere convertito ad altro se non a puntatore.