UNIVERSITÀ DEGLI STUDI DI PADOVA

Accesso ai membri di una classe

Stefano Ghidoni

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Agenda

- Accesso tramite get vs operator[]

Accesso al vettore

- Fino ad ora vector è letto e scritto con le funzioni membro get() e set()
- È più naturale un'interfaccia con operator[]
  - Lettura? Scrittura?

```
class vector {
    int sz;
    double* elem;

public:
    double operator[] (int n) { return elem[n]; }
    // ...
};
```

Accesso al vettore

- operator[] non permette di gestire questa situazione:

```
vector v(10);
double x = v[2];        // ok
v[3] = x;               // errore: v[3] non è un lvalue
```

- Possibile soluzione: usare un puntatore

```
    double* operator[] (int n) { return &elem[n]; }
```

  - Problemi?

```
*v[3] = x;              // ok, ma brutto e interazione
                        // innaturale
```

Accesso tramite reference

- Le reference risolvono il caso

```
class vector {
    int sz;
    double* elem;

public:
    double& operator[] (int n) { return elem[n]; }
    // ...
};
```

- Problemi?

Accesso tramite reference

- Le reference risolvono il caso

```
class vector {
    int sz;
    double* elem;

public:
    double& operator[] (int n) { return elem[n]; }
    // ...
};
```

- Problemi?
  - Non è utilizzabile per oggetti const

Overloading su const

- È possibile fare l'overloading di una funzione const e non const

```
class vector {
    int sz;
    double* elem;

public:
    double& operator[] (int n);
    double operator[] (int n) const;
    // ...
};
```

- Versione const: è possibile ritornare una copia o una const reference

l'overloading è possibile nonostante il numero di parametri sia lo stesso perché una funzione è const e l'altra no

Overloading su const

- Esempi:

```
void ff(const vector& cv, vector& v)
{
    double d = cv[1];          // ok: usa const []
    cv[1] = 2.0;               // errore: usa const []
    d = v[1];                  // ok: usa non-const []
    v[1] = 2.0;                // ok: usa non-const []
}
```

Reference a membro privato

- La seconda versione di operator[] ritorna una reference a membro privato
  - È corretto?
  - Ci sono caveat?

Reference a membro privato

- La seconda versione di operator[] ritorna una reference a membro privato
  - È corretto?
  - Ci sono caveat?

https://stackoverflow.com/questions/4706788/why-can-i-expose-private-members-when-i-return-a-reference-from-a-public-member

https://stackoverflow.com/questions/8005514/is-returning-references-of-member-variables-bad-practice

Reference a membro privato

5 Answers

Active | Oldest | Votes

25

`private` does not mean "this memory may only be modified by member functions" -- it means "direct attempts to access this variable will result in a compile error". When you expose a reference to the object, you have effectively exposed the object.

> Is it a bad practice to receive a returned private variable by reference ?

No, it depends on what you want. Things like `std::vector<t>::operator[]` would be quite difficult to implement if they couldn't return a non-`const` reference :) If you want to return a reference and don't want clients to be able to modify it, simply make it a `const` reference.

Share Improve this answer Follow

answered Jan 16 '11 at 17:27

Billy ONeal

99.3k ● 47 ● 299 ● 534

For people passing by : this question suggests it can be dangerous to return a reference to your class members. – Arthur Jun 12 '13 at 23:19

@Arthur: Yes, as I said, it depends on what you want to do. It works just fine for `std::vector` -- but there are plenty of cases where it is the wrong thing to do too. – Billy ONeal Jun 12 '13 at 23:28

https://stackoverflow.com/questions/4706788/why-can-i-expose-private-members-when-i-return-a-reference-from-a-public-member

UNIVERSITÀ DEGLI STUDI DI PADOVA

Reference a membro privato

63

There are several reasons why returning references (or pointers) to the internals of a class are bad. Starting with (what I consider to be) the most important:

1. **Encapsulation** is breached: you leak an implementation detail, which means that you can no longer alter your class internals as you wish. If you decided not to store `first_` for example, but to compute it on the fly, how would you return a reference to it ? You cannot, thus you're stuck.
2. **Invariant** are no longer sustainable (in case of non-const reference): anybody may access and modify the attribute referred to at will, thus you cannot "monitor" its changes. It means that you cannot maintain an invariant of which this attribute is part. Essentially, your class is turning into a blob.
3. **Lifetime** issues spring up: it's easy to keep a reference or pointer to the attribute after the original object they belong to ceased to exist. This is of course undefined behavior. Most compilers will attempt to warn about keeping references to objects on the stack, for example, but I know of no compiler that managed to produce such warnings for references returned by functions or methods: you're on your own.

As such, it is usually better not to give away references or pointers to attributes. *Not even const ones!*

For small values, it is generally sufficient to pass them by copy (both `in` and `out`), especially now with move semantics (on the way in).

For larger values, it really depends on the situation, sometimes a Proxy might alleviate your troubles.

https://stackoverflow.com/questions/8005514/is-returning-references-of-member-variables-bad-practice

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Accesso ai membri di una classe

Stefano Ghidoni

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE