C++ Day 2016

29 Ottobre / Firenze

# Le API e il miele

## Come scrivere un API senza rimanere invischiati

Marco Foco



www.italiancpp.org

# Forse non tutti sanno che…

Vivo in svizzera

Lavoro per NVIDIA

Sono un APIcoltore

Voglio diventare ricco vendendo un software per gestire la produzione di miele.

# Un database per... contarli

```
class MyDB : public map<string, long> {
public:
    long total () const { ... }
};


int main() {
    MyDB db;

    db["Luigi"] = 1;
    db["Giovanni"] = 3;
    db["Beppe 1"] = 6;
    db["Beppe 2"] = 5;
    db["Beppe 3"] = 7;

    cout << db.total() << endl;
}
```

# I miei utenti

Me

Mio cugino

Il mio cane

# Feedback

## Mio cugino: non compila

- Risolvo i problemi del tipo “a casa compilava”  e/o “a casa funzionava”
  - Esplicito tutti gli include files necessari

## Il mio cane: Woof (Traduzione: Nel mio ho aggiunto una funzione)

```
void view_db(const MyDB &db) {
    for(auto &x: db) {
        cout << x.first << '\t' << x.second << endl;
    }
}
```

# Conclusioni

Tutto funziona bene

Un “cliente” ha aggiunto una personalizzazione

Tutto va a gonfie vele

Diventerò a breve ricco, vendendo vasetti di miele e software

50’ per le domande

# Primi veri clienti

Non voglio dare il sorgente del “core”

- Separo il sorgente dall’implementazione
- Compilo il “core” a casa, producendo una libreria
- Gli do il file .h contenente le dichiarazioni

Win?

# Library.h

```
#include <map>
#include <string>


using namespace std;


class MyDB : map<string, long> {
public:
    long total() const;
};
```

# Cliente A

Ho personalizzato il programma, e ora ricevo errori

```
#include <OtherLibrary.h>
#include <Library.h>
#include <algorithm>

using namespace otherlibrary;
…
    int x = min(x1, x2); // min is ambiguous!
…
```

# Cliente A

Ho personalizzato il programma, e ora ricevo errori

# Problema

```
#include <map>
#include <string>

using namespace std;

class MyDB : map<string, long> {
public:
    long total() const;
};
```

# Soluzione

```
#include <map>
#include <string>


using namespace std;


class MyDB : std::map<std::string, long> {
public:
    long total() const;
};
```

# Cliente B

Compila

Linka

Crasha

# Cliente B

Compila

Linka

Crasha chiamando total

# Problema

Una diversa versione del compilatore ha una definizione di std::map diversa, con layout di memoria diverso

- L'allocazione avviene con la libreria standard del cliente B
- La funzione total usa la libreria standard con cui ho compilato la mia libreria

```
#include <map>
#include <string>


class MyDB : std::map<std::string, long> {
public:
    long total() const;
};
```

# Soluzione?

Risolvo spostando `std::map` all'interno della classe

```
class MyDB {
    std::map<std::string, long> _db;
public:
    long total() const;
};
```

# Oops…

```
int main() {
    MyDB db;

…

    db["Luigi"] = 1; // no operator[]
```

# Soluzione?

Anche le chiamate all'operatore [] avvenivano con la libreria del cliente B

- Devo ripristinare l'operatore []

```
class MyDB {
    std::map<std::string, long> _db;
public:
    long total() const;
    long &operator[](const string &s);
};
```

# Cliente C

Compila

Linka

Esegue il main()

Visualizza tutto correttamente

Crash in uscita

# Cliente C

Compila

Linka

Esegue il main()

Visualizza tutto correttamente

Crash in uscita (memoria corrotta)

# Problema

Una diversa versione del compilatore ha una definizione di `map` diversa, con **una dimensione diversa. La dimensione dell'oggetto è determinata dalla dimensione di** `map`

- Viene allocata la quantità di memoria determinata della libreria standard del cliente C (1)
- Viene usata la quantità di memoria memoria con il layout e la dimensione della libreria standard con cui ho compilato la libreria (2)

Se (1) < (2) scrivo fuori dalla memoria riservata alla mappa.

# Soluzione (PIMPL) – interfaccia

```
#include <string>


class MyDB {
    MyDBImpl *impl; // dimensione fissa
public:
    long total() const;
    long &operator[](const std::string &s);
    MyDB();
    virtual ~MyDB();
};
```

# Soluzione (PIMPL) – implementazione

```
#include <string>
#include <numeric>
#include <map>
#include "Library.h"

using namespace std;

class MyDBImpl : map<string, long> {
public:
    long total () const { … }
};

long MyDB::total () const { return impl->total(); }
long &MyDB::operator[](const string &s) { return (*impl)[s]; }
MyDB::MyDB() : impl(new MyDBImpl()) {}
MyDB::~MyDB() { delete impl; }
```

# Cliente D

Crasha nell'inserimento

- Ormai abbiamo capito

```
class MyDB {
    MyDBImpl *impl;
public:
    long total() const;
    long &operator[](const std::string &s);
    MyDB();
    virtual ~MyDB();
};
```

# Soluzione

```
class MyDB {
    MyDBImpl *impl;
public:
    int total() const;
    int &operator[](const char *s);
    MyDB();
    virtual ~MyDB();
};
```

# Cliente D

Ok, I crash sono risolti, ma ora la funzione total() ritorna valori assurdi (>>4.000.000.000...)

# Cliente D

Ok, I crash sono risolti, ma ora la funzione total() ritorna valori assurdi (>>4.000.000.000...)

- Compilatore diverso
- sizeof(long) è diversa!
  - La mia libreria ritorna interi a 32 bit
  - Il nuovo compilatore interpreta int come valore a 64 bit

# Soluzione

## Utilizzo interi di dimensione definita

```
#include <cstdint>


class MyDB {
    MyDBImpl *impl;
public:
    int32_t total() const;
    int32_t &operator[](const char *s);
    MyDB();
    virtual ~MyDB();
};
```

# Cliente E

La mia religione mi impedisce di utilizzare C++ per le mie applicazioni.

Posso chiamare la libreria da C?

I clienti a volte fanno richieste assurde…

# Riguardiamo il codice

```
#include <cstdint>


class MyDB {
    MyDBImpl *impl;
public:
    int32_t total() const;
    int32_t &operator[](const char *s);
    MyDB();
    virtual ~MyDB();
};
```

# Object vs Handle

```
#include <stdint.h>

typedef class MyDBImpl *MyDB_Handle;

int32_t MyDB_total(MyDBHandle h);
void MyDB_destroy(MyDB_Handle h);

MyDB_Handle MyDB_create();

//    Indexer ???
```

# Indexer

Come possiamo rendere l'indexer?

- `int32_t *MyDB_indexer(MyDB_Handle h, const char *s);`
  - Molto simile all'originale
  - Poco "idiomatico" per il C
- `void MyDB_insert(MyDB_Handle h, const char *s, int32_t value);`
  `int32_t MyDB_retrieve(MyDB_Handle h, const char *s);`

# Soluzione

A questo punto, perché avere un'interfaccia C++?

```
#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

typedef class MyDBImpl *MyDB_Handle;

int32_t MyDB_total (MyDBHandle h);
void MyDB_insert(MyDB_Handle h, const char *s, int32_t value);
int32_t MyDB_retrieve(MyDB_Handle h, const char *s);
MyDB_Handle MyDB_create();
void MyDB_destroy(MyDB_Handle h);

…
```

# Cliente A

Ehi, dov’è finita l’interfaccia C++?!?

# Cliente A

Ehi, dov'è finita l'interfaccia C++?!?

Riscriviamola basandola sull'interfaccia C

```
class MyDB {
    MyDB_Handle _handle;
public:
    MyDB() { _handle = MyDB_create(); }
    ~MyDB() { MyDB_destroy(_handle); }

    ??? operator[](const char *s) { return ???; }

    int32_t total() const { return MyDB_total(_handle); }
};
```

# Soluzione – pseudoreference

```
class href {
    friend class MyDB;
    MyDB_Handle _handle;
    const char *_s;
    href(const MyDB_Handle &handle, const char *s) :
        _handle(handle), _s(s) {}
public:
    operator int32_t () const {
        return MyDB_retrieve(_handle, _s);
    }

    href &operator = (int32_t value) {
        MyDB_insert(_handle, _s, value);
        return *this;
    }
};
```

# Soluzione – MyDB

```
class MyDB {
    MyDB_Handle _handle;
public:
    MyDB() { _handle = MyDB_create(); }
    ~MyDB() { MyDB_destroy(_handle); }

    href operator[](const char *s) { return { _handle, s}; }

    int32_t total() const { return MyDB_total(_handle); }
};
```

# Il mio cane

Woof! (Trad: La mia funzione per visualizzare il DB non compila più)

```
void view_db(const MyDB &db) {
    for(auto &x: db) { … }
}
```

```
invalid range expression of type 'const MyDB'; no viable 'begin'
function available
```

A prima vista non stava chiamando nulla.
In realtà chiama i metodi begin() e end(), (e ora fallisce non trovandoli).

# Soluzione 1 – Iteratori (1)

```
typedef MyDBItImpl *MyDB_It_Handle;

MyDB_It_Handle MyDB_begin(MyDB_Handle handle);
MyDB_It_Handle MyDB_end(MyDB_Handle handle);
void MyDB_It_increment(MyDB_Handle handle);
bool MyDB_It_compare(
    MyDB_It_Handle lhs, MyDB_It_Handle rhs);
void MyDB_It_destroy(MyDB_Handle handle);
const char *MyDB_It_name(MyDB_Handle handle);
int MyDB_It_value(MyDB_Handle handle);
```

# Soluzione 1 – Iteratori (2)

```
class MyDB {
    MyDB_Handle _handle;
public:
    MyDB() { … }
    ~MyDB() { … }
    href operator[](const char *s) { … }
    int32_t total() const { … }

    iter begin();
    iter end();
};
```

# Soluzione 1 – Iteratori (3)

```
class iter {
    friend class MyDB;
    MyDB_It_Handle itHandle;
    iter(MyDB_Handle &handle, bool begin) :
        itHandle(begin ? MyDB_begin(handle) : MyDB_end(handle))
        {}
    ~iter() { MyDB_It_destroy(itHandle); }
public:
    operator == (const iter &rhs) const {
        return MyDB_It_compare(itHandle, rhs.itHandle);
    }

    iter &operator ++() { MyDB_It_increment(itHandle); return
*this; }

    iter &operator ++(int) { ??? }
};
```

# Soluzione 1 – Iteratori (4)

```
MyDB_It_Handle MyDB_It_clone(MyDB_It_Handle lhs);

…

iter(const iter &rhs)

    : itHandle(MyDB_It_Clone(rhs.itHandle)) {}
iter &operator = (const iter &rhs) {…}


iter operator ++(int) {
    iter newIter = *this;
    ++newIter;
    return newIter;
}

pair<string, int> operator *() const {
    return { MyDB_It_name(itHandle),
        MyDB_It_value(itHandle) };
}
```

# Soluzione 1 – Iteratori

## Pro

- Mapping 1:1 con gli iteratori C++

## Contro

- Non idiomatici in C
- Non idiomatici in altri linguaggi (es. C#)
- Doppia allocazione (begin, end)
- Copia (con allocazione) su postincremento (`i++`)

# Soluzione 2 - Enumeratori

```
typedef MyDBEnImpl *MyDB_En_Handle;

MyDB_En_Handle MyDB_En_movenext(MyDB_En_Handle h);
void MyDB_En_destroy(MyDB_En_Handle h);

const char *MyDB_En_name(MyDB_En_Handle h);
int MyDB_En_value(MyDB_En_Handle h);
```

# Soluzione 2 - Enumeratori

```
class enumerator {
    MyDB_En_Handle h;
    enumerator() h(nullptr) {}
public:
    bool movenext() {
        h = MyDB_En_movenext(h);
        return h;
    }
    void reset() {
        MyDB_En_destroy(h);
        h = nullptr;
    }
    pair<string, int> operator *() const {…}
};
```

# Soluzione 3 – Callback (1)

## Pro

- Interfaccia più compatta
- Meno allocazioni (1 per enumerazione)
- Idiomatici in altri linguaggi (es. C#)

## Contro

- Non (ancora) idiomatici in C++
- Difficoltà a gestire range parziali

# Soluzione 3 – Callback (2)

```
// C interface
typedef void(MyDB_callback*)(const char *name, int value,
    void* ctx);

void MyDB_Loop(MyDB_En_Handle h, MyDB_callback c, void* ctx);

Interfaccia C++

class MyDB {
…
template<typename T>
void loop(T t) {
  MyDB_loop(itHandle,
    [](const char *name, int value, void* ctx) {
      (*((T*)ctx))(name, value);
    }, (void*)(&t));
}
```

# Soluzione 3 – Callback (3)

## Pro

- Interfaccia super compatta!
- Zero allocazioni!
- Idiomatico in C
- Simile a <algorithm>

## Contro

- Difficoltà a gestire range parziali
- Impossibilità di mantenere riferimenti

# Altre considerazioni

## Calling convention

- Non più molto importante
- (Win32) cdecl, stdcall, fastcall
- (x64) stdcall, vectorcall (SIMD)

## Gestione errori

- Ritornare un enum (errore) e muovere i ritorni su parametry by reference

# Conclusioni

## Interfaccia C

- Si può fare
- Ha meno problemi d’interfacciamento
- Si può interfacciare anche ad altri linguaggi

# Conclusioni