PRAVA

Teoria

SO - Laboratorio

1° Lezione

Composizione opzionale

- Definizione
- Classe `B`
- Classe `A`

2° Lezione

Classe vuota

- Metodi e operatori

Copia superficiale e profonda

- Copia superficiale
- Copia profonda

Costruttori con parametri

- Costruttori vuoti e con parametri
- Conversione implicita di tipo

3° Lezione

Numeri complessi

Ereditarietà

- Istanza di una classe derivata
- Keyword `virtual`
- Esempio classe base e derivata
- Classe puramente virtuale

4° Lezione

Programmazione generica

- Definizione
- Swap overloading
- Swap `void*`
- Swap `template`
- Swap classi virtuali
- Diversi tipi

Classe `Pair`

[1°][] Lezione

Composizione opzionale

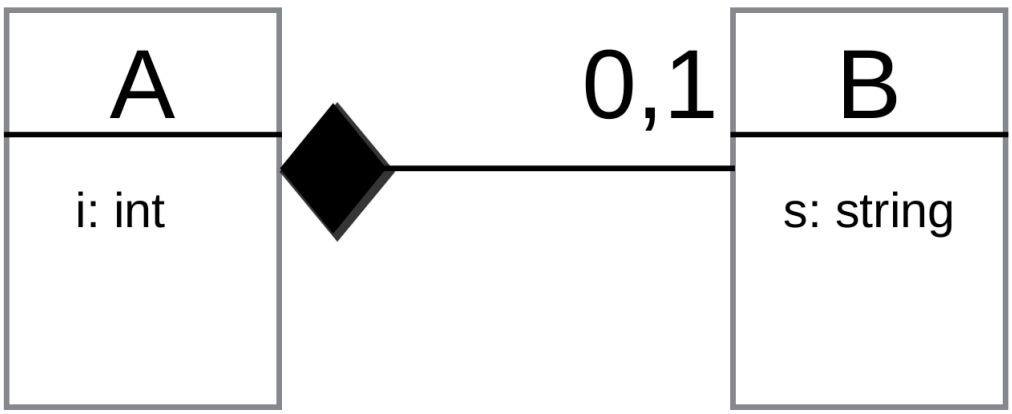
Definizione

- **Composizione**: A has-a B
- **Opzionale**: A could-have-a B (0,1)

L'istanza B **connessa** all'istanza A è **distrutta** quando l'istanza A è distrutta

Soluzione

- **Puntatore a B** in A
- La classe **A gestisce la memoria** in cui si trova l'oggetto B



Classe B

B.h

```
#ifndef __CLASS_B__
#define __CLASS_B__

#include <iostream>
#include <string>
using namespace std;

class B {
    string s;
public:
    B(string _s);
    string get_s();
};

#endif  // __CLASS_B__
```

- I file `.h` sono **inclusi** ma **non compilati**
- Le **guardie** evitano inclusioni multiple

B.cc

```
#include "B.h"

B::B(string _s) {
    s = _s;
}
string B::get_s() {
    return s;
}
```

Classe A

A.h

```
#ifndef __CLASS_A__
#ifndef __CLASS_A__

#include "B.h"
#include <iostream>
using namespace std;

class A {
    int i;
    B *pb;  // puntatore a B, diverso
dall'aggregazione
public:
    A();                        //
costruttore a zero parametri
```

```
    A(int _i, string _s);       //
costruttore a due parametri
    A(const A &_a);             //
costrutttore di copia
    ~A();                       //
distruttore
    A &operator=(const A &_a);  // operatore
di assegnazione
    int get_i();
    string get_s();
    void set_s(string _s);
};

#endif  // __CLASS_A__
```

A.cc

```
#include "A.h"

A::A() {
    cout << "Costruttore a zero parametri" <<
endl;
    i = 0;
    pb = nullptr;
}
A::A(int _i, string _s) {
    cout << "Costruttore a due parametri" <<
endl;
    i = _i;
    pb = new B(_s);
```

```
}
A::A(const A &_a) {
    cout << "Costruttore di copia" << endl;
    i = _a.i;
    if (_a.pb != nullptr) {
        pb = new B(*(_a.pb));  // costruttore
di copia di default
                               // copia
superficiale di *(_a.pb).s
    } else {
        pb = nullptr;
    }
}
A::~A() {
    cout << "Distruttore" << endl;
    if (pb != nullptr) { delete pb; }
}
A &A::operator=(const A &_a) {
    cout << "operator=" << endl;
    if (this->pb == nullptr) {  // oggetto
chiamante non ha B
        if (_a.pb != nullptr) {
            pb = new B((_a.pb)->get_s());
        }
    } else {  // oggetto chiamante ha B
        if (_a.pb != nullptr) {
            *pb = *(_a.pb);  // <--
        } else {
            delete pb;
            pb = nullptr;
        }
```

```
    }
    return *this;  // a = (b = c)
}
int A::get_i() {
    return i;
}
string A::get_s() {
    if (pb == nullptr) { return ""; }
    return pb->s;
}
void set_s(string _s) {
    if (pb != nullptr) {
        *pb = B(_s);
    } else {
        pb = new B(_s);
    }
}
```

[2°][] Lezione

Classe vuota

Metodi e operatori

```
class A {};
int main() {
    A a1;               // A::A();
instanziazione
    A a2(a1);           // A::A(const A&);
superficiale
    a1 = a2;            // A &A::operator=
(const A&);  superficiale
    A *pa = new A();  // instanziazione con
new
    delete pa;          // A::~A();
}
```

Copia superficiale e profonda

Copia superficiale

> **Copia superficiale**: copia degli attributi di un oggetto

Effettuata da **costruttore di copia** e **operatore di assegnazione** di **default**

```
class B {};
class A { public: B *pb; };
```

- In caso di **puntatori ad oggetti** allocati **dinamicamente**
  - Copiano l'**indirizzo** contenuto nel puntatore `pb`
  - Diverse istanze di A hanno `A::pb` che punta alla **stessa istanza di B**

```
// ???
int main() {
    A a;
    a.pb = new B();
    A a2(a);
    // 1 istanza di B ???
    A a3 = a2;
    // 1 istanza di B ???
    A *pa = new A();
    *pa = a;
    delete pa;
}
```

Copia profonda

Soluzione: **ridefinire** costruttore di copia e operatore di assegnazione

- In modo che costruiscano **nuove istanze** degli eventuali puntatori ad oggetti
- Necessario sempre con **allocazione dinamica**

Costruttori con parametri

Costruttori vuoti e con parametri

Quando si definisce un **costruttore con parametri**

- Il **costruttore vuoto `A::A()` non esiste più** nella versione di default
- Occorre **ridefinirlo** per poterlo usare (! errore di compilazione)
- Idea: se è necessario inizializzare un parametro, il costruttore vuoto non è più adatto a tale scopo

Conversione implicita di tipo

Costruttori a **un solo parametro** svolgono il ruolo di **convertitori impliciti di tipo**

- Il **sistema di deduzione dei tipi** del compilatore converte
  - Un **r-value**, di tipo equivalente a quello del costruttore ad un parametro
  - In un'**istanza** della classe richiesta, costruita con il costruttore ad un parametro

```
class A {
    int i;
public:
    A(int _i) { i = _i }
};

void f(const A &_a) {}

int main() {
    A a;
    a = 3;  // A::operator=(A(3));
    f(3);   // f(A::A(3)); f(A(3));
}
```

`explicit`

La **keyword `explicit`** inibisce la conversione implicita di tipo

```
class A {
    int i;
public:
    explicit A(int _i) { i = _i }
};

int main() {
    A a;
    a = 3;  // A::operator=(A(3));
    f(3);   // f(A::A(3)); f(A(3));
}
```

> Preferibile ridefinire gli operatori come metodi
>
> - Risparmia l'utilizzo di getters per gli attributi privati
> - Evita funzioni `friend`

[3°][] Lezione

Numeri complessi

complex.h

```
class Complex {
    double re, im;
public:
    Complex(double _re = 0, double _im = 0);

    // Metodi
    Complex &operator+=(const Complex &_c);
    Complex &operator*=(const Complex &_c);
    Complex operator-() const;  // meno
unario

    // Funzioni esterne
    friend ostream &operator<<(ostream &os,
const Complex &_c);
    friend Complex operator+(const Complex
&_c1,
                             const Complex
&_c2);
    friend Complex operator*(const Complex
&_c1,
```

```
                                const Complex
&_c2);
    friend Complex operator-(const Complex
&_c1,
                                const Complex
&_c2);
};

ostream &operator<<(ostream &os, const
Complex &_c);
Complex operator+(const Complex &_c1, const
Complex &_c2);
Complex operator*(const Complex &_c1, const
Complex &_c2);
Complex operator-(const Complex &_c1, const
Complex &_c2);

void test_Complex();
```

complex.cc

```
Complex::Complex(double _re, double _im) {
    re = _re;
    im = _im;
}

// Metodi
Complex &Complex::operator+=(const Complex
&_c) {
    re += _c.re;
```

```
    im += _c.im;
    return *this;
}
Complex &Complex::operator*=(const Complex
&_c) {
    (*this) = (*this) * _c;  // implementare
usando operator*
    return *this;            // return
(*this) * _c; non funziona
}
Complex Complex::operator-() const {
    return Complex(-re, -im);
}

// Funzioni esterne
Complex operator-(const Complex &_c1, const
Complex &_c2) {
    return _c1 + (-_c2);
}
Complex operator+(const Complex &_c1, const
Complex &_c2) {
    return Complex(_c1.re + _c2.re, _c1.im +
_c2.im);
}
Complex operator*(const Complex &_c1, const
Complex &_c2) {
    return Complex(_c1.re * _c2.re - _c1.im *
_c2.im,
                   _c1.re * _c2.im + _c2.re *
_c1.im);
}
```

```
ostream &operator<<(ostream &os, const
Complex &_c) {
    os << _c.re << " + " << _c.im << "i";
    return os;
}
```

Ereditarietà

- **Tipi**: `public`, `private`, `protected`
- Ha consequenza sulla **visibilità**

Visibilita' classe base B / Ereditarieta' classe derivata D	public	private	protected
public	Tutti	B	B\D
private	Tutti	_	D e suc
protected	D	_	D
	D e suc	_	D e suc

Visibilita' classe base / Ereditarieta' classe derivata	public	private	protected
public			
public	Public	Inaccessibile	Protected
private	Private	Inaccessibile	Private
protected	Protected	Inaccessibile	Protected

Istanza di una classe derivata

- Una **classe derivata `public`** può essere acceduta
  - Come istanza della classe derivata
  - Come istanza della classe base
    - Implementa la relazione **IS-A**
- I **puntatori alla classe base** possono puntare ad un **istanza** della classe **derivata**
- Un'istanza di una classe derivata può essere assegnata ad un'istanza di una classe base ma non il contrario

Keyword `virtual`

I **metodi** possono essere definiti **`virtual`**

- È il modo in cui in C++ si dichiara il **late binding**
  - **Early binding**: risolto dal compilatore
  - **Late binding**: risolto run-time
- Si usa late binding solo se si accede tramite **puntatore o reference**

Esempio classe base e derivata

`a.h` (classe base)

```
class A {
    int i;
public:
    A();
    explicit A(int _i);
    A(const A &_a);
    virtual ~A();  // necessario se si accede
alle istanze
                   // derivate con puntatori
alla classe base
    int get_i() const;
```

```
};

void test_A();
int quadrato(const A &_a);
int raddoppia(A _a);
```

a.cc (classe base)

```
A::A() {
    i = 0;
    cout << "A::A()" << this << endl;
}
A::A(int _i) {
    i = _i;
    cout << "A::A(int)" << this << endl;
}
A::A(const A &_a) {
    i = _a.i;
    cout << "A::A(const A&)" << this << endl;
}
A::~A() {
    cout << "A::~A()" << this << endl;
}
int A::get_i() const {
    return i;
}

void test_A() {
    A a;
    cout << a.get_i();
```

```
    A a3(3);
    cout << a3.get_i();
    cout << quadrato(a3);
    cout << raddoppia(a3);
}
int quadrato(const A &_a) {
    return _a.get_i() * _a.get_i();
}
int raddoppia(A _a) {
    return 2 * _a.get_i();
}
```

b.h (classe derivata)

```
class B : public A {
    string s;
public:
    B();
    B(int _i, string _s);
    ~B();  // non necessario
    string get_s();
};

void test_B();
```

b.cc (classe derivata)

```
B::B() {
    s = "";
    cout << "B::B()" << this << endl;
```

```
}
B::B(int _i, string _s) : A(_i) {
    s = _s;
    cout << "B::B(int,string)" << this <<
endl;
}
B::~B() {
    cout << "B::~B()" << this << endl;
}
string B::get_s() {
    return s;
}

void test_B() {
    B b;
    cout << b.get_s() << b.get_i();
    B b2(3, "Moli");
    cout << b2.get_s() << b2.get_i();
    cout << quadrato(b2);  // b è anche A
    cout << raddoppia(b2);
}
```

main.cc

```
int main() {
    test_A();
    cout << endl;
    test_B();
    cout << "main:" << endl;
    B *pb = new B(7, "note");
```

```
    cout << quadrato(*pb);
    delete pb;
    cout << "puntatore ad A:" << endl;
    A *pa = new A(11);
    cout << quadrato(*pa);
    delete pa;
    cout << "puntatore ad A che punta ad un
B:" << endl;
    pa = new B(13, "giorni");
    cout << quadrato(*pa);
    delete pa;  // importante che sia
definito virtual A::~A()

    A a;
    B b;
    a = b;      // b = a; non compila
                // possibile ridefinendo
l'operatore =
    cout << "assegnamento ad a" << endl;
    a = A(3);  // a = 3; non compila
(explicit)
    A a2(b);   // costruttore di copia di A

    system("pause");
    return 0;
}
```

Classe puramente virtuale

> **Metodo puramente virtuale**: metodo non implementato

> **Classe puramente virtuale**: classe con almeno un metodo puramente virtuale

- Una classe puramente virtuale **non ha istanze** (~ interfaccia Java)
- La sua dichiarazione nella classe è seguita da **`= 0;`**

```
class A {
    ...
    virtual int method() = 0;
    ...
};
```

I costruttori possono essere virtuali, i distruttori no

[4°][pdf-4] Lezione

Programmazione generica

Definizione

> **Programmazione generica**: possibilità data da un linguaggio di rappresentare tipi e implementare algoritmi che abbiano un tipo come parametro

Il **tipo parametrico** viene specificato a tempo di

- **Compilazione** (e.g. templates)
- **Esecuzione** (e.g. classi con virtualizzazione)

Scopo: implementare algoritmi indipendenti dal tipo su cui operano

- Idealmente al massimo livello di astrazione possibile

Swap overloading

Necessita la scrittura di **una funzione per ogni tipo** desiderato

```
void my_swap (int &f, int &s) {
    int tmp = f; f = s; s = tmp;
}
void my_swap (string &f, string &s) {
    string tmp = f; f = s; s = tmp;
}
```

Swap `void*`

> ## C-style casting

Puntatore `void`: puntatore che punta a variabili di **qualsiasi tipo**

- Necessitano di **casting** a puntatore al tipo specificato

```
#include <utility>

void my_swap(void *&f, void *&s) {
    void *tmp = f;
    f = s;
    s = tmp;
}
```

```
int main() {
    void *a;
    void *b;
    a = new string("hello");
    b = new string("world");
    cout << *((string *) a) << *((string *)
b) << endl;
    swap(a, b);
    cout << *((string *) a) << *((string *)
b) << endl;

    void *x;
    void *y;
    x = new int(33);
    y = new int(44);
    cout << *((int *) x) << *((int *) y) <<
endl;
    my_swap(x, y);
    cout << *((int *) x) << *((int *) y) <<
endl;

    cout << "a = " << *((int *) a) << endl;
    // no compile time error, no runtime
error
    // output a = 1919907594
}
```

static_casting

```
#include <utility>

void my_swap(void *&f, void *&s) {
    void *tmp = f;
    f = s;
    s = tmp;
}

int main() {
    void *a;
    void *b;
    a = new string("hello");
    b = new string("world");
    cout << *(static_cast<string *>(a));
    cout << *(static_cast<string *>(b)) <<
endl;
    swap(a, b);
    cout << *(static_cast<string *>(a));
    cout << *(static_cast<string *>(b)) <<
endl;

    void *x;
    void *y;
    x = new int(33);
    y = new int(44);
    cout << *(static_cast<int *>(x));
    cout << *(static_cast<int *>(y)) << endl;
    my_swap(x, y);
```

```
    cout << *(static_cast<int *>(x));
    cout << *(static_cast<int *>(y)) << endl;

    cout << "a = " << *(static_cast<int *>
(a)) << endl;
    // no compile time error, no runtime
error
    // output a = 1919907594
}
```

Swap `template`

> **Tipo parametrico**: tipo generico, istanziato o definito in altre parti del codice

Il codice compilato conterrà una **copia della funzione** per ciascuno dei tipi richiesti

```
#include <utility>

template<class T>
void my_swap(T &f, T &s) {
    T tmp = f;
    f = s;
    s = tmp;
}
```

```
int main() {
    int a = 3;
    int b = 4;
    cout << "before a = " << a << " b = " << b << endl;
    my_swap<int>(a, b);
    cout << "after a = " << a << " b = " << b << endl;

    string s1 = "hello";
    string s2 = "world";
    cout << "before s1 = " << s1 << " s2 = " << s2 << endl;
    my_swap<string>(s1, s2);

    cout << "after s1 = " << s1 << " s2 = " << s2 << endl;
    return 0;
}
```

Swap classi virtuali

Generalizzazione tramite **virtualità**, eseguita a **runtime**

> **A.h**

```
class A {
public:
    virtual A &operator=(const A &_a) = 0;
    virtual A *clone() const = 0;
    virtual ~A() {};
};
```

B.h

```
#include "A.h"
class B : public A {
    int i;
public:
    B(int _i) { i = _i; };
    B &operator=(const A &_b);
    B &operator=(const B &_b);
    B *clone() const;
    friend ostream &operator<<(ostream &os,
const B &_b);
};
ostream &operator<<(ostream &os, const B
&_b);
```

B.cc

```
#include "B.h"
void my_swap(A &f, A &s) {
    A *temp = f.clone();
    f = s;
    s = *temp;
```

```
    delete temp;
}
int main() {
    B x(33);
    B y(44);
    cout << x << y << endl;
    my_swap(x, y);
    cout << x << y << endl;
}
```

- La funzione **`clone` virtuale** effettua la copia profonda di **qualsiasi tipo**

Diversi tipi

```
template<typename T>
T min(T a, T b) {
    return a < b ? a : b;
}
template<typename T1, typename T2>
T1 min(T1 a, T2 b) {
    return a < b ? a : b;
}
int main() {
    cout << min<int>(3, 5) << endl;
    cout << min<int, double>(6, 5.5) << endl;
 // 5 o 5.5 ????
}
```

Classe `Pair`

```
template<typename F, typename S>
class Pair {
public:
    Pair(const F &f, const S &s);
    F get_first() const;
    S get_second() const;
private:
    F first;
    S second;
};

template<typename F, typename S>
Pair<F, S>::Pair(const F &f, const S &s) {
    first = f;
    second = s;
}
template<typename F, typename S>
F Pair<F, S>::get_first() const {
    return first;
}
template<typename F, typename S>
S Pair<F, S>::get_second() const {
    return second;
}

int main() {
```

```
    Pair<int, string> p(5, "Pippo");
}
```