

# Composizione di oggetti

Programmazione di Sistema  
A.A. 2017-18

Programmazione di Sistema



# Argomenti

- Oggetti composti
- Duplicazione degli oggetti
- Copia e movimento

# Oggetti composti

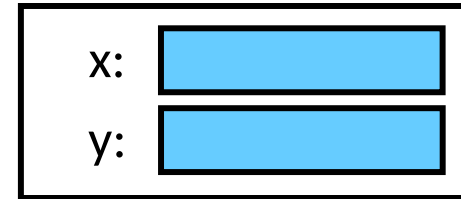
- Un oggetto può contenere altri oggetti
  - Come parte integrante della propria struttura dati
  - Oppure facendo riferimento ad essi tramite puntatori
- In entrambi i casi, è compito del costruttore inizializzare opportunamente tali oggetti
  - La sintassi è differente
  - La disposizione in memoria anche!

# Oggetti composti

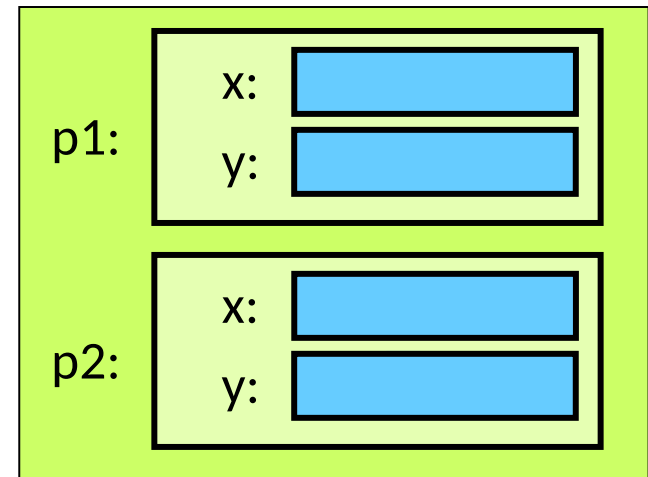
```
class CPoint {  
public:  
    CPoint(int x, int y);  
private:  
    int x, y;  
};
```

```
class CRect {  
public:  
    CRect(int x1, int y1,  
          int x2, int y2);  
private:  
    CPoint p1,p2;  
};
```

geom1.h



CPoint



CRect

# Oggetti composti

```
#include "geom1.h"
```

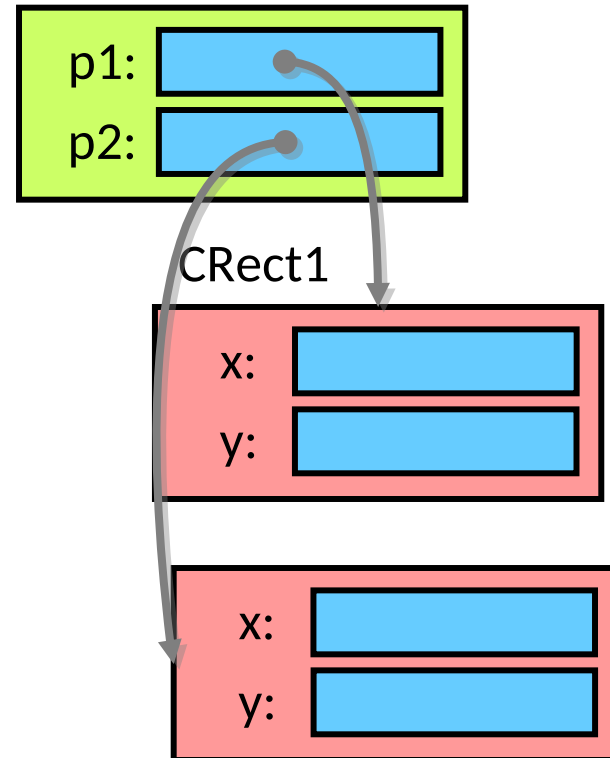
```
CPoint::CPoint(int x,  
               int y) {  
    this->x = x;  
    this->y = y;  
}
```

```
CRect::CRect(int x1, int y1,  
             int x2, int y2) :  
    p1(x1,y1),  
    p2(x2,y2) {  
}
```

# Oggetti composti

```
class CRect1 {  
public:  
    CRect1(int x1,  
           int y1,  
           int x2,  
           int y2);  
    ~CRect1();  
private:  
    CPoint *p1;  
    CPoint *p2;  
};
```

geom2.h



# Oggetti composti

```
CRect1::CRect1(int x1, int y1,  
               int x2, int y2) {  
    p1= new CPoint(x1,y1);  
    p2= new CPoint(x2,y2);  
}
```

```
CRect1::~~CRect1(){  
    delete p1; //p1=NULL;  
    delete p2; //p2=NULL;  
}
```

# Copia ed assegnazione

- Talora occorre copiare il contenuto di un oggetto in un altro oggetto
  - Esplicitamente, quando si esegue un'assegnazione
  - Implicitamente, quando si passa un oggetto per valore o si ritorna un oggetto
- Se l'oggetto destinazione esisteva prima della copia si parla di assegnazione
- Se, invece, viene creato al momento della copia, si parla di costruzione di copia



# Copia e assegnazione

```
class CPoint {  
    int x,y;  
public:  
    CPoint(int x,int y);  
};  
  
CPoint p1(10,5)  
CPoint p2(3,3);  
  
f(p1,p2); //costruzione  
           //di copia
```

```
void f(CPoint A,  
       CPoint B) {  
    A = B; //assegnazione  
  
    CPoint C(A);  
    //costruzione  
    //di copia  
    CPoint D = B;  
    //costruzione di  
    //di copia!  
}
```

# Copia e assegnazione

- Di base, il compilatore ricopia il contenuto di tutte le variabili istanza dell'oggetto sorgente nell'oggetto destinazione
  - Se si tratta di puntatori, questo comportamento può essere problematico

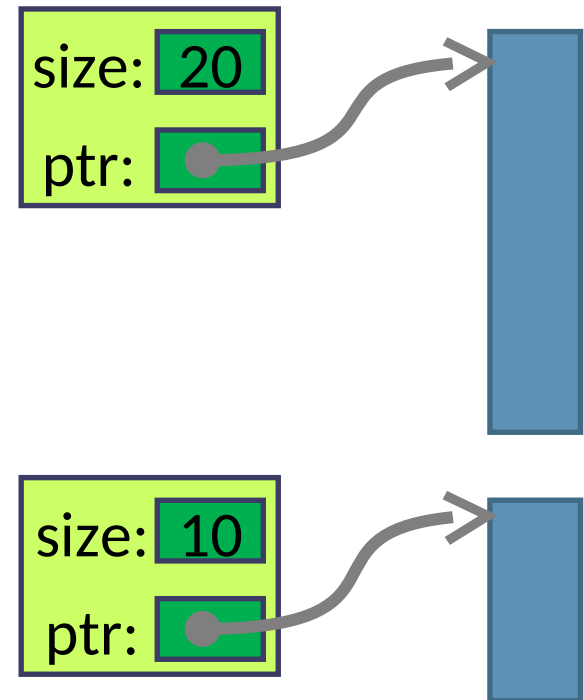
# Esempio

```
class CBuffer {  
    int size;  
    char* ptr;  
public:  
    CBuffer(int size);  
    ~CBuffer();  
};  
  
CBuffer b1(20), b2(10);  
  
b1 = b2;
```

```
CBuffer::  
CBuffer(int size) {  
    this->size=size;  
    ptr=new char[size];  
}  
  
Cbuffer::  
~CBuffer() {  
    delete[] ptr;  
}
```

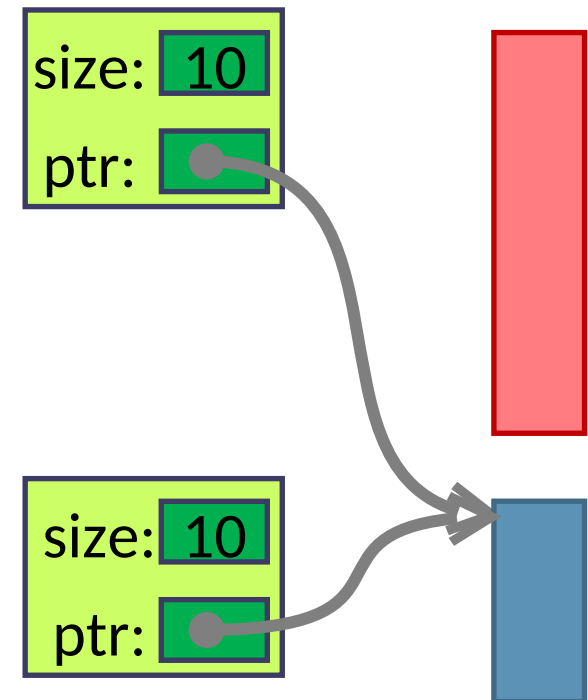
# Esempio

```
class CBuffer {  
    int size;  
    char* ptr;  
public:  
    CBuffer(int size);  
    ~CBuffer();  
};  
  
CBuffer b1(20), b2(10);  
  
b1 = b2;
```



# Esempio

```
class CBuffer {  
    int size;  
    char* ptr;  
public:  
    CBuffer(int size);  
    ~CBuffer();  
};  
  
CBuffer b1(20), b2(10);  
  
b1 = b2;
```



# Copiare i puntatori

- Ogni volta che si deve copiare un puntatore sorge un'ambiguità
  - Basta duplicare il puntatore facendo riferimento allo stesso indirizzo?
  - Oppure occorre duplicare il blocco puntato?
  - Se l'oggetto destinazione esisteva già, che fine fa il vecchio puntatore?
  - Il compilatore, di base, sceglie la prima strategia
  - Il programmatore può modificarla, definendo le proprie procedure

# Costruttore di copia

- Indica al compilatore come comportarsi quando deve inizializzare un nuovo oggetto a partire da un originale

```
class CBuffer {  
    CBuffer(const CBuffer& source);  
};
```

# Costruttore di copia

```
class CBuffer {  
    int size;  
    char* ptr;  
public:  
    CBuffer(const CBuffer& source) {  
        this->size=source.size;  
        this->ptr=new char[size];  
        memcpy(this->ptr, source.ptr, size);  
    }  
};
```



# Operatore di assegnazione

- Indica al compilatore come comportarsi quando si assegna un nuovo valore ad un oggetto già esistente

- Che ha risorse da liberare

```
CBuffer& operator=(const CBuffer&  
source );
```

# Operatore di assegnazione

- Distrugge il contenuto dell'oggetto destinazione
  - Prima di inizializzarlo con la copia dell'oggetto sorgente
- Restituisce un riferimento all'oggetto destinazione

# Operatore di assegnazione

```
CBuffer& operator=(const CBuffer&  
                    source) {  
    if (this!= &source) {  
        delete[] this->ptr;  
        this->ptr=null;  
        this->size=source.size;  
        this->ptr=new char[size];  
        memcpy(this->ptr, source.ptr, size);  
    }  
    return *this;  
}
```

# Operatore di assegnazione

```
CBuffer& operator=(const CBuffer&  
                    source) {  
    if (this!= &source) {  
        delete[] m_data;  
        this  
        this  
        this  
        memc  
    }  
    return  
}
```

Se questa riga viene omessa...

...e si assegna un  
oggetto a se stesso, i dati contenuti  
vengono distrutti

# Operatore di assegnazione

```
CBuffer& operator=(const CBuffer&  
                    source) {  
    if (this!= &source) {  
        delete[] this->ptr;  
        this->ptr=null;  
        this->size=source.size();  
        this->ptr=new char[source.size()];  
        memcpy(this->ptr, source.ptr, source.size()*sizeof(char));  
    }  
    return *this;  
}
```

...in caso di  
eccezione, il  
distruttore  
rilascerebbe due  
volte la memoria

# Operatore di assegnazione

Questa è la fonte dei guai...  
...Se l'oggetto è grosso, e non c'è la memoria  
richiesta, lancia un'eccezione

```
this->ptr=new char[size];
```

```
memcpy(this->ptr, source.ptr, size);
```

```
return *this;
```

# Equivalenza semantica

- Costruttore di copia ed operatore di assegnazione devono essere semanticamente equivalenti

```
CBuffer b1(5);  
Cbuffer b2(7); //b2 viene creato  
b2=b1;      //e poi sovrascritto  
CBuffer b3(b1); //b3 e b2 devono avere  
                //lo stesso contenuto
```



# Generazione automatica

- Di base, il compilatore inserisce automaticamente sia il costruttore di copia che l'operatore di assegnazione
  - Copiando o assegnando in modo ricorsivo tutti i membri della classe
- Si può impedire la duplicazione di un oggetto dichiarando privati sia il costruttore di copia che l'operatore di assegnazione
  - Se ne impedisce l'utilizzo da parte di codice esterno



# Generazione automatica

```
class CBuffer {  
    //...  
private:  
    CBuffer(const CBuffer&);  
    CBuffer& operator=(const CBuffer&);  
};
```

```
CBuffer b1(10), b2(5);  
b1=b2; //Errore di compilazione  
CBuffer b3(b1); //idem
```

# Assegnazione e risorse

- L'operatore di assegnazione deve rilasciare tutte le risorse possedute
  - Per non creare leakage
  - Si sovrappone al distruttore
  - Occorre fare attenzione in caso di manutenzione!

# La regola dei tre

- Se una classe dispone di una qualunque di queste funzioni membro, occorre implementare le altre due
  - Costruttore di copia
  - Operatore di assegnazione
  - Distruttore
- In mancanza di ciò, il compilatore fornirà la propria implementazione
  - La quale, per lo più, non potrà essere corretta

# Copia e movimento

- Copiare un oggetto in un altro può essere un'operazione dispendiosa
  - In alcuni casi, lo si vuole evitare esplicitamente
  - Garantendo il controllo centralizzato delle risorse contenute nell'oggetto

# Movimento

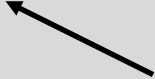
- Accanto all'operazione di copia, la specifica C++ 2011 introduce il concetto di movimento
  - «Svuotare» un oggetto che sta per essere distrutto del suo contenuto e «travasarlo» in un altro oggetto
- Candidati al movimento
  - Le variabili locali al termine del blocco in cui sono state definite
  - I risultati delle espressioni temporanee
  - Gli oggetti anonimi costruiti a partire dal tipo
    - ▮ `std::string("ciao");`
  - Tutto ciò che non ha un nome e può comparire solo a destra di "=" nelle assegnazioni (RVALUE)

# Costruttore di movimento

- Nelle classi in cui è utile, si può definire il costruttore di movimento
  - `TypeName(TypeName&& source);`
  - L'operatore `&&` indica un candidato al movimento del tipo che lo precede
  - RValue reference
- Il suo compito è «smontare» l'originale e trasferire il suo contenuto nell'oggetto destinazione
  - L'originale verrà modificato, ma poiché sta per essere distrutto, questo non ha importanza
  - A patto che i "resti" dell'oggetto smontato possano essere distrutti senza creare danni

# Costruttore di movimento

```
class CBuffer {  
    int size;  
    char* ptr;  
public:  
    CBuffer(CBuffer&& source) {  
        this->size=source.size;  
        this->ptr=source.ptr;  
        source.ptr=NULL;  
    }  
};
```



Risparmia l'operazione, onerosa, di allocare un nuovo blocco e inizializzarlo con il contenuto della stringa di partenza.

**MODIFICA L'ORIGINALE!**

# Costruttore di movimento

- A differenza di quello di copia, il costruttore di movimento non è generato automaticamente dal compilatore
  - È compito del programmatore scegliere se e dove implementarlo
- Ogni volta in cui occorre copiare un valore, il compilatore valuta se effettuare una copia o un movimento del dato
  - A patto che sia disponibile il costruttore di movimento relativo



# Semantica del movimento

- Se si applica il movimento, il compilatore genera le seguenti pseudo-chiamate
  - `Obj_MoveConstructor(dst,src);`
  - `Obj_Destructor(src);`
- Occorre fare in modo che la chiamata al distruttore non elimini le risorse spostate

# Utilità del movimento

- Riduce, quando possibile, il costo del passaggio per valore
  - Sia nei parametri che nei dati che vengono restituiti da una funzione

```
string f() {  
    string x("..."); //Dichiara una stringa  
    string a(x); //Non si può muovere,  
                //x: LVALUE, si COPIA x in a  
    string b(a+x); // Il risultato di a+x  
                //viene mosso in b  
    string c(funzioneCheRitornaString());  
                //il risultato è mosso in c  
    return c; //c viene mosso nel risultato  
}
```

# Operatore di movimento

- In alcuni casi (ad es., `std::unique_ptr`) la copia non è possibile
  - Il movimento, invece, sì
  - Rende possibile creare una funzione che ritorna un oggetto di tipo `unique_ptr`, senza violarne le regole
  - Se un oggetto contiene solo valori elementari o tipi valore, il movimento equivale alla copia

# Operatore di movimento

- Se i tipi valore contenuti, rappresentano una risorsa esterna, il vantaggio può essere grande
  - Descrittori di file, socket, connessioni a base dati, ...
  - L'alternativa sarebbe acquisire una seconda copia della risorsa e rilasciare la risorsa originale subito dopo

# Usi nella libreria standard

- Le classi della libreria standard sono state riviste per supportare la semantica del movimento
  - Aumentandone di molto l'efficienza e rendendole utili in applicazioni a basso livello

# Assegnazione per movimento

- Analogamente a quanto avviene per l'operatore di assegnazione semplice...
  - ...occorre dapprima liberare le risorse esistenti e poi trasferire il contenuto dell'oggetto sorgente

```
CBuffer& operator=(CBuffer&& source) {  
    if (this != &source) {  
        delete[] this->ptr;  
        this->size=source.size;  
        this->ptr=source.ptr;  
        source.ptr=NULL;  
    }  
    return *this;  
}
```

# Il paradigma Copy&Swap (1)

- Il meccanismo di assegnazione (sia normale che per movimento) è, in generale, fonte di problemi
  - Unisce le operazioni di distruzione e di copia
  - In caso di eccezioni, può generare mostri

```
class intArray {  
    std::size_t mSize;  
    int* mArray;  
public:  
    intArray(std::size_t size=0): //costruttore  
        mSize(size), mArray(mSize? new int[mSize]: NULL) {}  
  
    intArray(const intArray& that): //costruttore di copia  
        mSize(that.mSize), mArray(mSize? new int[mSize]:NULL)  
    {  
        std::copy(that.mArray, that.mArray+mSize, mArray);  
    }  
    ~intArray() { delete [] mArray; } //distruttore
```

**//segue...**



# Il paradigma Copy&Swap

## (2)

- È facile scrivere implementazioni errate dell'operatore di assegnazione
  - Specialmente rispetto a possibili eccezioni
- Se si omette (1)
  - Si copierebbe un oggetto su se stesso, distruggendo l'array sorgente su cui si andrebbe poi a leggere
- Se si omette (2)
  - In caso di eccezione successiva, il distruttore potrebbe rilasciare due volte la memoria
- (3) è fonte di guai
  - Se l'oggetto è grosso, potrebbe non esserci la memoria richiesta
  - new[] lancia un'eccezione e l'esecuzione si interrompe lasciando un oggetto corrotto

```
intArray& operator=(  
    const intArray& that) {  
  
    if (this!=&that) { //(1)  
        delete mArray;  
        mArray=NULL; //(2)  
        mSize=that.mSize;  
        mArray = new int[mSize]; //  
(3)        std::copy(that.mArray,  
                    that.mArray+mSize,  
                    mArray);  
    }  
    return *this;  
}
```

- Occorre riscrivere la funzione in modo da evitare il rischio che le eccezioni possono introdurre



# Il paradigma Copy&Swap

## (3)

- Si introduce la funzione swap(...)
  - Definendola come friend della classe
  - Si occupa di scambiare il contenuto delle risorse tra due istanze
- Si appoggia sulla funzione std::swap
  - Per scambiare i riferimenti elementari contenuti sugli oggetti
- Si definisce l'operatore di assegnazione con un parametro di tipo valore
  - La logica dell'operatore di copia provvederà a fare un duplicato nel parametro that
  - In caso di eccezione, \*this non viene modificato (l'eccezione si verifica nel tentativo di creare la copia, prima di effettuare uno swap)
  - Si evita il test di identità tra sorgente e destinazione
- Swap permette anche di implementare il costruttore di movimento
  - Con le stesse garanzie

```
friend void swap(intArray& a,
                intArray& b) {

    std::swap(a.mSize, b.mSize);
    std::swap(a.mArray, b.mArray);
}

intArray& operator=(intArray
that){

    //that è passato per valore,
    //copiato o mosso a seconda del
    //contesto in cui è usato

    swap(*this, that);
    return *this;
}

//costruttore di movimento
intArray(intArray&& that):
    mSize(0), mArray(NULL) {
    swap(*this, that);
}
```

# La funzione `std::move(...)`

- Funzione di supporto per trasformare un oggetto generico in un riferimento RValue, così da poterlo utilizzare nella costruzione o assegnazione per movimento
  - Definita nel file `<utility>`
  - Forza l'oggetto passato come parametro ad essere considerato come un riferimento RValue rendendolo disponibile per usi ulteriori (viene eseguito uno `static_cast<T&&>(t)` )
  - Aiuta a rendere esplicita la conversione, quando si vuole forzare l'uso del movimento rispetto alla copia

```
std::string str("hello");  
std::vector<std::string> v;  
  
v.push_back(str); // v = ["hello"], str = "hello"  
  
v.push_back(std::move(str)); // v = ["hello", "hello"], str = ""
```

# Spunti di riflessione

- Si implementi la classe MyString, che incapsula un'array di caratteri allocato dinamicamente
- Si creino i costruttori di copia e movimento e i corrispondenti operatori di assegnazione