Programmazione ad oggetti in C++

Michelangelo Diligenti

Ingegneria Informatica e dell'Informazione

diligmic@dii.unisi.it

Argomenti

- Programmazione ad oggetti con sua implementazione in C++
  - Classi e loro implementazione in C++
  - Constness in metodi e dati
  - Costruttori e distruttori
  - Allocazione dinamica di memoria
  - Namespaces
  - Direttive static ed inline

Perché il C++

- Vantaggi
  - Popolarità e diffusione
  - Efficienza: adeguato per sistemi ad alte prestazioni
  - Espressività
  - Estende il C (che conoscete)
- Svantaggi
  - Non architettato per essere facile o pulito
    - Non agevole la gestione della memoria (come in Java per problemi semplici)
  - Curva di apprendimento lenta per capire certi dettagli
  - Meno librerie standard di Java

bool: tipo per variabili booleane

- In C++ esiste un tipo specifico per variabili booleane
  - bool
  - Assume solo valori true e false
  - Esempio

```
bool CheckValue(int i) {
  bool valid = true;
  if (i < 10 && i > 3)
    valid = false;
  return valid;
}
```

C++: dichiarazione variabili

- ANSI C: dichiarazione di variabili ad inizio funzione
- C++: dichiarazione dove la variabile serve!
  - Maggiore chiarezza e pulizia

Codice C

```
int Funzione() {
  int i,j;
  for (i=0; i <10; ++i) {
    j=i+1; …
  }
}
```

Codice C++

```
int Funzione() {
  for (int i=0; i <10; ++i) {
    int j=i+1;
    …
  }
}
```

Classi in C++: metodi const

- Vediamo come arricchire l'interfaccia con metodi const
- Un metodo const è un metodo che non modifica lo stato della classe
  - Non modifica i dati
  - Non chiama altri metodi che non sono const
  - Compilatore controlla che nulla venga cambiato
- Definito mettendo la specifica “const” dopo gli argomenti di un metodo (fuori dalla parentesi)
- Metodo const può essere chiamato su oggetti const

Classi in C++: metodi const

- Esempio:

```
class Date {
 public:
  Date();  // constructor
  Date(int month_, int day_, int year_);
  void SetYear(int year_);
  int GetYear( ) const;     // returns Year
  int GetMonth( ) const;    // returns Month
  int GetDay( ) const;      // returns Day
 private:
  int year;
  int month;
  int day;
};
```

Questo metodo cambia lo stato della classe, non è const

Questi metodi ritornano una Copia del campo Year,Month,day Ma non cambiano i valori interni

Classi in C++: metodi const

- Esempio:

```
class Date {  ...
  int GetYear( ) const;
  void SetYear(int year_ );
 ...
};

int main() {
const Date date(1,1,2011);
cout << date.GetYear();
date.SetYear(2011);
return 0;
}
```

Questo metodo cambia lo stato della classe, non è const

date è const non deve cambiare

OK, metodo const è chiamabile su oggetto const.

NO, metodo non const non chiamabile

Classi in C++: metodi const

- Esercizio:

```
class Date { ...
  int GetYear( ) const {
    return Year;
  }
  int* GetYear( ) const {
    return &Year;
  }
  const int* GetYear( ) const {
    return &Year;
  }
  const int& GetYear( ) const {
    return Year;
  }
};
```

1) Quali di questi metodi sono corretti? Per quali il compilatore darà errore?

2) Quali di questi metodi sono chiamabili su un "const Date date"?

Classi: interfaccia ed suo uso

- Abbiamo visto come creare interfaccie
- Vediamo ora
  - Come implementarle
  - Come usarle
- Usiamo la classe Date come esempio
  - Implementazione dei metodi inline
  - Implementazione dei metodi normale

Classi: implementazione inline

- Implementazione dei metodi **inline**

```
class Date {  // inizio definizione classe
  int GetYear( ) const {
    return Year;  // implementazione direttamente dentro la definizione
  }
};
```

- Se il metodo è implementato inline, il compilatore cercherà di copiare il suo contenuto nel chiamante
  - Non si chiama effettivamente la funzione
    - Ma il risultato è equivalente
    - Operazione trasparente per l'utente

Classi: implementazione inline

- Consigliabile prependere implementazioni inline con la direttiva inline, perché sia chiaro

```
class Date {
  inline int GetYear( ) const {...}
};
```

- Vantaggio inline è la potenziale maggiore velocità
  - Evita context switching quando si chiama funzione
- Svantaggi inline
  - Interfacce sporche (con implementazione insieme)
  - Binari più grandi con problemi di caching => lentezza
    - Inline può avere risultato opposto a quello per cui si usa
    - Inlining è suggerimento per il compilatore, che non è obbligato a fare l'inlining se non lo ritiene opportuno

Classi: implementazione classica

- Implementato fuori dalla definizione della classe
- Modo standard che non sporca la definizione e chiama i metodi in modo standard

```
class Date {  // inizio definizione classe
  ...
  int GetYear( ) const;
};  // fine definizione
```

Nome classe

Nome metodo implementato

```
int Date::GetYear() const {
  return Year;  // implementazione fuori dalla la definizione
}
```

Interfaccia ed implementazione

- Come in C, spesso si separa definizione ed implementazione
- Definizione di classe con nome “classe”
  - In file classe.h
  - Protetto da ifndef per evitare inclusioni multiple

```
#ifndef NOME_FILE_H
#define NOME_FILE_H
…  // il codice del .h
#endif
```

- Implementazioni della classe
  - In file classe.cc

Interfaccia ed implementazione

Date.h

```
class Date {
 public:
  Date();  // constructor
  Date(int month_, int day_, int year_);
  void SetYear(int year_);
  int GetYear() const;
  int GetMonth() const;
  int GetDay() const;
 private:
  int year;
  int month;
  int day;
};
```

Date.cc

```
#include “Date.h”
int Date::GetYear() const {
  return year;
}
int Date::GetMonth() const {
  return month;
}
int Date::GetDay() const {
  return day;
}
void Date::SetYear(int year_) {
  year = year_;
}
```

Uso della classe nei clienti

- Qualunque client che voglia usare la classe deve
  - Includere il file .h dove la classe viene definita
  - Compilare il file .cc relativo alla classe ottenendo un file oggetto .o
    - In gcc con il comando

      g++ -c -o file.o file.cc
  - Aggiungere il file .o a quelli in input al linker e creare l'eseguibile
    - In gcc con il comando

      g++ -o nome_binario file1.o file2.o …

Uso della classe nei clienti

- In un file.cc per creare un istanza di una classe inclusa da un .h si chiama un costruttore
  - NomeClasse nomeIstanza(argomenti_costruttore);
  - Esempi:

    Date date;

    Date date(8,11,2011);

Uso della classe nei clienti

- Esempio

```
#include “date.h”
int main() {
  Date date(8,11,2011);
  cout << date.GetDay()<<”/”<<date.GetMonth() << “/”<<date.GetYear();
}
```

- Naturalmente solo i metodi pubblici sono in generale visibili e chiamabili
  - Per gli altri il compilatore ritorna un errore

Overloading di metodi

- **Overloading**: più metodi con stesso nome e diversi argomenti (fatto anche per costruttori)
  - Esempio

```
class Date {…
  void Add(int days);
  void Add(int days, int months);
  void Add(int days, int months, int years);
};
```

  - Compilatore decide quale chiamare in base ai parametri

```
Date d;

d.Add(10);  // chiama il primo

d.Add(10, 2);  // chiama il secondo
```

Namespaces

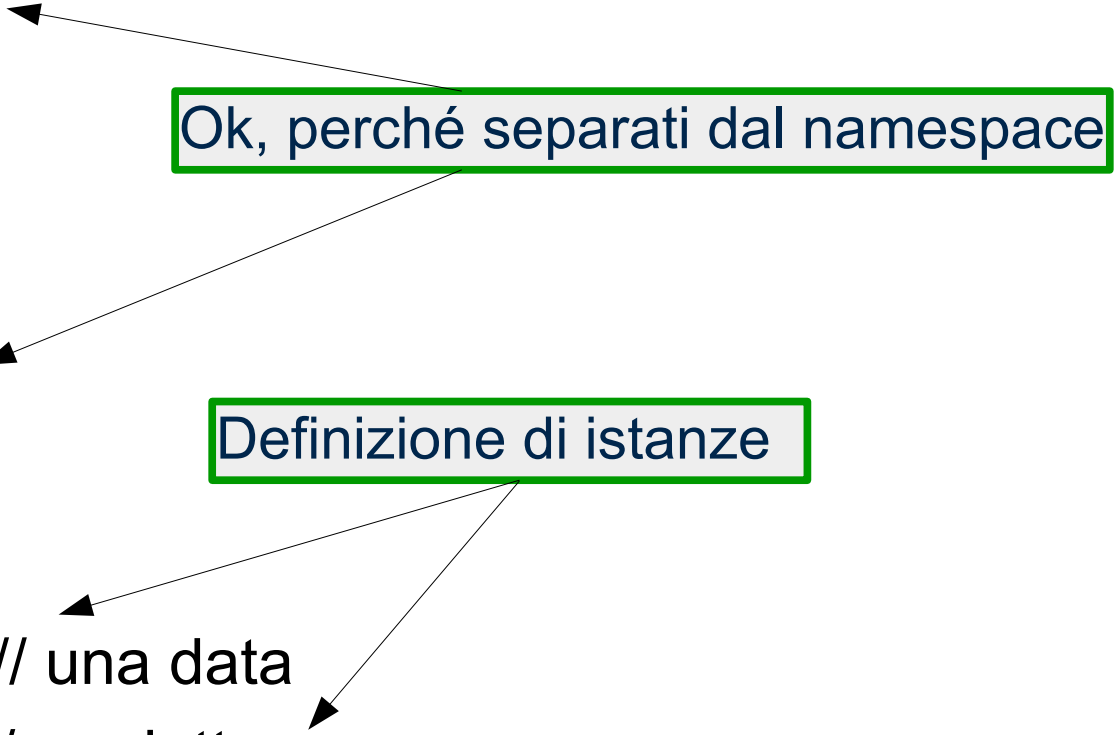
```
namespace nome_namespace {
  // codice
};
```

- Creano uno spazio separato di nomi
  - Tipo definito dentro un namespace non collide con un tipo con lo stesso nome definito in altro namespace
  - In programmazione ad oggetti vi sono tanti tipi quanti oggetti da modellare!
    - Utili nella gestione di progetti grossi
- Da fuori il namespace, si riferisce ad un tipo nel namespace, preponendo nome_namespace::

Namespaces

- Esempio:

```
namespace Time {
  class Date { … };
}

namespace Fruit {
  class Date { ... };
}

Time::Date tdate;  // una data
Fruit::Date fdate;  // un dattero
```



Namespaces: using

- Keyword **using** permette di dichiarare che un namespace verra' usato
  - Una volta dichiarato l'utilizzo non e' necessario specificarlo per ogni dichiarazione di istanza
  - Esempio

```
using namespace Time;
Date d;  // invece di Time::Date d;
```

  - La direttiva using e' valida nel blocco in cui viene fatta.

Namespaces: using

- **using** permette di dichiarare l'uso di una specifica classe o funzione:

  using Time::Date;

- Direttiva è valida nel contesto in cui viene fatta
  - Spesso si mette le direttive using all'inizio di un file.cc cosi che siano visibili nell'intero file
  - Oppure aggiunte in una singola funzione o metodo per alleggerire la sintassi
  - Cercare di non usare la direttiva using in un file .h
    - Chi include il file riceve la direttiva spesso senza saperlo
      - Facile ottenere errori inaspettati difficili da capire

Namespaces

- Namespace sono aperti
  - Possibile aggiungere codice al namespace in qualsiasi file
- Utilizzo tipico
  - Mettere tutto il codice sia del .h che del .cc dentro il namespace

Date.h

```
namespace Time {
  class Date { … };
}  // fine namespace
```

Date.cc

```
namespace Time {
  Date::Date() { … }
}  // fine namespace
```

Unamed Namespaces

- **Unnamed namespaces** o namespace senza nome
  - Limitano la visibilità di parti di codice
  - Simile a come veniva usato la direttiva static in C

```
namespace {
  void Function(int a, int b);
  ...
}
```

Namespace senza nome

Le funzioni, classi ecc. definite nel namespace sono visibili e riferibili solo nel file attuale

  - **Vantaggio**: Si può scegliere nomi di funzioni/classi semplici ed intuitivi
    - Non si mette simboli nel namespace globale

Global Namespaces

- Namespace globale (global) è namespace di default
- Tipi o funzioni dichiarati fuori da un qualsiasi namespace sono nel namespace globale
- Se create classi per delle librerie, cercate di non aggiungerle al global namespace
  - Aggiunta inopportuna al global namespace viene detta: inquinamento del namespace globale (***polluting global namespace***)
- ECCEZIONE: i main vanno sempre sul global namespace

Classi di comune utilità

- In C++ molte librerie standard sono messe nel namespace std (indica Standard)
  - Fondamentale imparare ad usarle
  - Tutte implementate ad oggetti
  - Uno dei principali vantaggi del C++ rispetto al C
  - Come primo esempio di classi di libreria, studiamo adesso le classi std::iostream e std::string

iostreams

- Permettono di leggere o scrivere su/da file, tastiera/schermo, stringhe, ecc.
- VANTAGGI
  - **Correttezza**: tipate, evitano i problemi di stdio.h nel caso di type mismatch, esempio in C:

```
#include <stdio.h>
double i = 4.0;
printf(”%d”, i);     // stampo double come intero => spazzatura
std::cout << i;      // nessuna possibilità di errori
```

  - Gestiscono internamente i tipi, senza possibili errori da parte dell'utente
  - **Estendibilità**: possibile estenderle per gestire qualsiasi tipo/classe

iostreams: standard output

- Per stampare i tipi fondamentali (int, long, float, double, ecc.), il comando ha forma

cout << variabile|stringa << … << variabile|stringa; // standard output

cerr << variabile|stringa << … << variabile|stringa; // standard error

  - Esempio:

```
#include<iostream>

int x = 0; float y = 2.0f; char* str = “pippo”;

std::cout << x << “::” << y << “ “ << str << std::endl;
```

endl = “\n”
cioè newline, va a capo

Streams: standard input

- Per leggere i tipi fondamentali da tastiera possibile usare std::cin
  - Vantaggio: codice per la lettura non deve essere cambiato se cambiano i tipi!
  - Esempio:

```
#include<iostream>
using namespace std;
int main() {
  int x = 0; float y; char* str = “pippo”;
  cin >> x >> y >> str;
}
```

Stringhe

- Realizzate con la classe std::string
  - Permetteno la gestione di stringhe di caratteri
  - Grosso miglioramento rispetto a char* in C
    - **Non usare mai char*** per le stringhe in C++ a meno che non siate costretti dall'uso di **legacy code**
      - **Legacy code**: vecchie librerie ereditate ma fuori dal vostro controllo
  - VANTAGGI
    - niente bugs perché si usa l'interfaccia e si nasconde l'implentazione (puntatori, allocazionedi memoria, ecc)
      - grazie all'**astrazione** permessa dall'OOP
    - interfaccia chiara e pulita
    - Lunghezza delle strighe viene gestita automaticamente
  - RISULTATO: maggiore velocità di sviluppo

Classe string

- Esempio 1

```
#include <string>
#include <iostream>
int main() {
  std::string str(“pippo”);  std::string str1(“pluto”);
  str += “pluto”;  // concatenazione
  std::cout << str << “ lun:” << str.length() << “ terzo char:” << str[2] << std::endl;
  if (str.find(“pippo”) != std::string::npos)
     cout << “Contiene pippo in pos:” << str.find(“pippo”) << std::endl;
  std::cout << “Sub:” << str.substr(0, 4) << std::endl;
  const char* c = str.c_str();
}
```

string[n] ritorna carattere n-esimo
Sembra strano, [] non e' un metodo.
Non e' delega speciale per le classi di libreria!
Vedremo come farlo anche nelle vostre classi.

Lo stesso discorso fatto per il [], vale per +=, vedremo come farlo nelle vostre classi

ricerca di sottostringhe

std::string::npos è valore speciale definito dentro la classe come una costante (in genere =-1) che definisce una non-posizione (posizione non valida)

sottostringa estratta da indice iniziale per lunghezza sottostringa

Trasforma string in stringa c (char*)

Classe string

- Esempio 2 (simile al precedente ma con using)

```
#include <string>
#include <iostream>
using namespace std;
int main() {
  string str = “pippo”;
  string str1 = “pluto”;
  str += str1;
  if (str == str1)          { cout << “Stringhe uguali”; }
  if (str != str1)      { cout << “Stringhe diverse”; }
  cout << str << “ lun:” << str.length() << “ terzo char:” << str[2] << endl;
  if (str.find(“pippo”) != string::npos)
        cout << “Contiene pippo in pos:” << str.find(“pippo”) << endl;
  cout << “Sub:” << str.substr(0, 4) << endl;
}
```

string1 == string2 o string1 != string2 permettono di controllare se le stringhe sono uguali o differiscono

Classe string: metodi principali

- int length(): ritorna la lunghezza della stringa
- int find(const string): cerca stringa dentro la stringa
- char string[const int]: carattere i-esimo
- string substr(const int s, const int l): ritorna sottostringa che parte dall's-esimo carattere e lunga l caratteri
- const char* c_str(): ritorna stringa come char* del C
- s = s1; riassegna la stringa s1 a stringa s
- cout << s; stampa la stringa sullo stream. Preferirlo per chiarezza al pur corretto printf(“%s”, s.c_str());

Classe string: metodi principali

- bool s1 == s2; operatore che ritorna se le stringhe sono uguali
- bool s1 != s2; operatore che ritorna true se le stringhe sono diverse
- string s = s1 + s2; operator+ concatena due stringhe
- s += s1; operator+= concatena s1 su s (in place)

Streams: scrivere files

- **ofstream** per scrivere
- Per usarli #include <fstream>
  - Uso identico a scrittura/lettura su schemo/tastiera

```
#include<fstream>
using namespace std;
int main() {
  ofstream of(”./pippo.txt”);
  if (!of.good()) {  // controlla tutto OK (file trovato ed apribile, ecc.)
    cerr << ”Non posso aprire il file\n”;
    return 1;
  }
  int x = 0; float y; char* str = “pippo”;
  of << x << ” ”<< y << ”::” << str;
  return 0;
}
```

Streams: leggere files

- **ifstream** per leggere da file
- Per usarli #include <fstream>
  - Uso identico a scrittura/lettura su schemo/tastiera

```
#include<fstream>
using namespace std;
int main() {
  ifstream is(”./pippo.txt”);
  if (!is.good()) {  // controllo tutto OK (file trovato ed apribile, ecc.)
    cerr << ”Non posso aprire il file\n”;
    return 1;
  }
  int x = 0; float y; string str = “pippo”;
  is >> x >> y >> str;
  return 0;
}
```

Streams: leggere files

- Possibile leggere iterativamente da uno stream
  - Un iostream ritorna un valore Booleano per controllare se la lettura ha avuto successo

```
…  // come sulla slide precedente
int main() {
  ifstream is(”./pippo.txt”);
  if (!is.good()) {  … }  // come sulla slide precedente
  string str;
  while (is >> str) {    // va avanti fino al termine del file
  ...
  }
  return 0;
}
```

Streams: iostream altre funzioni

- Metodo close chiude uno stream aperto

```
ifstream is(“./pippo.txt”);
is >> i >> j;
is.close();  // da ora in poi non posso leggere dallo stream
```

- Metodo open riapre uno stream gia' costruito

```
ifstream is(“./pippo.txt”);  is >> i >> j;  is.close();
is.open(“./pippo.txt”);  // riapro lo stream
is >> i; // posso leggere di nuovo dallo stream
```

- Metodo seekg(posizione) porta stream alla posizione

```
ifstream is(“./pippo.txt”);
is >> i >> j;
is.seekg(0);  // riavvolgo lo stram all'inizio (posizione 0)
```

Classe string: esercizio

- Scrivere e testare il codice per analizzare un file di testo in input e determinare se la parola “computer” è contenuta nel testo
  - Suggerimento: usare la classe ifstream e string

Streams: leggere files

- Esempio di uso in una classe

```
class FindInFile {
private:
  std::ifstream is;
public:
  FindInFile(const std::string& file) {
    is.open(file.c_str());
    if (!is.good()) { … };
  }
  bool Trova(const std::string& word) {
    is.seekg(0);
    std::string str;
    while (is >> str) {
      if (str == word) return true;
    }
    return false;
  }
};
```

La direttiva static

- Un dato o un metodo possono essere **static**
- Dato static è un dato descrittivo della classe (o meglio dell'entità), non della singola istanza
- Metodo static modella una funzionalità comune a tutte le istanze della classe
  - Non la si applica ad un'istanza specifica
  - Simile ad usare la classe come namespace

Esempio Dato static

- Per tracciare l'anno della Date più recente creata

```
class Date {
 public:
  static int latestYear;
  Date(int year_) { SetYear(year_); }
  void SetYear (int year_) {
    year = year_;
     if (year > latestYear) latestYear = year; }
  int year;
};
```

- Utilizzo

```
Date d1(2001); Date d2(2098); Date d3(1998);

cout << Date::latestYear;      // Date:: perché è proprietà della classe nel
                               // suo insieme non di un'istanza. Accedibile
                               // solo se public (vedremo come evitarlo)!
```

Definizioni di costanti

- In C, le definizioni sono realizzate tramite il preprocessore

```
#DEFINE MAXINT 10000000
```

- In C++, si utilizza la clausola const

```
const int MAXINT = 10000000;
```

  - Vantaggio è nel fatto che la costante ha un tipo
    - Compilatore può realizzare controlli di correttezza

Definizioni di costanti in classi

- In C, le definizioni sono realizzate tramite il preprocessore e sono globali
  - Nel global namespace, non bello in progetti grossi!
- In C++, si unisce le direttive static e const per realizzare costanti relative ad una classe

```
class Date {…
  static const int DEFAULT_YEAR;
  Date() { Year = DEFAULT_YEAR; … };
};
const int Date::DEFAULT_YEAR = 2000;
```

  - VANTAGGI
    - **Modularità**: non si corrompe il global namespace
    - **Chiarezza**: Le costanti sono nell'interfaccia
    - **Correttezza**: le costanti hanno un tipo

Definizioni di costanti

- In C, le definizioni sono realizzate tramite il preprocessore

```
#DEFINE MAXINT 10000000
```

- In C++, si utilizza la clausola const

```
const int MAXINT = 10000000;
```

  - Vantaggio è nel fatto che la costante ha un tipo
    - Compilatore può realizzare controlli di correttezza

Metodi static

- Metodo static è un metodo relativo all'intera classe e non ad un'istanza specifica
  - Le funzionalita' restano nel contesto della classe a cui si riferiscono
  - Hanno accesso ai dati privati
- Quando utilizzarli
  - Se un metodo lavora solo su dati static (**getters statici**)
  - Nel caso di funzioni in cui entrano in gioco più istanze di una classe
    - così restano nel contesto della classe
    - visibili nell'interfaccia

Metodi static: esempio1

- Static Getter per accedere a dato static
  - Vantaggio: dato static resta privato e modificabile solo da metodi della classe

```
class Date {
private:
  static int LatestYear;
public:
  static int LatestYear() { return latestYear; }
  ...
};
```

- Utilizzo

```
Date d1(2001); Date d2(2098); Date d3(1998);

cout << Date::LatestYear();  // Simile ad usare la classe come namespace!
```

Metodi static: esempio2

- Realizzare funzione che compara due date e decide quale precede l'altra

```
class Date { …
  static int Precede(const Date& a, const Date& b) {
     if (a.GetYear() < b.GetYear()) return -1;
     else if (a.GetYear() > b.GetYear()) return 1;
     else if (a.GetMonth() < b.GetMonth()) return -1;
     else if (a.GetMonth() > b.GetMonth()) return 1;
     else if (a.GetDay() < b.GetDay()) return -1;
     else if (a.GetDay() > b.GetDay()) return 1;
     return 0;
  }
};
```

Utilizzo di Metodi static

- Metodo static non si applica ad un istanza ma si chiama Preponendo NomeClasse::
  - Esempio

```
class Date { …
  static int Precede(const Date& a, const Date& b) {
    … // definito come in slilde precedente
  }
};

// Per chiamarlo
Date a(1,1,2011);
Date b(8,9,2009);
cout << Date::Precede(a, b);
```

Metodi static esercizio

- Realizzare i metodi Sum e Product per la classe NumeroComplesso usando metodi statici

- Usare i dati/metodi statici per contare il numero di istanze della classe che sono state create (Date o ComplexNumber)
  - Usare un dato static per il contatore
  - Aggiungere dei metodi static che permettano di leggere il contatore ed azzerarlo

Typedef

- Typedef permette di definire il nome di un tipo in modo compatto
- VANTAGGI
  - In una classe permette di definire il tipo in un solo posto centralizzato
  - si deve modificare una sola linea per cambiare tipo
  - Evita che si debba digitare nomi di tipi lunghi ed aumenta la leggibilità
    - Lo capirete quando vedremo i templates
- Sintassi

  typedef tipo nuovo_nome;

Typedef: esempio

- Classe per scambiare due interi:

```
class Swap {
public:
  static void Swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
  }
};
```

Per passare da int ad un altro tipo come float o string, devo cambiare 3 linee

- Con typedef

```
class Swap {
public:
  typedef float T;
  static void Swap(T* a, T* b) {
    T tmp = *a;
    *a = *b; *b = tmp;
  }
};
```

Adesso cambio tipo con un cambiamento su una sola riga

Allocazione di memoria in C++

- Usa direttiva **new**
- Per allocare un singolo elemento sull'heap
  - Chiama il default constructor
    - Tipo* nomeIstanza = new Tipo;
  - Chiama il costruttore selezionato
    - Tipo* nomeIstanza = new Tipo(argomenti_costruttore);
- O per allocare un vettore di N elementi
  - Tipo* nomeIstanza = new Tipo[N];
  - Il default constructor viene chiamato per ogni elemento
  - Non possibile chiamare un altro costruttore

Allocazione di memoria in C++

- New contro malloc/calloc
  - Date* vec = new Date[N]; // C++
  - Date* vec = (Date*)malloc(N*sizeof(Date)); // C
- new chiama il costruttore per ogni elemento creato
- malloc non lo chiama, non supporta OOP
  - lascia gli elementi in uno stato non deterministico
  - malloc non conosce i tipi. Ritorna puntatore generico (void*)
  - richiede che utente faccia il cast
  - Richiede che utente specifichi quantità di memoria
  - Frequente causa di bugs

Deallocazione di memoria in C++

- Usa comando **delete**
- Per singoli elementi
  - Date* date = new Date;
  - delete date;
- Per vettori di istanze
  - Date* vec = new Date[N];
  - delete[] vec;
- delete contro malloc/calloc
  - **delete date;** in C++ uguale a **free(date);** in C
  - Delete invoca distruttore per ogni istanza cancellata. Il distruttore non viene mai chiamato direttamente!

Deallocazione per new di un singolo elemento

Deallocazione per vettori di elementi

Costruttore di default

- Costruttore di default invocato se non specificato alternativamente
  - Compilatore lo implementa per noi se non si sono definiti altri costruttori
  - Spesso il costruttore di default non fa quello che si vuole
- Quando definire il proprio default constructor?
  - Si vuole un comportamento diverso dal default
  - Esempio se la data di default deve essere 1/1/2000

```
Date() {
  Year = 2000; Month = 1; Day = 1;
};
```

Costruttori di default

- Default constructor ha il nome della classe e nessun parametro
  - Chiamato di default se non si danno specifiche
  - Esempio queste sono equivalenti

    Date date; ← Default costructor chiamato se non si specifica nulla

    Date date = Date();

Costruttore di copia (copy)

- Copy constructor
  - Prende in input un'instanza della stessa classe
  - Responsabile di copiare l'istanza
- Esempio

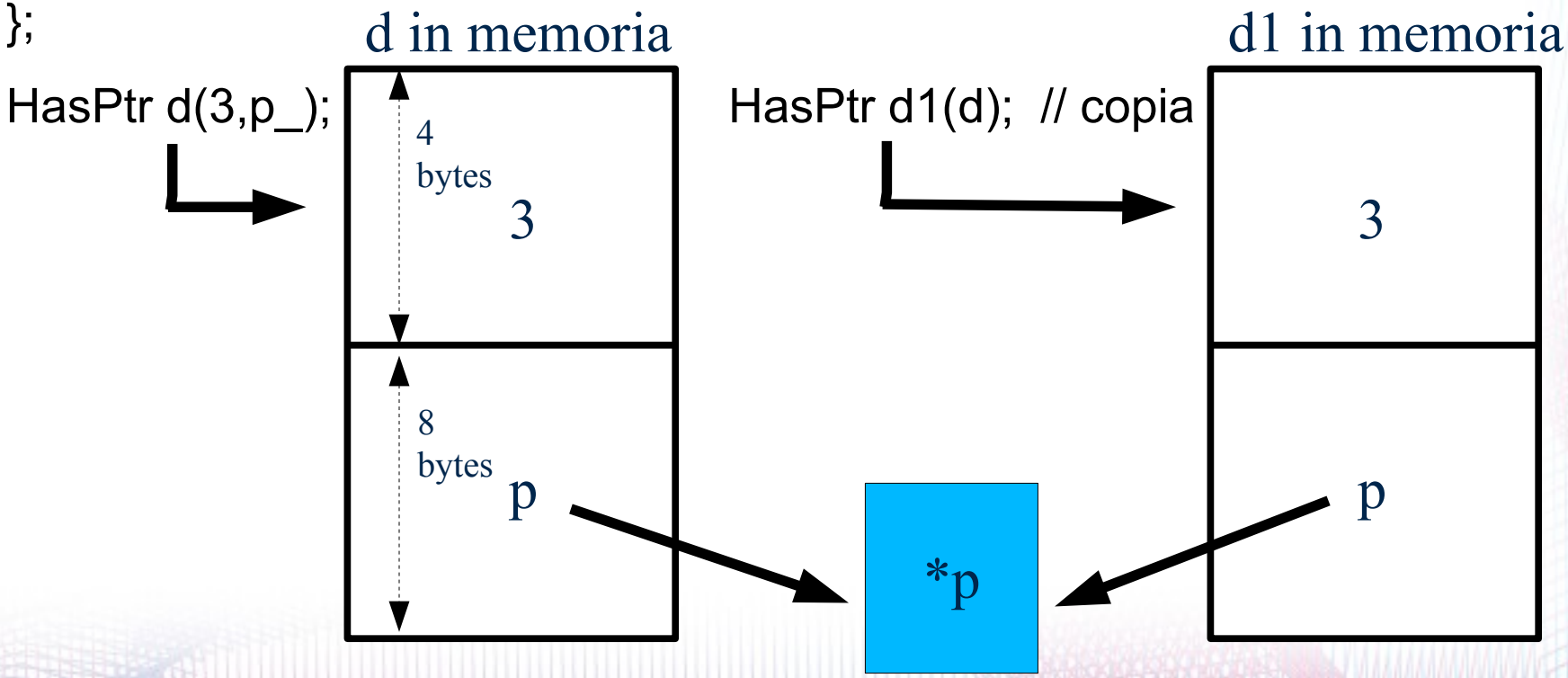
```
class Date {
  ...
  Date();  // default constructor
  Date(const Date& date);  // copy constructor
};
```

- Se non definito, compilatore ne definisce uno
  - Copy constructor fa byte copy dei dati della classe

Copy constructor e Byte copy

```
class HasPtr {
    int v;
    int* p;
    HasPtr(int v_, int psize) { v=v_; p = new int [psize]; }
};
```

Copy constructor di default copia byte a byte:
i due puntatori puntano alla stessa cella.
Non ho veramente duplicato le istanze!



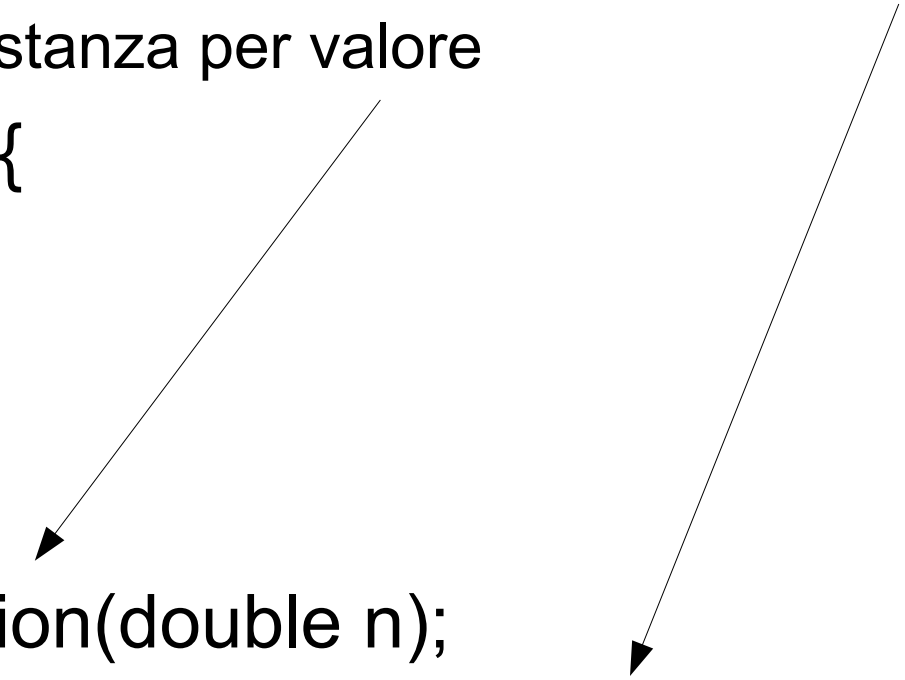
Costruttore di Copia

- Quando definire il proprio copy constructor?
  - Quando il byte copy non è opportuno
  - Se la classe contiene puntatori ad altri dati
  - Non si deve copiare solo i puntatori ma gli oggetti puntati (**deep copy**)
  - Il byte copy copia solo i puntatori (**shallow copy**)
- Se non vi sono puntatori, in genere non serve definire il proprio copy constructor

Costruttore di Copia

- Costruttore di copia chiamato implicitamente se
  - Passi istanza come argomento per valore
  - Ritorni istanza per valore

```
class Numero {
  double* num;
  ...
};
Numero Function(double n);
                void Function(Numero n);
```



Esempio di classe: costruttori

```
#include <string.h>  // per memset
#include <stdlib.h>  // per malloc
class Vector {
 private:
  typedef int Element;
  Element* data;
  int N;  // dimensione del vettore
 public:
  Vector(int N_) {
    N = N_;
    data = (Element*)malloc(N*sizeof(Element));
    memset(data, 0, N*sizeof(Element));
  }
  Vector(const Vector& vec) {
    N= vec.N;
    data = (Element*)malloc(N*sizeof(Element));
    memcpy(data, vec.data, N*sizeof(Element));
  }
  inline Element Get(int i) const { return data[i]; }
  inline void Set(int i, const Element& el) { data[i] = el; }
};
```

dati del vettore sono in puntatore

Costruttore:
crea vettore di dimensione N, con tutti valori pari a 0

Copy constructor
va definito altrimenti di vec viene copiato il puntatore, ed i due vettori punterebbero allo stesso *data*

Costruttori convertitori

- Costruttore di classe C che prende un solo parametro di tipo T è detto convertitore
  - Viene usato per convertire in modo automatico da T a C, in assegnazioni o chiamate di funzione
  - Esempio

```
class C {
public:
  C(int c) { …}
};
void function(C c) {…}
int main() {
  function(3);
  C c = 5;
}
```

Costruttore convertitore da int a C

Funzione vuole C come argomento

OK function vuole C ma il costruttore convertitore viene chiamato automaticamente per convertire 3 in C. Equivalente a function(C(3));

Costruttore convertitore chiamato in modo automatico. Equivalente a C c(5);

Costruttori convertitori

- Costruttori convertitori sono criptici e poco intuitivi
  - Tra poco vediamo come disabilitarli
- Ma non sono solo sempre inutili e da evitare
  - Esempio di costruttore convertitore nelle librerie standard

```
class string {
  string(const char* c);  // converte char* in std::string
};
void Function(std:string s);
```

  - Permette di chimare Function in modo intuitivo:

```
Function(“pippo”);
```

  - “pippo” e' char*, convertito in std::string dal costruttore

Explicit e Costruttori

- **explicit** marca un costruttore che non deve essere usato in modo automatico dal compilatore
  - Disabilita i costruttori convertitori
  - Oppure disabilita il copy constructor che copia un dato passato per valore
  - Buona norma marcare tutti i costruttori convertitori explicit
    - Molti pensano che sarebbe dovuto essere default nello standard
  - Usarlo sempre per copy constructor di oggetti grandi
    - Evita che il passaggio per valore possa avvenire

Explicit e Costruttori

- Esempio di uso di explicit

```
class C {
public:
  explicit C(int c) { … }
  explicit C(const C& c) { … }
};

void function(C c) {…}
void function1(const C& c) {…}

int main() {
  function(3);
  C c(3);
  function(c);
  function1(c);
  return 1;
}
```

Costruttori sono esplicit

No, il costruttore convertitore esplicito non è chiamabile automaticamente

OK, la chiamata è esplicita

No, il copy construtor è solo esplicito

OK, qui si passa per reference e non si chiama il copy constructor

Esempio: costruttori explicit

```
#include <string.h>  // per memset
#include <stdlib.h>  // per malloc
class Vector {
 private:
  typedef int Element;
  Element* data;
  int N;  // dimensione del vettore
 public:
  explicit Vector(int N_) {
    N = N_;
    data = new Element[N];  // in C (Element*)malloc(N*sizeof(Element));
    memset(data, 0, N*sizeof(Element));
  }
  explicit Vector(const Vector& vec) {
    N= vec.N;
    data = new Element[N];  // in C (Element*)malloc(N*sizeof(Element));
    memcpy(data, vec.data, N*sizeof(Element));
  }
  inline Element Get(int i) const { return data[i]; }
  inline void Set(int i, const Element& el) { data[i] = el; }
};
```

Esempio di classe: costruttori

- Copy costructor explicit evita costose copie implicite
  - Garanzia di efficienza del codice generato
  - Fondamentale in team di molti programmatori

```
class Vector {
  ...
 explicit Vector(const Vector& vec) {
   N= vec.N;
   data = new Element[N];  // in C (Element*)malloc(N*sizeof(Element));
   memcpy(data, vec.data, N*sizeof(Element));
 }
...
};

void Function(const Vector v) {}
void Function1(const Vector& v) {}
Vector v(10000000);
Function(v);  // Errore in compilazione, Vector non e' copiabile implicitamente
Function1(v);  // OK, v non e' copiata ma passata per riferemento
```

Distruttori

- Definisce cosa fare quando si distrugge un'istanza
  - Contrario del costruttore
  - Metodo speciale senza parametri indicato come

    ~NomeClasse() {}
  - Esempio

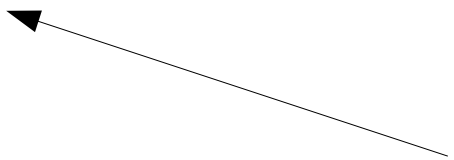
```
class Date {
  ...
  ~Date() { cout << “Nel distruttore\n”; }
};
```

  - **ATTENZIONE**: non si chiama mai in modo esplicito

Distruttori

- Il distruttore viene chiamato automaticamente
  - Quando una variabile va “out-of-scope”

```
for (int i = 0; i < 100; ++i) {
  Date d(i%30 + 1, i%12 + 1, 2011);
  cout << d.GetDay() << “ “ << d.GetMonth() << “\n”;
}
```



Fine blocco a cui d appartiene.
L'istanza d viene ad ogni iterazione distrutta.
Viene chiamato il distruttore automaticamente.

Distruttori

- Spesso non serve definire un distruttore
  - Il compilatore ne definisce uno di default
- Va definito il distruttore se
  - Chiudere evenutali connessioni a database o a server remoti
  - Chiudere file aperti, nel caso degli ostream del C++ viiene fatto automaticamente
  - **Distruttore va definito se il costruttore aveva allocato memoria dinamica**
    - Distruttore deve deallocarla
    - Altrimenti si hanno dei leaks
    - Se serviva copy constructor => serve distruttore

Esempio: costruttori e distruttori

```
#include <string.h>  // per memset
#include <iostream>
#include <stdlib.h>
class Vector {
 private:
  typedef int Element; Element* data;
  int N;
 public:
  explicit Vector(int N_) {
    N = N_; data = new Element[N];
    memset(data, 0, N*sizeof(Element));
  }
  ~Vector() { delete[] data; }
  explicit Vector(const Vector& vec) {
    std::cout << "Nel copy constructor\n";
    N= vec.N; data = new Element[N];
    memcpy(data, vec.data, N*sizeof(Element));
  }
  inline Element Get(int i) const { return data[i]; }
  inline void Set(int i, const Element& el) { data[i] = el; }
};
```

dati del vettore sono in puntatore

Distruttore va definito per deallocare quello che era stato allocato dinamicamente

Copy constructor va definito altrimenti di vec viene copiato il puntatore, ed i due vettori punterebbero allo stesso *data*

- Aggiungete main che chiama il copy constructor

Constructors,distruttore e new/delete: esempio

```
class Vector {
  …
    Element* data;
  public:
    explicit Vector(int N_) {
      N = N_;
      data = new Element[N];
    }
    ~Vector() {
      delete[] data;
    }
    explicit Vector(const Vector& vec) {
      N = vec.N;
      data = new Element[N];
      memcpy(data, vec.data, N*sizeof(Element));
    }
  ...
};
```

dati del vettore sono in puntatore per allocare dinamicamente la memoria

serve distruttore per liberare data quando il vettore viene distrutto. Altrimenti si ha memory leak e la memoria viene persa fino alla fine del processo

va definito altrimenti vec viene copiato il puntatore, i due vettori punterebbero allo stesso *data*

Che succederebbe in dellocazione se non avessi il copy constructor?

Puntatore this

- **this** è una keyword speciale in C++
  - Dentro una classe, è puntatore all'istanza attuale
    - Non usabile dentro metodi static
  - Esempio, metodo che decide se una data cade in Aprile

```
class Date { …
  bool IsApril() const {
     return (this->Month == 4);
  }
};
```

  - Keyword this spesso solo opzionale ma aiuta a rendere il codice chiaro
    - vedremo casi in cui invece è necessaria

Argomenti di Default

- Possibile fornire valori di default per parametri di una funzione
  - Si mette il valore direttamente nella definizione
  - Se un valore di default è fornito anche tutti gli altri argomenti sulla destra devono avere valori di default
  - Comodi perché evitano la definizione di più funzioni
  - Pericolosi perché il codice è meno intuitivo che quando tutti i parametri sono visibili nella chiamata
  - Chiamante spesso perde traccia di cosa succede

Argomenti di Default

- Esempio

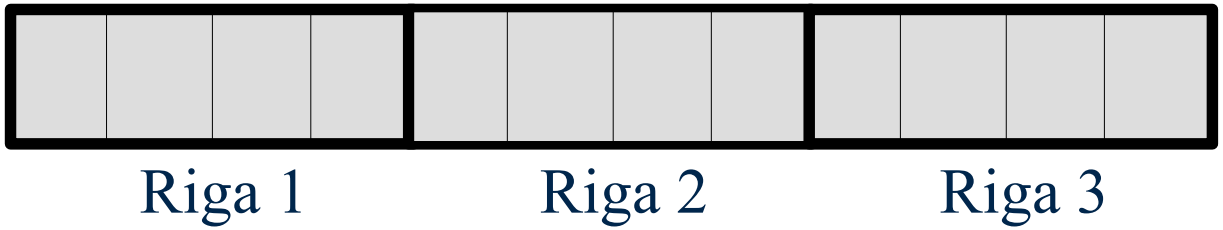
```
class Date {…
  Date(int day, int month=1, int year=2011) {
    Day=day; Month=month; Year = year;
  }
};


Date d(1);  // OK, usa default month=1 e year = 2011
Date d(1,2);  // OK, usa default year = 2011
Date d(1,2,2000);  // OK, defaults non sono usati
Date d(,2,2000);  // NO! Errore del compilatore se un parametro è omesso
                  // anche tutti quelli alla sua destra devono essere omessi
```

Esercizio

- Implementare una classe libreria per la gestione di matrici (classe Matrix) che
  - Usa typedef per specificare il tipo di dato memorizzato
  - Permette di inizializzare una matrice NxM di elementi (assumiamo che il typedef sia fatto per float)
  - permette di leggere o modificare un valore
  - permette di sommare due matrici
  - permette di moltiplicare la matrice per uno scalare α
  - permette di moltiplicare la matrice con un'altra di dimensione MxK

Note sull'esercizio

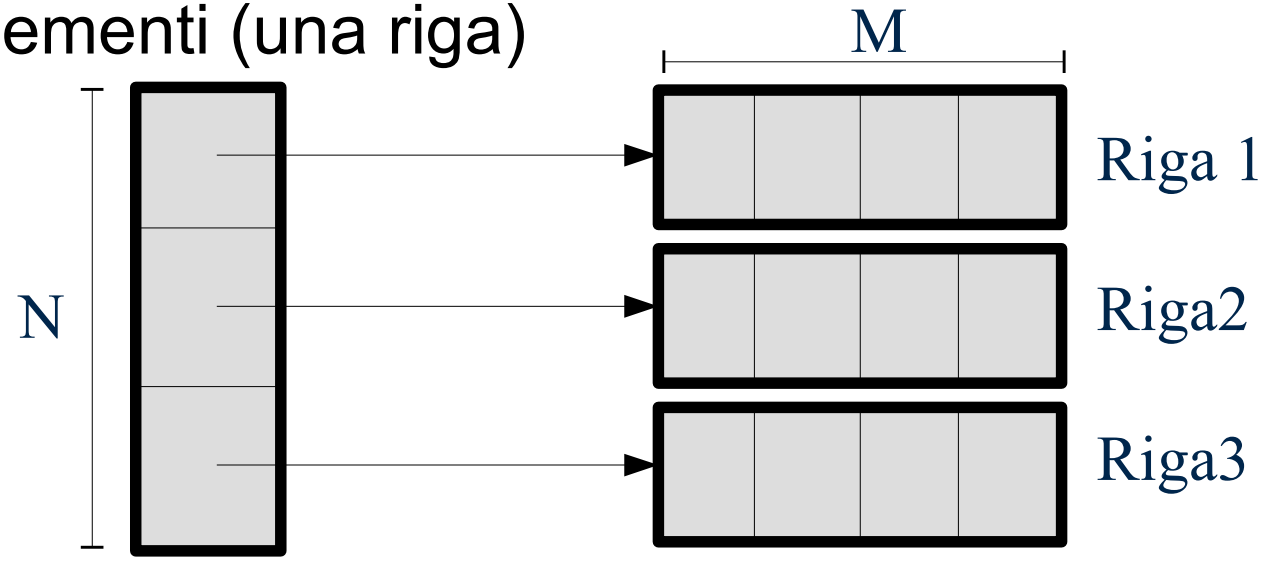
- Potete decidere voi come allocare la matrice
  - Soluzione 1: unico array con N*M elementi



  - Dati memorizzati tramite puntatore float* chiamato data
  - Allocazione: data = new float[N*M];
  - Deallocazione: delete[] data;
  - Con questa soluzione, dati gli indici (i,j) dovete voi trovare la posizione dell'elemento nel vettore

Note sull'esercizio

- Soluzione 2: array di N puntatori ad array di dimensione M elementi (una riga)



  - Dati memorizzati tramite doppio puntatore float**
  - Allocazione: data = new float*[N];

    for (int i = 0; i < N; ++i) data[i] = new float[M];
  - Deallocazione: for (int i = 0; i < N; ++i) delete[] data[i];
    delete[] data;
  - Accesso ad elemento (i,j) semplicemente: data[i][j]

Esercizio: commento

- Se una classe ha allocazione dinamica
  - Costruttore/distruttore devono gestire allocazione e deallocazione della memoria
  - Talvolta risulta comodo poter reinizializzare un'istanza già inizializzata
    - Come non duplicare il codice del costruttore/distruttore?
    - Possibile Soluzione

      1) Definire metodo Clear() che dealloca l'istanza

      2) Definire metodo Init() che prima chiama Clear() e poi alloca/inizializza l'istanza

      3) Distruttore chiama Clear() e riusa il suo codice

      4) Costruttori chiamano Init() risfruttando il suo codice

Esercizio: interfaccia Matrix

```
#ifndef MATH_MATRIX_H_
#define MATH_MATRIX_H_
namespace Math {
class Matrix {
 public:
  typedef float T;
 private:
  int N; int M;
  T** data;
  bool Init(const int N_, const int M_);      // Inizializza la matrice N*M
  void Clear();                               // Svuota e dealloca la Matrice
 public:
  Matrix();                                   // costruttore di default, matrice vuota
  Matrix(const int N_, const int M_);         // crea matrice N*M, chiama Init()
  Matrix(const Matrix& matrix);         // copia matrix, chiama Init()
  ~Matrix();                            // distrugge Matrix (chiama Clear)
  … // CONTINUA SU SLIDE DOPO
};  // end Matrix
}  // end Math
#endif // MATH_MATRIX_H_
```

Esercizio: interfaccia Matrix

```
#ifndef MATH_MATRIX_H_
#define MATH_MATRIX_H_
namespace Math {
class Matrix {
  … // COME SU SLIDE PRECEDENTE
  // Getters
  inline int GetM() const {return M; }              // numero colonne
  inline int GetN() const {return N; }              // numero righe
  inline T Get(const int i1, const int i2) const { return data[i][j]; }
  // Setters
  inline void Set(const int i1, const int i2, const T& val);   // setta un elemento a val
  void Set(const T& val);                                      // setta tutti gli elementi a val
  // Operazioni
  void Add(const Matrix& m);                        // aggiunge m all'istanza attuale
  static bool Add(const Matrix& m1, const Matrix& m2, Matrix* m3);       // m3 = m1+m2
  void Multiply(const T& val);                      // istanza * val
  void Print() const;                               // stampa l'istanza
};  // end Matrix
}  // end Math
#endif // MATH_MATRIX_H_
```

Esercizio: soluzione parte 1

```
void Matrix::Clear() {  // dealloca e azzera puntatori e dati
  if (data != NULL) {
    for (int i = 0; i < N; ++i)
      if (data[i] != NULL) delete[] data[i];
      delete[] data;
  }
  M = 0; N = 0; data = NULL;
}
bool Matrix::Init(const int N_, const int M_) {
  this->Clear();  // prima di dealloca, necessario se la matrice era già allocata
  M = M_; N = N_;
  if (N <= 0 || M <= 0) {
    cerr << "Can not generate matrix " << N << "x" << M << endl; return false;
  }
  if (M > 0 && N > 0) {
    data = new T*[N];
    for (int i = 0; i < N; ++i) data[i] = new T[M];
  }
  return true;
}
```

Inizializza la matrice. Funziona sia se la matrice era già esistente o se viene creata adesso

Esercizio: soluzione parte 1

```
void Matrix::Clear() {  // dealloca e azzera puntatori e dati
  if (data != NULL) {
    for (int i = 0; i < N; ++i)
      if (data[i] != NULL) delete[] data[i];
      delete[] data;
  }
  M = 0; N = 0; data = NULL;
}
bool Matrix::Init(const int N_, const int M_) {
  this->Clear();  // prima di dealloca, necessario se la matrice era già allocata
  M = M_; N = N_;
  if (N <= 0 || M <= 0) {
    cerr << "Can not generate matrix " << N << "x" << M << endl; return false;
  }
  if (M > 0 && N > 0) {
    data = new T*[N];
    for (int i = 0; i < N; ++i) data[i] = new T[M];
  }
  return true;
}
```

Inizializza la matrice. Funziona sia se la matrice era già esistente o se viene creata adesso

Esercizio: soluzione parte 2

```
Matrix::Matrix(const int N_, const int M_) {
  N = 0; M = 0; data = NULL;
  this->Init(N_, M_);  // costruttore chiama Init() e riusa codice
}

Matrix::~Matrix() {  // distruttore risfrutta codice di Clear()
  this->Clear();
}

Matrix::Matrix(const Matrix& matrix) {
  N = 0; M = 0; data = NULL;
  this->Init(matrix.N, matrix.M);  // anche qui si sfrutta il metodo Init()

  for (int i = 0; i < N; ++i)  // si fa la copia
    for (int j = 0; j < M; ++j)
      data[i][j] = matrix.Get(i, j);
}
```

Esercizio: soluzione parte 3

- Soluzione 1 per Add static

```
bool Matrix::Add(const Matrix& m1, const Matrix& m2, Matrix* m3) {
  if (m1.N != m2.N || m1.M != m2.M)  return false;
  m3->Init(m1.N, m1.M);
  for (int i = 0; i < m3->N; ++i)
    for (int j = 0; j < m3->M; ++j)
      m3->Set(i, i, m1.Get(i,j) + m2.Get(i,j));
  return true;
}
```

ATTENZIONE: Init() e Clear() e la gestione della reinizializzazione non sono strettamente necessari. Servono a gestire elegantemente questo caso.
Allocate direttamente in costruttore e deallocate in distruttore altrimenti.

- Soluzione 2 per Add static

```
Matrix* Matrix::Add(const Matrix& m1, const Matrix& m2) {
  if (m1.N != m2.N || m1.M != m2.M)  return NULL;
  Matrix* m3 = new Matrix(m1.N, m1.M);
  for (int i = 0; i < m3->N; ++i)
    for (int j = 0; j < m3->M; ++j)
      m3->Set(i, i, m1.Get(i,j) + m2.Get(i,j));
  return m3;
}
```

In questo caso non serve Init() e Clear(). Pero' chi chiama il metodo deve gestire il puntatore allocato dinamicamente e ricordarsi di deallocarlo. Design classe semplice ma maggiore attenzione richiesta al chiamante.

Friends

- La keyword **friend** permette di rompere l'incapsulazione
  - Possibile accedere a membri e metodi privati di una classe
  - Dichiarata dentro la classe che mette disponibili i suoi metodi e dati privati
  - Sintassi

```
class NomeClass {
  friend NomeFunzione();    // funzione vede tutto della classe NomeClass
  friend class NomeClassAmica;  // tutti i metodi accedono a NomeClass
  friend NomeClassAmica::NomeMetodo(i);  // metodo che vede tutto di NomeClass
};
```

Friends: esempio

```
class Integer {
  friend class TestInteger;
 private:
  int i;
 public:
  Integer(int j) { i = j; }
  inline void Add(int j) { i += j; }
};
class TestInteger {  // classe che testa Integer
 public:
  TestInteger() {
    Integer integer(3); integer.Add(5);
    if (integer.i != 8)
      std::cerr << “Errore: i=” << integer.i << std::endl;
    else
      std::cout << "Corretto: i=" << integer.i << std::endl;
  }
};
```

Classe amica, accede ai membri privati per controllare che tutto funzioni come deve

Friends

- Ci sono casi in cui la keyword friend è una buona soluzione. Casi tipici:
  - Testing della classe
  - Operator overloading (lo vedremo)
- Non bisogna abusare della keyword
  - Rompe la pulizia della programmazione ad oggetti
  - Se necessaria troppo spesso è indice di un design errato

Operator overloading

- Sommare due interi o float lo si fa con un semplice

  a+=b;

- Tuttavia sommare due vettori o matrici richiede chiamare un metodo, ad esempio:

  a.Add(b);

- C++ fornisce modo per definire come gli operatori lavorano su oggetti appartenenti di una classe
  - Ad esempio com e+,*,-,=,+=,<,>
  - Definita la += per matrici, la somma la scrivo come se le due matrici fossero interi:

    a+=b;

Operator overloading

- Vantaggi
  - Rendono la sintassi molto più leggera
  - Uso degli oggetti intuitivo ed elegante
- Svantaggi
  - Meno chiaro cosa succede dietro il codice
    - Esempio1: alcuni operatori fanno una copia dell'oggetto nel ritornarlo, lento per oggetti grandi, lo vedremo
    - Esempio2, *= definito per matrici

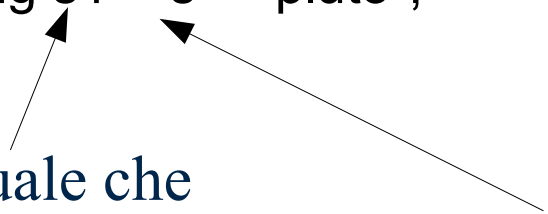
      Matrix a(1000, 1000);

      Matrix b(1000, 1000);

      a *= b; // l'innocente comando costa 1000^3 operazioni!

Operator overloading

- Usato molto nelle librerie standard
- Ad esempio, quando scrivete
  - string s(“pippo”);
  - string s1 = s + “pluto”;



Overloading dell'uguale che copia le stringhe

Overloading della somma che fa concatenazione tra stringhe

Operator overloading

- Sintassi semplice, come funzione o metodo operator@(argomenti) dove @ è un operatore
- Non possibile definire operatori personalizzati
  - Solo quelli che già esistono
  - Ad esempio non si può definire un operatore operator** perché non esiste in C++
    - operator@ in effetti non esiste!
  - Definito l'operatore la sua sintassi e numero di argomenti è fissata
  - Tutti gli operatori del C/C++ possono essere overloaded

Overloading di operatori unari

- Non coinvolgono altro che l'istanza, esempio

```
class Integer {
 private:
  friend const Integer& operator++(Integer& a);
  int i;
 public:
  Integer(int j) { i =j; }
  inline int Get() const { return i; }
};

const Integer& operator++(Integer& a) {
  a.i++;
  return a;
}
// Utilizzo
Integer a(5);
++a;
std::cout << a.Get() << std::endl;
std::cout << (++a).Get() << std::endl;
```

Operatore dichiarato friend perché modifica i dati privati della classe

Implementazione dell'operatore

Operatore ritorna argomento stesso, in modo che sia possibile concatenare operatori

Utilizzo dell'operatore

Possibile concatenare

Overloading di operatori unari

- Caso in cui non serve dichiarare friend l'operatore

```
class Integer {
 private:
  int i;
 public:
  Integer(int j) { i =j; }
  inline void Set(int j) { i = j; }
  inline int Get() const { return i; }
};

const Integer& operator++(Integer& a) {
  a.Set(a.Get() + 1);
  return a;
}
```

Operatore non dichiarato friend

Metodi pubblici mi permettono di fare quello che mi serve.

Dichiarare un operator come friend solo se necessario!

Overloading di operatori unari

- Possibile alternativamente definirli come metodi
  - Interni alla classe, non serve farli friend
  - Non serve passare l'istanza come argomento

```
class Integer {
  // Come prima
 public:
  const Integer& operator++();
};

const Integer& Integer::operator++() {
  i++;
  return *this;
}
// Utilizzo identico a prima
Integer a(5);
++a;
std::cout << (++a).Get() << std::endl;
```

Definizione dell' operatore come metodo

Implementazione dell'operatore

Operatore ritorna se stesso, in modo che sia possibile concatenare operatori

Overloading di operatori binari

- Prendono due argomenti in ingresso
  - Esempio per Integer (libro di testo ha lista completa)
  - friend opzionale, solo se alterano dati privati
  - Operatori aritmetici

```
[friend] Integer operator+(const Integer& left, const Integer& right);
[friend] Integer operator-(const Integer& left, const Integer& right);
[friend] Integer operator*(const Integer& left, const Integer& right);
[friend] Integer operator/(const Integer& left, const Integer& right);
[friend] Integer& operator+=(Integer& left, const Integer& right);
[friend] Integer& operator-=(Integer& left, const Integer& right);
[friend] Integer& operator*=(Integer& left, const Integer& right);
[friend] Integer& operator/=(Integer& left, const Integer& right);
```

Overloading di operatori binari

- Prendono due argomenti in ingresso
  - Esempio per Integer (libro di testo ha lista completa)
  - friend opzionale, solo se alterano dati privati
  - Operatori di confronto

```
// Operatori di confronto
[friend] bool operator==(const Integer& left, const Integer& right);
[friend] bool operator!=(const Integer& left, const Integer& right);
[friend] bool operator<(const Integer& left, const Integer& right);
[friend] bool operator>(const Integer& left, const Integer& right);
[friend] bool operator<=(const Integer& left, const Integer& right);
[friend] bool operator>=(const Integer& left, const Integer& right);
```

Overloading di operatori binari

- Operatori binari implementazione: alcuni esempi

```
Integer operator+(const Integer& left, const Integer& right) {
  return Integer(left.i + right.i);  // copia dell'oggetto! Lento per oggetti grandi
}

Integer& operator+=(Integer& left, const Integer& right) {
  left.i += right.i;
  return left;  // ritorna left che non è const, infatti += modifica l'oggetto sinistro
}

Integer operator*(const Integer& left, const Integer& right) {
  return Integer(left.i * right.i);
}
Integer& operator*=(Integer& left, const Integer& right) {
  left.i *= right.i;
  return left;
}
```

Overloading di operatori binari

- Operatori binari implementazione: alcuni esempi

```
bool operator==(const Integer& left, const Integer& right) {
  return left.i == right.i;
}

bool operator<(const Integer& left, const Integer& right) {
  return left.i < right.i;
}
```

Overloading di operatori binari

- Possibile anche mischiare i tipi ma rispettando la forma dell'operatore
  - Esempio, operatore che somma un Integer ed un int

```
// Definizione
Integer& operator+=(Integer& left, const int& right);

// Implementazione
Integer& operator+=(Integer& left, const int& right) {
  left.i += right;
  return left;
}
// Utilizzo
Integer i(3);
int j = 5;
i+=j;
```

Overloading di operatori binari

- Operatori binari possono essere dichiarati esterni alla classe
  - Spesso in tal caso sono dichiarati friend
- Oppure possono essere dichiarati come metodi

```
class Integer {
 public:
  int a;
  Integer(int a_) : a(a_) { }
  const Integer operator+(const Integer& i) { return Integer(a + i.a); }
};
```

- Entrambe valide
  - Consiglio di usare la soluzione esterna vista per ora
  - Solo per evitare confusione, sono equivalenti

Overloading di operatori binari

- Soluzione 1: operatori esterni

```
class Integer {
 public:
  int a;
  Integer(int a_) { a = a_; }
  friend Integer operator+(const Integer& a,
                           const Integer& b);
  friend Integer& operator+=(Integer& a,
                             const Integer& b);
};
const Integer operator+(const Integer& a,
                        const Integer& b) {
  return Integer(a.a + b.a);
}
Integer& operator+=(Integer& a, const Integer& b) {
  a.a += b.a; return a;
}
```

- Soluzione 2: operatori metodi

```
class Integer {
 public:
  int a;
  Integer(int a_)  { a = a_; }
  Integer operator+(const Integer& b) {
    return Integer(a + b.a);
  }
  Integer& operator+=(const Integer& b) {
    a += b.a;  return *this;
  }
};
// Main comune ai 2 casi
int main() {
  Integer i(3); Integer j(2);
  i += j;
  Integer l = i + j;
  std::cout << i.a << " " << l.a <<std::endl;
  return 0;
}
```

operator[]

- Definisce cosa fare quando si chiama istanza[argomento]
  - Lo vedemmo usato dalla classe std::string
  - Esempio

```
class Vector {...
  Element& operator[](int i) {
    return data[i];
  }
};
Vector v(10);
cout << v[4];
```

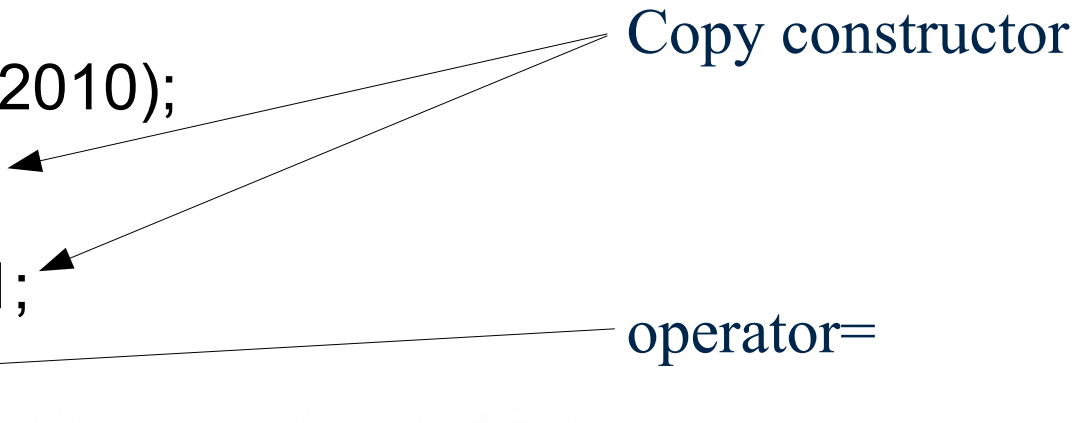
operator=

- Se non definito, compilatore ne implementa uno
  - Default fa byte-copy dell'oggetto
    - Come il copy constructor di default
  - Se copy constructor va definito allora
    - va definito anche l'operator=
  - Questo accade quando la classe ha dati membri allocati dinamicamente
    - byte copy non copia il contenuto del puntatore ma solo l'indirizzo

operator=

- Vi è un operatore speciale detto operator=
  - Simile al Copy Constructor
  - **Deve** essere definito come metodo
    - Non possibile dichiararlo friend
  - Definisce cosa fare quando si assegna un istanza ad un'altra
  - Esempio

```
Date d1(2,3,2010);
Date d2(d1);
Date d3 = d1;
d2 = d1;
```



operator=

- Se non definito, compilatore ne implementa uno
  - Default fa byte-copy dell'oggetto
    - Come il copy constructor di default
  - Se copy constructor va definito allora
    - va definito anche l'operator=
  - Questo accade quando la classe ha dati membri allocati dinamicamente
    - byte copy non copia il contenuto del puntatore ma solo l'indirizzo

operator=

- Sintassi dell'operator= per generica classe C

```
class C { …

C& operator=(const C& right) {
   if(this != &right) {
        …
   }
   return *this;
  }
};
```

operator= sta dentro la classe, non è friend che poi viene definito fuori

Gestisce self-assignment a=a;

Copia dei campi, codice specifico per classe C

Ultima riga del metodo sempre return this*;

operator=: esempio

```
class Vector { ...
  Vector& operator=(const Vector& vec) {
    if(this == &vec)  return *this;              // evita self-assignment
    if (data) delete[] data;                     // dealloca eventuali dati precedenti
    N = vec.N;
    data = new Element[N];                       // rialloca lo spazio
    memcpy(data, vec.data, N*sizeof(Element));  // copia
    return *this;                                // ritorna se stesso
  }
};

int main() {
  Vector vec(5);
  Vector vec1(vec);      // chiama copy constructor
  Vector vec2 = vec;     // chiama copy constructor
  vec1 = vec;            // chiama operator=
}
```

operator<< ed operator>>

- Se definito, permette di inserire un oggetto arbitrario in uno stream, sia in input che output
  - Esempio

```
class C { …
  friend std::ostream& operator<<(std::ostream& out, const C& c);
  friend std::istream& operator>>(std::istream& in, C& c);
};
std::ostream& operator<<(std::ostream& out, const C& c) {
  out << c.x() << "::" << c.y() << endl;
  return out;
}
std::istream& operator>>(std::istream& in, C& c) {
    int x, y;
    in >> x >> y;
    c.SetX(x); c.SetY(y);
    return in;
}
```

operator<<: esempio

```
class Vector { …
  friend std::ostream& operator<<(std::ostream& out, const Vector& vec);
  … // resto classe come prima
};

// Implementazione dell'operatore
std::ostream& operator<<(std::ostream& out, const Vector& vec) {
  for (int i = 0; i < vec.N; ++i)  out << vec.Get(i) << " ";
  out << endl;
  return out;
}

int main() {
  Vector vec(5);
  vec.Set(3, 2);  vec.Set(1, 1);
  std::cout << vec;  // si chiama operator<<
  return 0;
}
```

Esercizio

- Implementare una classe libreria per la gestione di matrici (classe Matrix) con
  - operator= (consiglio riusare Clear() ed Init())
  - operator+ tra matrici
  - operator== tra matrici
  - operator+= tra matrici
  - operator*= tra matrici
  - operator*= tra matrice ed scalare
  - operator<<
  - Chiamare gli operatori dal main
    - Anche scrivere (e leggere) la matrice da(su) file