Accesso ai membri di una classe

- La visibilità dall'esterno dei dati membro e delle funzioni membro di una classe dipende dalle restrizioni imposte
- All'interno di una classe tutti i membri sono visibili e accessibili tramite il proprio nome
- Le variabili definite all'interno di una funzione membro sono visibili solo alla funzione stessa
- Per accedere dall'esterno si usano (sugli oggetti di una classe) gli stessi operatori di accesso usati per le struct "." e "**->**"

```
nomeOggetto.nomeFunzione (valori dei parametri)
```

- Possibili tipi di controllo ai membri di una classe sono:

public, private, protected

Accesso ai membri di una classe

- è possibile controllare la visibilità esterna mediante specificatori d'accesso:
  - la sezione *public* contiene membri a cui si può accedere dall'esterno della classe
  - la sezione *private* contiene membri ai quali si può accedere solo dall'interno della classe
  - ai membri che seguono lo specificatore *protected* si può accedere dall'interno della classe ed eventualmente anche da metodi di classi derivate (per i dettagli vedere slide su ereditarietà)

```
class NomeClasse
{
   public:
      Sezione pubblica   // dichiarazione di membri pubblici
   protected:
      Sezione protetta   // dichiarazione di membri protetti
   private:
      Sezione privata    // dichiarazione di membri privati
};
```

Accesso ai membri di una classe

```
#include <iostream>
using namespace std;
class Count {
 public: int x;
 void print() { cout << x << endl; } };


int main() {
 Count counter; // create counter object
 Count *counterPtr = &counter; // pointer to counter
 Count &counterRef = counter; // reference to counter
 cout << "Assign 7 to x and print using the object's name: ";
 counter.x = 7; // assign 7 to data member x
 counter.print(); // call member function print
 cout << "Assign 8 to x and print using a reference: ";
 counterRef.x = 8; // assign 8 to data member x
 counterRef.print(); // call member function print
 cout << "Assign 10 to x and print using a pointer: ";
 counterPtr->x = 10; // assign 10 to data member x
 counterPtr->print(); // call member function print
 return 0; }
```

Accesso ai membri di una classe

- Lo scopo dei membri *public* è di presentare a chi utilizza una classe una interfaccia di servizi disponibili
- i dati e le funzioni *public* sono accessibili a tutte le altre funzioni del programma
- i membri *private* sono accessibili solo alle funzioni membro di tale classe. Non è possibile per alcuna funzione esterna accedere ai dati o funzioni *private* di una classe
- Vale sempre il principio del minimo privilegio: *Consentire il livello necessario di accesso e niente di più*
- i dati membro dovrebbero essere privati
- le funzioni membro dovrebbero essere pubbliche
- funzioni membro di servizio (utilizzate da altre funzioni membro pubbliche) dovrebbero essere private

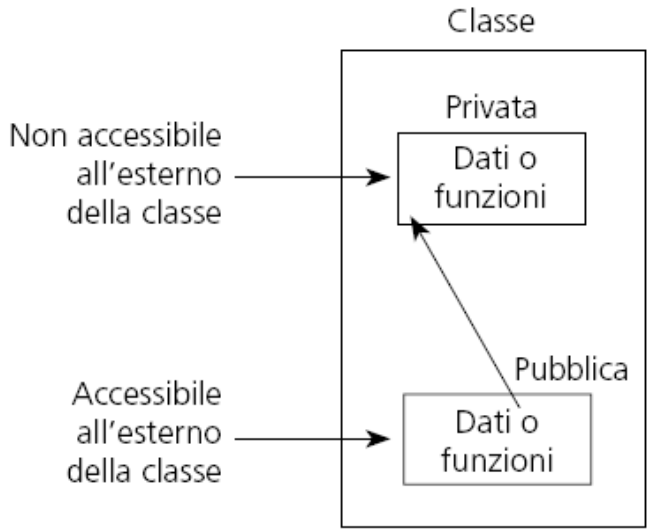
Accesso ai membri di una classe

```
class Tempo {
  public:
    Tempo();
    void set(int, int);
    void stampa();
    void stampa_secondi();
  private:
    int ora;
    int min;
    int converti_sec();
  };

  void Tempo::stampa_secondi() {
    int sec = converti_sec();
    cout << "Secondi da mezzanotte : " << sec << endl; }

  int Tempo::converti_sec() {
    return min * 60 + ora * 3600;
}
```

Accesso ai membri di una classe

- Un principio fondamentale di buona ingegneria del software è separare *l'interfaccia dall'implementazione*
- Tutto quello che deve sapere un programma client è contenuto nelle dichiarazioni public di una classe
- L'implementazione delle funzioni e tutto ciò che è private può essere cambiato in modo trasparente senza dover effettuare alcuna modifica sul resto programma

```
class Semaforo
{
   public:
      void cambiareColore();
      //...
   private:
      int c;
};
```



Accesso ai membri di una classe

- La dichiarazione di una classe va in un header file (es. Tempo.h)
- Le definizioni delle funzioni membro vanno in un file sorgente che viene collegato in fase di link dal compilatore (es. Tempo.cpp), il file sorgente include il corrispondente file header
- Tutti i file sorgenti del progetto che utilizzano una certa classe devono includere il suo header
- È comunque possibile definire il codice dei metodi anche all'interno della classe (nel file header, solitamente per funzioni brevi)
- È indifferente collocare prima la sezione pubblica o quella privata; è però consigliabile collocare la sezione pubblica prima per mettere in evidenza le operazioni che fanno parte dell'interfaccia utente pubblica
- Le parole chiavi *public* e *private* seguite da due punti, segnalano l'inizio delle rispettive sezioni pubbliche e private; benché non sia comune, una classe può avere varie sezioni pubbliche e private

sovraccarico (overloading) di metodi

- l'overloading permette di utilizzare uno stesso nome per più metodi che si distingueranno solo per i parametri passati all'atto della chiamata
- due funzioni membro sovraccaricate non possono avere lo stesso numero e tipo di parametri

```
class Prodotto
{
public:
   int prodotto (int m, int n);                  // metodo 1
   int prodotto (int m, int p, int q);           // metodo 2
   int prodotto (float m, float n);              // metodo 3
   int prodotto (float m, float n, float p);     // metodo 4
}
```

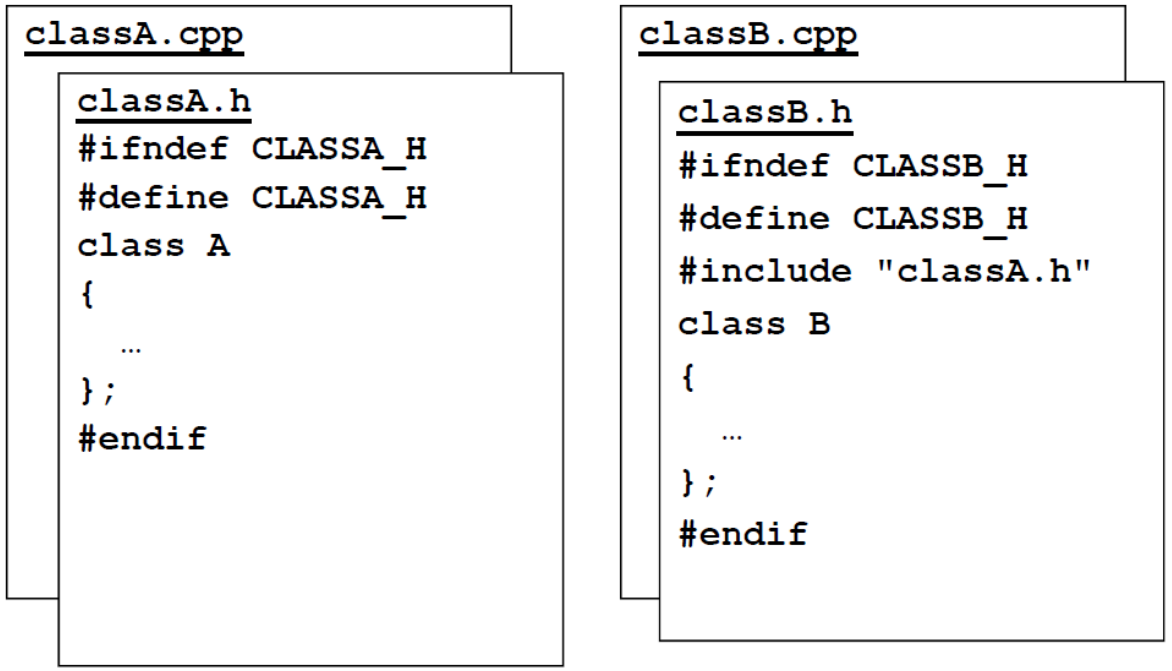
Inclusione condizionale

- Le direttive #ifndef e #define permettono di evitare l'inclusione ripetuta dello stesso file header in programmi costituiti da più file

```
#ifndef TIME1_H
#define TIME1_H
class Tempo {
  public: Tempo();
   void set(int, int);
   void stampa();
   void stampa_secondi();
  private:
   int ora;
   int min;
   int converti_sec(); };
#endif
```

Inclusione condizionale

- Usando *#ifndef var* il pre-processore accede alla parte rimanente del codice solo nel caso in cui non sia mai stata definita *var*
- Ma l'istruzione successiva definisce proprio *var,* in questo modo la prima volta che si incontra #ifndef si includerà la definizione mentre la seconda volta no. Si dovrebbe **sempre** utilizzare questa tecnica



Costruttori

- All'atto della creazione di un oggetto di una classe è possibile inizializzarne i dati membro tramite il *costruttore di classe*
- un costruttore è una funzione membro speciale di una classe che viene automaticamente eseguito all'atto di creazione di un oggetto
- il nome del costruttore è lo stesso del nome della classe
- i costruttori non restituiscono nessun valore (nemmeno void)
- il costruttore o è scritto esplicitamente oppure ne esiste uno di default

```
class Contatore {
  public:
    Contatore();
        .
        .
};
```

```
Contatore::Contatore()
{
 value = 0;
}
```

```
Contatore cont1, cont2; //  viene chiamato il costruttore senza argomenti per
                        // per ciascun oggetto
```

Costruttori

- i costruttori possono ricevere dei parametri (valori di inizializzazione)
- Si possono avere diversi costruttori per casi differenti di inizializzazione utilizzando l'overloading
- Si possono (dovrebbero) utilizzare costruttori con argomenti di default
- Nel caso si dichiari un costruttore che riceve parametri allora il compilatore non genera automaticamente il costruttore di default, per cui si potranno usare solo i costruttori esplicitamente dichiarati

```
class Contatore {
  public:
    Contatore();
    Contatore(int val);
        .
        .
};
```

```
Contatore::Contatore()
{
 value = 0;
}
Contatore::Contatore(int val)
{
 value = val;
}
```

```
Contatore cont1(10);  //  viene chiamato il costruttore parametrizzato
```

Distruttore

- Il distruttore viene chiamato automaticamente quando l'esecuzione del programma oltrepassa lo scope in cui l'oggetto è istanziato
- Viene anche chiamato automaticamente quando viene eseguito l'operatore *delete* su puntatori ad oggetti allocati dinamicamente
- il nome del distruttore è composto da una tilde "~" seguito dal nome della classe (è una specie di complemento del costruttore)
- Un distruttore non riceve parametri e non restituisce alcun valore (nemmeno void)
- Una classe può avere un solo distruttore
- Se non si dichiara esplicitamente ne viene creato uno di default
- **Occorre definire esplicitamente il distruttore di una classe quando il costruttore alloca memoria dinamica. Il distruttore viene utilizzato per liberare la memoria dinamica allocata da un oggetto.**

Costruttore/distruttore

```
#include <iostream>
using namespace std;
class Line {
   public:
     void setLength(double len);
     double getLength(void);
     Line(double len); // This is the constructor
     ~Line(); // This is the destructor: declaration
   private:
     double length; };

Line::Line(double len) {
  cout << "Object is being created, length = " << len << endl;
  length = len; }

Line::~Line(void) { cout << "Object is being deleted" << endl; }

void Line::setLength(double len) { length = len; }
double Line::getLength(void) { return length; }

int main() {
    Line line(10.0); // get initially set length.
    cout << "Length of line : " << line.getLength() << endl;
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() << endl;
return 0; }
```

Output:

Object is being created, length = 10
Length of line : 10
Length of line : 6
Object is being deleted

Definizione oggetti

- creazione di un oggetto di una classe con inizializzazione dei dati membro tramite costruttore:

```
Rettangolo rect(25, 75,25,75); // rect è ISTANZA di Rettangolo
```

- Il costruttore viene chiamato anche quando un oggetto viene allocato dinamicamente:

```
Rettangolo* nr = new Rettangolo(25, 75,25,75); // nr punta
                                // una nuova ISTANZA di Rettangolo

...
delete nr; // viene liberata la memoria e chiamato il distruttore
```

Inizializzazione di un array di oggetti

```
class student {
        public:
                string name;
                int x, y; // (x , y Axis) Student Location in class
        public:
          student(int a, int b, string c)
          { x = a; y = b; name = c; }
};

int main(int argc, char** argv)
{
 student a[3] = { student(1,3,"Mary"), student(1,2,"pippo"), student(4,2,"topolino") };
 std::cout << a[0].x << " " << a[0].y << " " << a[0].name << endl;
 std::cout << a[1].x << " " << a[1].y << " " << a[1].name << endl;
 std::cout << a[2].x << " " << a[2].y << " " << a[2].name << endl;
}
```

Output:

1 3 Mary

1 2 pippo

4 2 topolino

Costruttore di copia

- Esiste un tipo speciale di costruttore detto costruttore di copia
- Un costruttore di copia serve per inizializzare un oggetto tramite un altro oggetto
- il costruttore di copia è una funzione membro che ha lo stesso nome della classe e ha come argomento un riferimento costante ad un oggetto della stessa classe

NomeClasse(const NomeClasse &)

- Il compilatore crea un costruttore di copia bit a bit di default, cioè ogni dato membro viene copiato
- Ci sono tre casi in cui viene invocato il costruttore di copia (vedi prossima slide)

Costruttore di copia

- inizializzazione esplicita

```
Time t1(12,0);
Time t2(t1);
Time t3=t1;
```

- passaggio per valore ad una funzione

```
void main(){
 Time t(12,0);
 func(t);}
void func(Time myT){ cout<<myT;}
```

- restituzione di un oggetto temporaneo

```
void main(){
  Time t(12,0), t2;
  t2=func(t); }

Time func(Time myT){
   myT.ora+=1;
   return myT; }
```

Costruttore di copia

- E' buona pratica creare esplicitamente un costruttore di copia quando all'interno di in un oggetto vi sono dati membro puntatori su cui avviene una allocazione dinamica della memoria
- Per oggetti con dati membro puntatori che allocano dinamicamente la memoria, la copia bit-a-bit di default copierebbe solo l'indirizzo di memoria a cui punta il puntatore, pertanto se un oggetto B fosse inizializzato come copia di un oggetto A i puntatori in essi contenuti risulterebbero puntare alla stessa area di memoria

```
class Array{
  public:
   Array(int usr_size=10){
    size=usr_size;
    ptr=new int[size]; }

   Array(const Array& copia){
    size=copia.size;
    ptr=new int[size];
    for(int i=0;i<size;i++)
       ptr[i]=copia.ptr[i]; }
   ~Array(){delete [] ptr;}
  private:
    int *ptr; int size; };
```

Costruttore di copia

- in questo corso utilizzeremo poco i costruttori di copia
- nelle classi che vedremo per le strutture dati dinamiche (es:liste) i costruttori di copia saranno DISABILITATI dichiarandoli come membri privati e senza implementazione:

```
private:
  List(const List&) {} // Protect copy constructor
```

```
List L2(L1); //errore di compilazione
```

Ancora sul qualificatore const

- Alcuni oggetti devono poter essere modificati, altri no
- E' possibile dichiarare un oggetto costante (immodificabile) con

const Tempo mid(12,0);

- Ad es. l'oggetto *mid* che rappresenta mezzogiorno può essere utilizzato come costante in operazioni di confronto
- Un oggetto costante può essere solo inizializzato
- Qualsiasi tentativo di modificare un oggetto *const* genera un errore in fase di compilazione

Ancora sul qualificatore const

- Anche i dati membro possono essere di tipo costante
- Un dato membro di una classe dichiarato come const non può essere usato a sinistra di operazioni di assegnamento
- Un dato membro const può solo essere inizializzato

```
class Data{
  public:
   Data(int usr_dat=0):dat(usr_dat){}
   const int dat;
};
```

Ancora sul qualificatore const

- Anche le funzioni membro possono essere di tipo costante
- E' un errore se una funzione membro costante tenta di modificare un qualsiasi dato membro: **una funzione membro costante non può modificare dati membro di alcun tipo**
- ..ma internamente alla funzione possono essere dichiarate *e modificate* variabili locali
- Non è possibile chiamare funzioni membro per oggetti const *a meno che* le funzioni non siano dichiarate const

```
//nella dichiarazione di classe
int restituisci_ora() const;

//nella definizione
int Tempo::restituisci_ora() const {
   return ora;
}
```

Variabili static

- Una variabile statica è locale ad una particolare funzione.
- E' inizializzata una sola volta, la prima volta che tale funzione viene chiamata e il suo valore resta inalterato quando si esce dalla funzione, per cui quando si richiama nuovamente la funzione tale variabile ha ancora il valore assegnatogli precedentemente.
- Per definire statica una variabile è sufficiente anteporre la parola *static* alla dichiarazione della variabile.

```
void main(void) {
  incrementa();
  incrementa();
  incrementa(); }

void incrementa(void) {
    static int x = 0;
    ++x;
    cout << x <<endl; }
```

```
Output:
1
2
3
```

Variabili static

- anche i dati membro di una classe possono essere definiti come *static*
- viene creata una sola copia di quella variabile per tutti gli oggetti della classe
- tutti gli oggetti di quella classe nel programma utilizzeranno la stessa variabile statica
- è un modo per condividere la stessa informazione tra oggetti di una stessa classe

```
class Box {
  public:
       static int objectCount; // Constructor definition
  Box(double l = 2.0, double b = 2.0, double h = 2.0) {
       cout <<"Constructor called." << endl;
       length = l; breadth = b; height = h;
       // Increase every time object is created
       objectCount++; }
  double Volume() { return length * breadth * height; }
  private:
       double length; // Length of a box
       double breadth; // Breadth of a box
       double height; // Height of a box
};
```

Classi friend

- è possibile definire una classe come *friend* di un'altra classe
- tutte i metodi della classe dichiarata come *friend* hanno accesso ai membri privati della classe
- non vale il viceversa

```
#include <iostream>
using namespace std;
class XYZ {
       private: char ch='A';
       int num = 11;
       public: friend class ABC; };
class ABC {
       public: void disp(XYZ obj){
              cout<<obj.ch<<endl;
              cout<<obj.num<<endl; } };

int main() { ABC obj; XYZ obj2;
              obj.disp(obj2); return 0; }
```

Il puntatore this

- *this* identifica un puntatore speciale che contiene l'indirizzo dell'oggetto che ha invocato un metodo.
- l'uso della parola riservata *this* è esclusivamente confinato all'ambito della definizione dei metodi di una classe.
- Il tipo del puntatore *this* è quello di un puntatore alla classe
- viene utilizzato nella definizione di alcuni metodi speciali, come gli operatori, per la restituzione del valore di ritorno.

Overloading degli operatori

- In C++ è possibile eseguire l'overloading della maggior parte degli operatori per consentire loro di svolgere operazioni specifiche rispetto a classi definite dal programmatore
- Si realizza per mezzo delle funzioni ***operator***
- Vediamo alcuni modi per ri-definire un operatore:
  - funzione membro senza argomenti (operatore unario prefisso) oppure con un solo argomento (operatore binario)
  - funzione non membro con due argomenti (operatore binario)
- Nella maggior parte dei casi le funzioni operator restituiscono un oggetto della classe su cui operano

Overloading degli operatori

- Una funzione *operator* come membro di una classe ha la sintassi:

```
Tipo nome-classe::operator#(Tipo1 arg1)
{
  …
}
```

- dove # è il simbolo dell'operatore da sovraccaricare
- quando si esegue l'overloading di un operatore binario come membro di una classe, la funzione *operator* ha un solo argomento (è l'oggetto di sinistra nell'espressione a generare la chiamata all'operatore), se l'operatore è unario la funzione membro non ha argomenti
- Un operatore definito come funzione non membro ha la sintassi:

```
Tipo operator#(Tipo1 arg1, Tipo2 arg2)
{
  …
}
```

Overloading degli operatori

- Esempio: ‘’+’’ operatore binario come funzione membro, ‘’-’’ operatore binario come funzione non-membro, ‘’++’’ operatore unario prefisso

```
#include <iostream>
using namespace std;
class Complex {
 public: Complex(float r = 0.0, float i = 0.0) { re = r; im = i; };
 float getRe() const { return re; }; float getIm() const { return im; };
 void SetRe(float r) { re=r; }; void SetIm(float i) { im=i; };
 void show();
 Complex operator+(Complex op2); // operatore binario come funzione membro
 Complex operator++(); // operatore unario prefisso come funzione membro
 private: float re; float im; };

 void Complex::show() { cout << endl << "re: " << re << " im: " << im; }

 Complex Complex::operator+(Complex op2) {
 Complex tmp; tmp.re = re + op2.re; tmp.im = im + op2.im; return tmp; }

 Complex Complex::operator++() { re++; im++; return *this; }
```

Overloading degli operatori

```
// esempio di operatore "-" come funzione non-membro
Complex operator-(Complex op1, Complex op2) {
 Complex tmp;
 tmp.SetRe(op1.getRe() - op2.getRe());
 tmp.SetIm(op1.getIm() - op2.getIm());
return tmp; }

// esempio di operatore "<<" come funzione non-membro
ostream& operator<<(ostream& os, Complex op) {
 os << op.getRe();
 if (op.getIm() > 0) os << " + ";
 os << op.getIm()<< "i"; return os; }

int main() { Complex c1, c2(1, 1), c3(4, 5);
 c1.show(); c2.show(); cout << " " << c2;
 c3.show();
 c1 = c2 + c3; c1.show();
 c1 = c2 - c3; c1.show();
 ++c1; c1.show();
 c2 = ++c1; c2.show();
 return 0; }
```

Overloading degli operatori

- In C++ l'overloading è possibile sui seguenti operatori

+	-	*	/	%	^	&	|
	~						
!	=	<	>	+=	-=	/=	
	*=	%=					
^=	&=	|=	<<	>>	>=	<=	
	==	!=					
>>=	<<=	&&	||	++	--	->*	,
	->						
[]	()	new	new[]	delete	delete[]		

Overloading degli operatori

- Se si vuole permettere il flusso su stream (es: standard output) della propria classe occorre fare l'overloading dell'operatore << della classe *ostream* che contiene le funzionalità per le operazioni di output, per esempio come funzione non membro:

```
ostream& operator<<(ostream& os, Complex op) {
    os << op.getRe();
    if (op.getIm() >= 0)
       os << " + ";
    os << op.getIm()<< "i";
   return os;
}
```

c2(1, -1);

cout << c2;

Output: 1 - 1i

Overloading degli operatori

- L'operatore di assegnamento viene usato in tutte le espressioni in cui è presente il simbolo '='
- L'assegnamento di default effettua una copia bit-a-bit. Pertanto l'overloading è necessario per le classi che utilizzano la memoria dinamica
- E' definibile come void operator=(const C&)
- nelle classi che vedremo per le strutture dati dinamiche (es:liste) l'operatore di assegnamento sarà disabilitato dichiarandolo come membro privato e senza implementazione:

```
private:
  void operator =(const List&) {} // Protect assignment

List L2,L1;
L2=L1; //errore di compilazione
```

Overloading degli operatori

- Esempio:

```
#ifndef HEADERFILE_H
#define HEADERFILE_H
#include <iostream>
using namespace std;

class Distanza {
 private:
     int metri;
     int centimetri;
 public:
 Distanza(int m = 0, int c = 0){ metri = m; centimetri = c; }
 bool operator <(Distanza d);
 int getmetri() { return metri; }
 int getcentimetri() { return centimetri; }
};

ostream& operator<<(ostream& os, Distanza op); #endif
```

Overloading degli operatori

- Esempio:

```
#include "esempio_1_27_1_distanza.h"

bool Distanza::operator <(Distanza d) {
   if (metri < d.metri) { return true; }
   if ((metri == d.metri) && (centimetri < d.centimetri))
      { return true; }
   return false;
}


ostream& operator<<(ostream& os, Distanza op) {
   os << op.getmetri() << " " << op.getcentimetri();
   return os;
}
```

Overloading degli operatori

- Esempio:

```
#include "esempio_1_27_1_distanza.h"

int main() {
  Distanza D1(11, 10), D2(5, 11);
  cout << "D1=" << D1 << endl;
  cout << "D2=" << D2 << endl;
  if (D1 < D2) {
    cout << "D1 is less than D2 " << endl; }
  else {
    cout << "D2 is less than D1 " << endl; }
  return 0;
}
```

Funzioni inline

- Come regola i prototipi delle funzioni vanno definiti nei file header (.h), mentre la loro implementazione va definita nei file .cpp
- Talvolta per evitare il proliferare dei file sorgenti di un progetto è utile definire direttamente il corpo di una funzione nei file header
- Questa pratica di solito viene ammessa per ‘’piccole’’ funzioni (oppure per operatori definiti come funzioni non membro)
- per evitare problemi di compilazione è opportuno utilizzare la parola riservata **inline** prefissata alla dichiarazione della funzione; inline indica al compilatore di sostituire il codice della definizione di funzione per ogni istanza di chiamata della funzione stessa

Esempio d'uso in un file .h:

```
inline void funzione( )
{
  // contenuto della funzione
}
```

Eccezioni

- Il C++ supporta le eccezioni come funzionalità per gestire anomalie di esecuzione rilevabili in un programma a tempo di esecuzione
- Esempi di anomalie:
  - Divisione per 0
  - Accesso ad un elemento di un array con indice che eccede la dimensione massima
  - Esaurimento della memoria di sistema
- Quando un programma rileva una condizione di anomalia può comunicarlo sollevando (o ''lanciando'') una *eccezione*
- Le eccezioni si realizzano tramite i costrutti ***try, throw*** e ***catch***
- Le eccezioni consentono di trasferire il flusso di esecuzione del programma dal punto in cui si verifica l'eccezione ad un blocco di codice specifico che gestisce l'eccezione

Eccezioni

- Sintassi:

```
try {
  // codice che si vuole controllare
  throw exception;
}
catch(type_1 exception) {
  // trattamento di eccezioni di tipo 'type_1'.
}
catch(type_2 exception) {
  // trattamento di eccezioni di tipo 'type_2'.
}
```

- Blocco ***try:*** racchiude tutte le istruzioni che si intende controllare
- Espressione ***throw:*** genera una eccezione, l'espressione che segue la parola chiave *throw* ha come tipo di dato quello dell'eccezione lanciata
- Blocco ***catch:*** procedura di gestione dell'eccezione

Eccezioni

- Esempio:

```
#include <iostream>
using namespace std;
int main() {
    int n1 = 5; int n2 = 1;
    float res; float e=0.0;
    try {
          if (n2 == 0)
                      throw 'a';
                      //throw n2;
          else if (n2 == 1)
                      throw e;
          else {
                 res = (float) n1/n2;
                 cout << "res = " << res << endl; }
        }

    catch (int a) { cout << "Exception caught: Division by zero: " << a; }
    catch (float d) { cout << "Exception caught: n2=" << n2; }
    catch (...) { cout << "Generic exception caught"; }
}
```

Eccezioni

- Se non si verifica alcuna eccezione: il codice nel blocco *try* viene eseguito normalmente mentre il blocco di gestione dell'eccezione (*catch*) non viene eseguito
- Se viene lanciata una eccezione: viene eseguito il blocco *catch* corrispondente al tipo di eccezione che è stata lanciata. Se il blocco *catch* non contiene una istruzione *return*, dopo l'esecuzione del blocco di codice *catch* il flusso del programma riprende dall'istruzione che segue l'ultima clausola *catch*
- Se viene lanciata una eccezione ma non esiste nessuna clausola *catch* in grado di gestire l'eccezione l'esecuzione del programma termina
- Clausola *catch(…)* generica per gestire qualunque tipo di eccezione

Algoritmi e complessità

- Un problema (computazionale) specifica una relazione input-output
- Esempio:

un problema di ordinamento

  - Input: una sequenza di numeri naturali
  - Output: la sequenza ordinata

- Un algoritmo che risolve il problema specifica una procedura effettiva per ottenere la relazione desiderata
- Ogni problema può avere diverse soluzioni algoritmiche…o non averne alcuna. . .
- Qual è l'algoritmo "migliore"?
- Occorre un modo per confrontare l'efficienza degli algoritmi
- Bisogna definire che cosa si intende per "efficienza"

Algoritmi e complessità

- L'efficienza si può misurare sulla base delle risorse richieste dall'algoritmo
- Quali risorse? Principalmente:
  - tempo di esecuzione
  - spazio di memoria
- Soprattutto il tempo è un fattore importante: lo spazio si può riutilizzare, il tempo no
- In generale, il tempo di esecuzione cresce con la dimensione dell'input
- Se si potesse valutare il tempo di calcolo in unità standard come i secondi la valutazione del costo di un programma sarebbe semplice: basterebbe eseguire il programma misurando il tempo necessario per l'esecuzione.

Algoritmi e complessità

- Le unità di misura che si assumono in teoria della complessità non sono unità di tempo propriamente intese perché il confronto dei tempi di esecuzione di due programmi sarebbe lecito solo se i programmi fossero tradotti con il medesimo compilatore e fossero eseguiti con lo stesso elaboratore.
- Il tempo di esecuzione effettivo di un algoritmo dipende anche dal calcolatore
- Un processore Ghepardo a 1Ghz esegue un algoritmo di ordinamento che richiede $2n^2$ operazioni per ordinare n numeri
- Un processore Bradipo a 1Mhz (anni '80) esegue un algoritmo di ordinamento che richiede $50n \log_2 n$ operazioni su n numeri
- Tempo per ordinare 10^7 numeri

$$\text{Ghepardo: } \frac{2\cdot(10^7)^2}{10^9} = 2\cdot 10^5 \approx 56 \text{ ore}$$

$$\text{Bradipo: } \frac{50\cdot(10^7)\log_2 10^7}{10^6} \approx 3.2 \text{ ore}$$

Algoritmi e complessità

$f(n)$	Dimensione del problema					
	$n = 10$	$n = 20$	$n = 40$	$n = 50$	$n = 100$	$n = 1.000$
$\log_2 n$	$3 \cdot 10^{-8}$ sec.	$4 \cdot 10^{-8}$ sec.	$5 \cdot 10^{-8}$ sec.	$5 \cdot 10^{-8}$ sec.	$6 \cdot 10^{-8}$ sec.	10^{-7} sec.
n	10^{-7} sec.	$2 \cdot 10^{-7}$ sec.	$4 \cdot 10^{-7}$ sec.	$5 \cdot 10^{-7}$ sec.	10^{-6} sec.	10^{-5} sec.
$n \log_2 n$	$3 \cdot 10^{-7}$ sec.	$8 \cdot 10^{-7}$ sec.	$2 \cdot 10^{-6}$ sec.	$2 \cdot 10^{-6}$ sec.	$6 \cdot 10^{-6}$ sec.	0,0001 sec.
n^2	10^{-6} sec.	$4 \cdot 10^{-6}$ sec.	0,000016 sec.	0,000025 sec.	0,0001 sec.	0,01 sec.
n^3	0,00001 sec.	0,00008 sec.	0,00064 sec.	0,00125 sec.	0,01 sec.	10 sec.
n^4	0,0001 sec.	0,0016 sec.	0,0256 sec.	0,0625 sec.	1 sec.	16 min.
2^n	0,00001 sec.	0,01 sec.	3 ore	130 gg	miliardi di anni	...
$n!$	0,036 sec.	7 secoli	miliardi di anni	...	...	...

TABELLA 1: Tempo di esecuzione di algoritmi con funzioni di complessità differenti al variare della dimensione del problema su un calcolatore che esegue cento milioni di operazioni al secondo

In generale quindi il tempo di esecuzione di un programma dipende da:

- L'algoritmo scelto per risolvere il problema
- La dimensione dell'input
- La velocità della macchina

Algoritmi e complessità

- Nella teoria della complessità l'analisi della efficienza di un programma è basata sull'ipotesi che il costo di ogni istruzione semplice (assegnamento, operazioni aritmetiche, logiche, lettura/scrittura) e di ogni operazione di confronto sia pari ad un'unità di costo indipendentemente dal linguaggio e dal sistema usato.
- Si adotta un **modello di costo** in cui si contano le operazioni eseguite o alcune operazioni chiave o preminenti ammettendo che il tempo complessivo di esecuzione sia proporzionale al numero di tali operazioni.
- Per tener conto del numero di dati con cui si esegue il programma assumiamo che la dimensione dell'input rappresenti l'argomento della funzione che esprime il costo di esecuzione di un programma.

Algoritmi e complessità

- Fissata quindi la dimensione n dei dati in ingresso l'obiettivo è esprimere la complessità in tempo come funzione di n
- Spesso ci si limita a studiare il comportamento di tale funzione al crescere di n (complessità asintotica) considerando così i soli termini prevalenti e tralasciando a volte anche le costanti moltiplicative.
- Ciò che interessa è l'andamento, o ordine di grandezza, del tempo di esecuzione
- Si considerano di solito tre differenti tipi di complessità: complessità nel caso medio, nel caso ottimo e nel caso peggiore.
- Complessità media: valore della complessità di un algoritmo, mediato su tutte le possibili occorrenze iniziali dei dati (si usa spesso la teoria della probabilità, il calcolo è spesso difficile).

Algoritmi e complessità

- Complessità nel caso ottimo (ovvero nella nella migliore situazione possibile): si ottiene considerando, a parità di dimensione dei dati, la configurazione che dà luogo al minimo tempo di esecuzione.

- complessità nel caso peggiore: si intende la complessità relativa a quella particolare occorrenza iniziale dei dati per cui l'algoritmo ha il comportamento peggiore. E' particolarmente importante perché fornisce un limite superiore al tempo di esecuzione ("non può andare peggio di così")

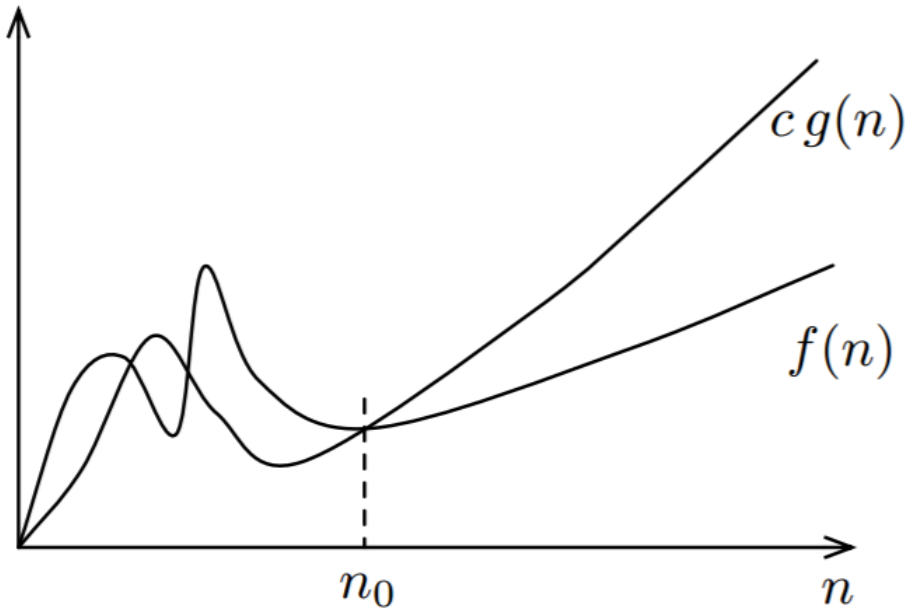
Algoritmi e complessità

- Esempio: un algoritmo di ricerca, nel caso peggiore, richiede un tempo proporzionale al numero degli elementi ***n:*** c***n***, dove c è una costante
- Nell'analisi di complessità non interessa conoscere il valore delle costanti

- Riassumendo: la complessità di un algoritmo è funzione della dimensione dei dati n, ovvero della mole dei dati del problema da risolvere. Determinare la complessità in tempo di un algoritmo significa determinare l'andamento asintotico di una funzione di complessità f(n) che fornisca la misura del tempo al variare della dimensione dei dati, n. Le funzioni di complessità sono caratterizzate da due proprietà:
  - Assumono solo valori positivi
  - Sono crescenti rispetto alla dimensione dei dati.

Algoritmi e complessità

- *g(n)* è un limite superiore asintotico per una funzione *f(n)* se



$$O(g(n)) = \{\, f(n) \mid \exists c > 0 \exists n_0 > 0 \quad \forall n \geq n_0 . 0 \leq f(n) \leq c\,g(n) \,\}$$

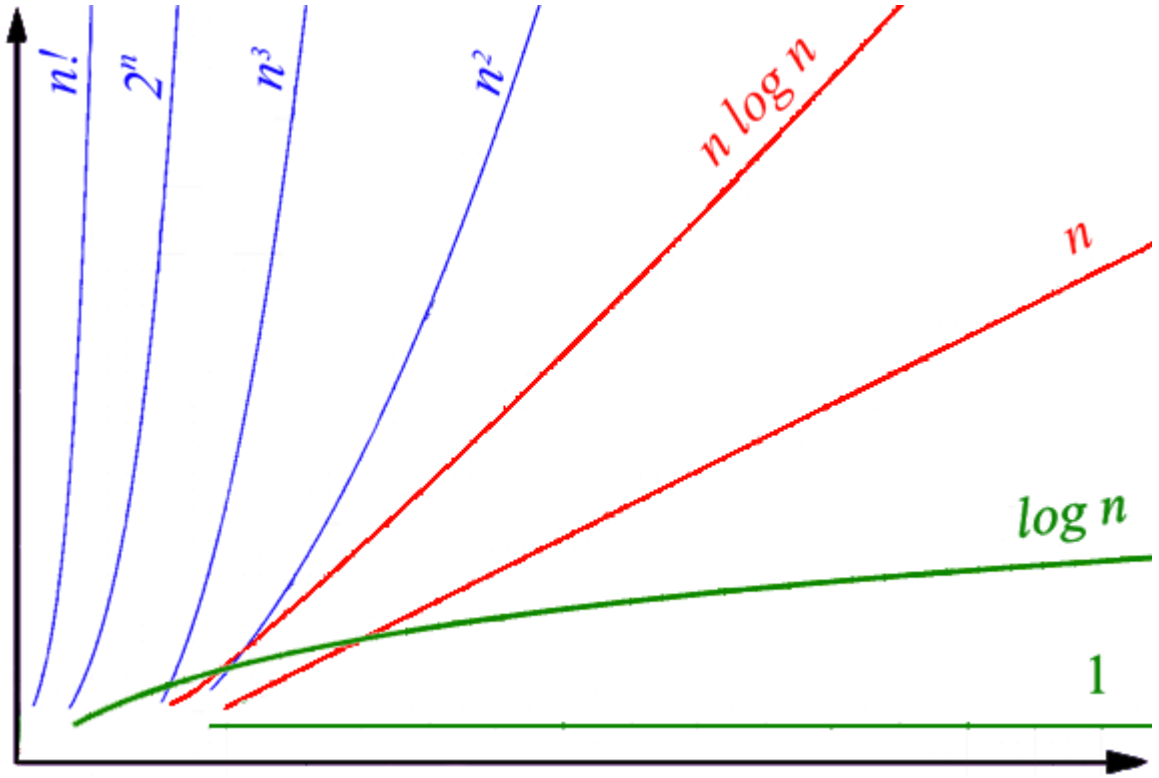
Algoritmi e complessità

- Classi di complessità

	Ordine	Esempio
$O(1)$	costante	Determinare se un numero è pari
$O(\log n)$	logaritmico	Ricerca di un elemento in un array ordinato
$O(n)$	lineare	Ricerca di un elemento in un array disordinato
$O(n \log n)$	pseudolineare	Ordinamento mediante Merge Sort
$O(n^2)$	quadratico	Ordinamento mediante Bubble Sort
$O(n^3)$	cubico	Prodotto di due matrixi $n \times n$ con definizione
$O(c^n)$	esponenziale, base $c > 1$	
$O(n!)$	fattoriale	Determinante con espansione dei minori
$O(n^n)$	esponenziale, base n	

Algoritmi e complessità

- Classi di complessità



Algoritmi e complessità

- Due programmi i cui costi sono espressi rispettivamente dalle funzioni

$f_1(n)=(3 + n)$ e $f_2(n)=(100n + 3027)$

sono caratterizzati dalla stessa complessità asintotica (lineare).

- Due programmi i cui costi sono espressi rispettivamente dalle funzioni

$f_1(n)= 3n^2 – 4n + 2$ $f_2(n)=2n + 3$

$f_1(n)$ ha complessità quadratica

$f_2(n)$ ha complessità lineare

Algoritmi e complessità

- Dati due problemi A e B, supponiamo che:
- esista un algoritmo per risolvere A in tempo O(n) nel caso peggiore
- il miglior algoritmo per B sia $O(2^n)$ nel caso peggiore
- Allora, possiamo concludere che B è un problema "più difficile" di A

- un problema con complessità polinomiale nel caso peggiore($O(n^d)$ per qualche d è considerato "trattabile"
- un problema con complessità almeno esponenziale nel caso peggiore ($O(c^n)$ per qualche c è considerato "intrattabile"