estensioni del C++
capitolo 9 del testo

1. overloading resolution
2. compilazione separata
3. costanti e puntatori a costanti
4. cast
5. break e continue
6. precisazioni sui cicli for

1. overloading o sovraccaricamento

```
void print(int);

void print(const char*);

void print(double);

void print(long int);

void print(char);


char c; float f; short int i;

print(c); print(f); print(i); print("a");
```

le conversioni hanno un costo:

1) perfetta uguaglianza
2) promozioni
3) contrario delle promozioni
4) conversione definita dall'utente

Esempio

una invocazione f(e1,e2) e 2 funzioni:

f(T1 x,T2 y) (e1,e2) → (T1,T2) → conversioni (c1,c2)

f(H1 z,H2 w) (e1,e2) → (H1,H2) → conversioni (h1,h2)

per scegliere quale f vada invocata, confrontiamo (c1,c2) e (h1,h2)

(c1,c2) è meglio di (h1,h2) se

- c1 <= h1 e c2 <= h2
- e inoltre o c1 < h1 o c2 < h2 o entrambe

se né (c1,c2) è meglio di (h1,h2), né viceversa

allora ambiguità ➔ERRORE viene segnalato dal compilatore

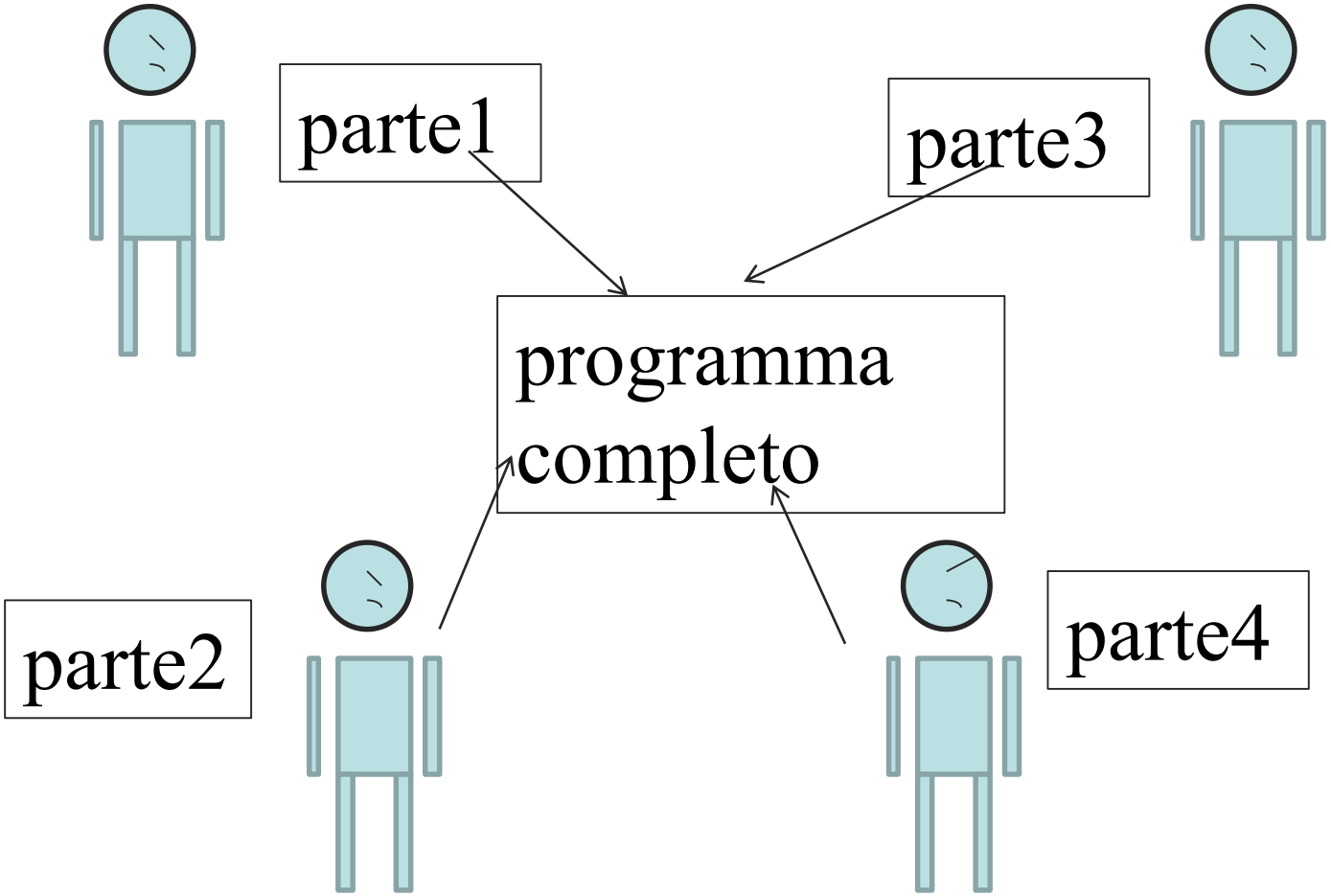
algoritmo di overloading resolution

viene applicato dal compilatore

per scegliere la «migliore» funzione per ogni invocazione

per esempio per operator<<

2. compilazione separata



la parte x (x=1/2/3/4) usa cose definite nelle altre parti.
Quali cose?

- variabili globali

- tipi ad hoc

- funzioni

il programmatore x non può aspettare di avere tutte le 4 parti per compilarle

deve essere in grado di compilare la sua parte “da sola” altrimenti come trova gli errori ?

come si fa la compilazione separata

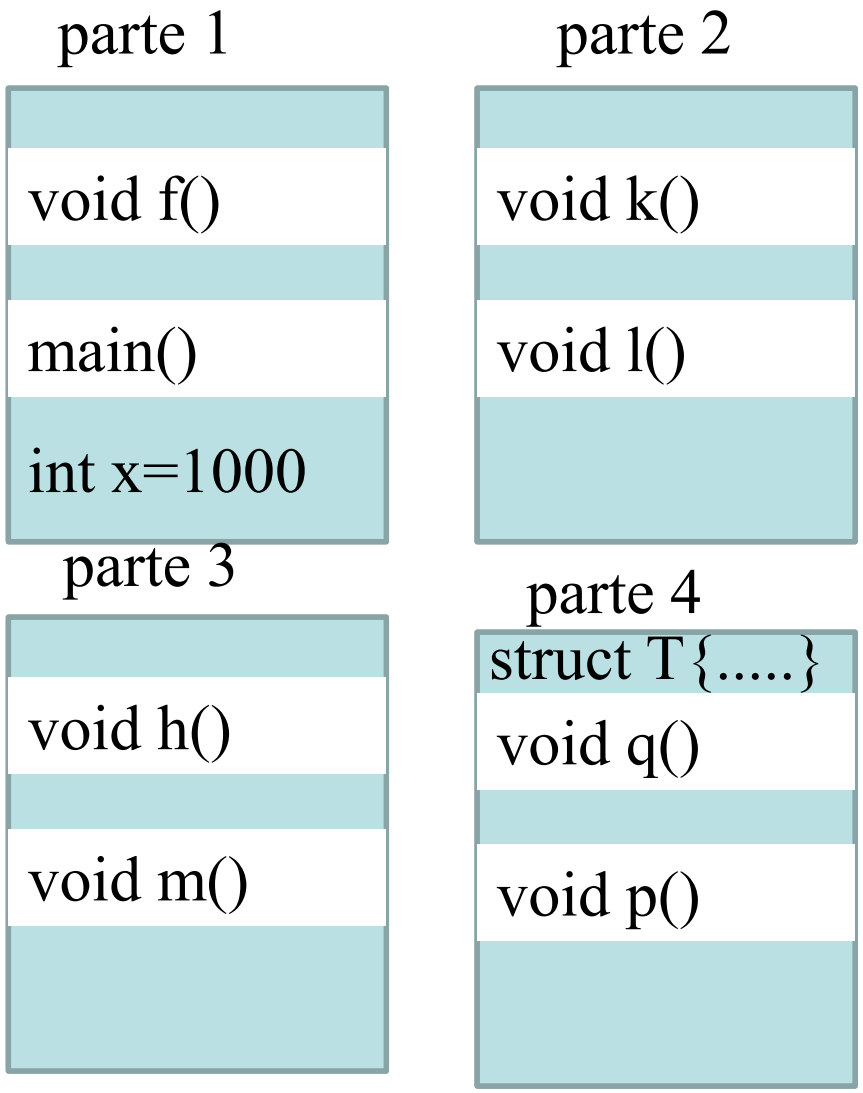
g++ -c parte_x.cpp ➔ parte_x.o

g++ -c parte_y.cpp ➔ parte_y.o

compilazione e link delle diverse parti

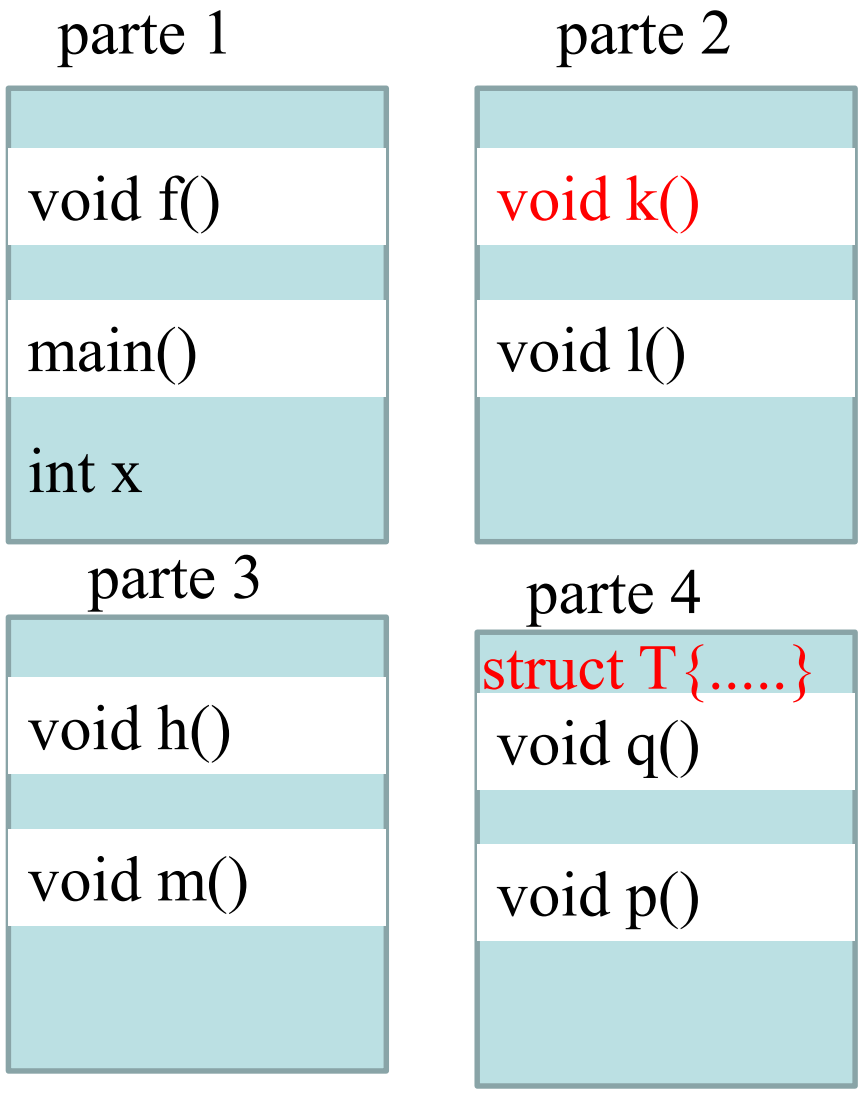
g++ parte_x.o parte_y.o

Makefile visti su eclipse



le zone azzurre vanno viste come un unico blocco

il blocco globale



supponiamo che la parte 1 usi k() della parte 2 e il tipo T della parte 4

-deve includere il prototipo di k()

-deve includere la definizione completa di T esattamente

sistematizzare la cosa: invece di un unico file .cpp servono 2 file

parte_x.h e parte_x.cpp

- parte_x.h = file header con le dichiarazioni
- parte_x.cpp= file con le definizioni

il file header (.h) contiene le dichiarazioni di tipo, i prototipi delle funzioni definite nella parte x e che servono nelle altre

parte_x.h = interfaccia di parte_x

parte_1.cpp inizia con

#include “parte_1.h”
#include “parte_2.h” per k()
#include “parte_4.h” per struct T

nel file parte_x.h: dove x=1/2/3/4

#ifndef PARTE_X_H
#define PARTE_X_H

definizioni di parte_X

#END_IF

per le variabili globali non funziona bene

parte_1.h
int x=1000;

parte_1.h
int x=1000;

parte_11.h
extern int x;

void f()

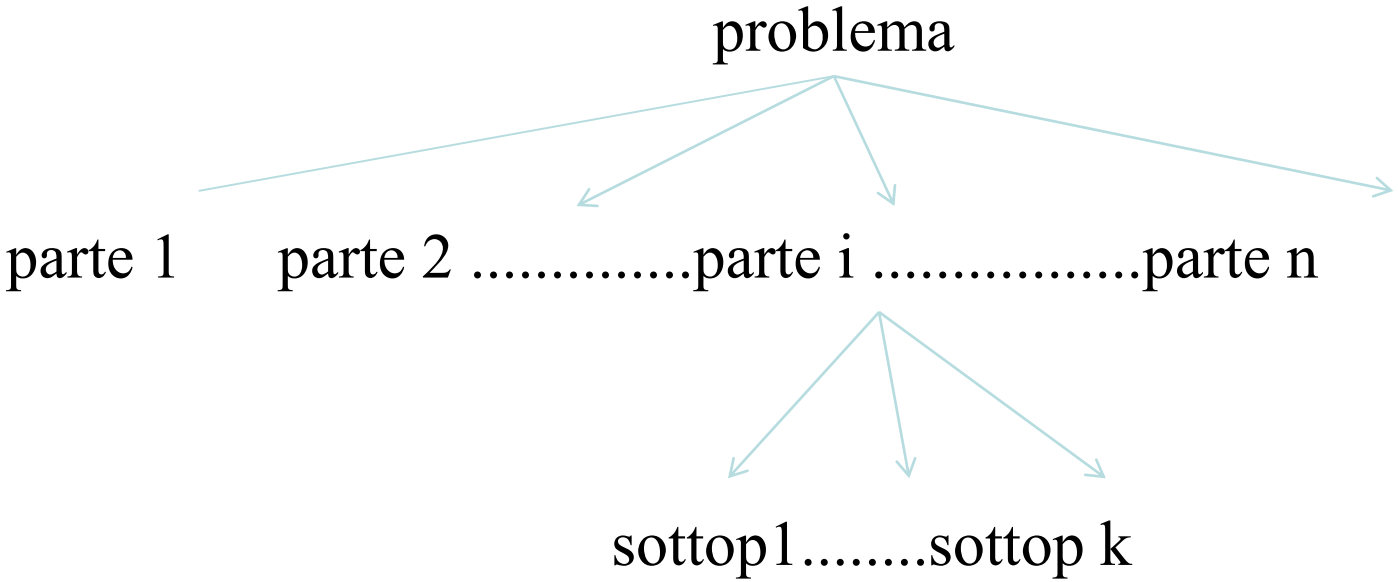
main()

void l()

void m()

ERRORE doppia definizione di x

organizzazione logica e fisica dei programmi:



facili interferenze di nomi ➔ namespace permette di “isolare” le diverse parti

come si costruisce un namespace

namespace pippo{prototipi delle funzioni e tipi} è un .h

le definizioni delle funzioni di pippo sono su altri file == .cpp

necessario dichiarare che si intende usare un namespace

tutto disponibile: using namespace pippo;

altrimenti: using pippo::f(.....); ➔ f(..) disponibile

oppure: pippo::f(...)

using namespace std; ➔ molto grossolano

3. Costanti e puntatori a costanti

```
const int x=2, *p=&x; // OK
int *q=p;      // NO
int y=3;
p=&y;          // OK

(*p)++; // NO
  y++;   // SI
```

CONSTANT POINTER

VS

POINTER TO CONSTANT

aggiungere const è OK, toglierlo NO

```
int x=10, * const y=& x;  // y punta a x ed è costante
```

quindi

```
*y++; // OK
y++ ; // NO
```

const serve soprattutto per proteggere parametri passati alle funzioni

void F(const double & y,....) *// F non può cambiare y*
void F1(const int A[],...) *// F non può cambiare A*

int X[100]; double y=3.14;

F(y); *// OK anche se y non è const*

F1(X,...) *// OK anche se X non è const*

F(3.14) ; *// OK viene costruito un double temporaneo*

4. cast

operazioni C e C++ per convertire un valore di un tipo in un valore «equivalente» di un altro tipo

il C ha un solo cast: se T indica un qualsiasi tipo

sintassi: T x = (T) exp;
si calcola il valore di exp lo si converte in un valore del tipo T e lo si assegna a x. Una volta veniva fatto e basta (se il compilatore "sa" come farlo)

ma il C++ è più prudente:

```
int x=10, *y=&x;
int z = (int) y;  // da errore con compilazione normale
```

```
g++ -fpermissive prof.cpp   // da solo warning
```

ma il programmatore sa veramente che pericolo comporta la conversione che sta chiedendo?

- Il C assume che lo sappia

- il C++ ha altro approccio: meglio avere diversi cast in modo che il programmatore dichiari esplicitamente quale conversione richiede.
  In questo modo, se la conversione dichiarata non fosse appropriata per i tipi coinvolti, ci sarebbe un messaggio d'errore

Il C++ ha 4 operazioni di cast:

- static_cast<T>(exp);
- const_cast<T*>(const_T*);
- reinterpret_cast<T1*> (T*)

- dynamic_cast<T1>(T); *//per gli oggetti, lo vedrete in P2*

static_cast per promozioni e il loro opposto

int → double ma anche double →int o double → char
solo int → enum non va

const_cast toglie il const ai puntatori

```
const int x=10, *p=&x;
int  *q=const_cast<int*>(p);
(*q)++;
cout<<x<<*q<<*p;   // cosa stampa?
```

reinterpret_cast Riguarda i puntatori…. è il più PERICOLOSO !!

double* → int*

int* → double*

esempio

```
int x=20, *y=&x;
double * z= reinterpret_cast<double*> (y)
```

z punta a 8 byte e i primi 4 contengono l'R-valore di x, mentre i secondi 4 non sappiamo cosa contengano. Probabile errore.

ATTENZIONE: Non viene fatta alcuna conversione del valore puntato !

5. break e continue

sono dei ricordi di uno stile di programmazione di altri tempi

meglio non usarli (a parte casi eccezionali)

complicano le prove di correttezza

```
int x=0;
for(int i=0; i<10; i++)
{
   cout<<x<<‘ ‘<<i<<endl;
   break;
   x++;
}
```

```
int x=0;
for(int i=0; i<10; i++)
{
   cout<<x<<‘ ‘<<i<<endl;
   continue;
   x++;
}
```

```
for(int j=0; j<10; j++)
{
   int x=0;
   for(int i=0; i<10; i++)
   {
      cout<< x <<‘ ‘ << i << ‘ ‘ << j <<endl;
      break;
      x++;
   }
}
```

si esce da 1 ciclo

switch

```
char X='a';
……
switch(X)
{
   case 'a': X='A'; break;
   case 'b': X='B'; break;
   ……;
   default: ……;

}
```

6. qualche osservazione sui for

```
int x=0, y=10;
for(int i=x, j=y; i<10 && j>0; i++, j--)
{
        x=x+10;
        y=y-10;
        cout << i << j << x << y<<endl;


}
```

l'inizializzazione è fatta solo la prima volta

i è dichiarata in diversi blocchi annidati

```
int i=100;
  for(int i=0; i <10; i++)
{
      cout<< i<< endl;
      for(int i=10; i <20 ; i++) cout << i<<endl;

      i++;
      cout << i << endl;
}
 cout << i<< endl;
```