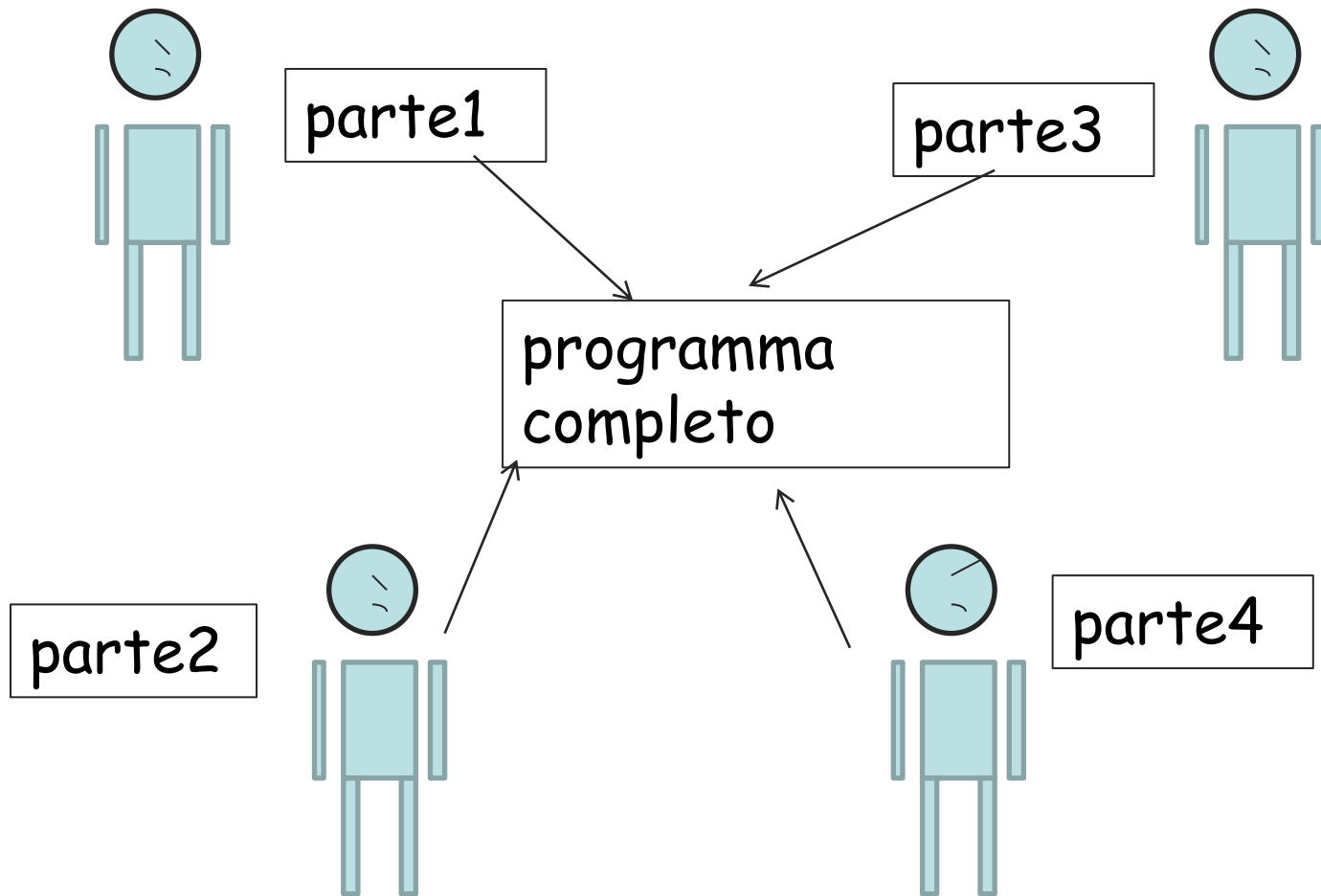


capitolo 9

1. compilazione separata
2. eccezioni
3. overloading
4. costanti e puntatori a costanti
5. cast

1. compilazione separata (9.7)

programmi scritti su più file



la parte i usa cose definite nelle altre parti. Quali cose?

- variabili globali

- tipi ad hoc

- funzioni

il programmatore i 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_mia.cpp → parte_mia.o
```

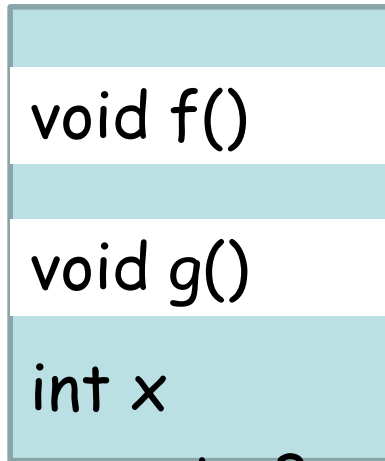
```
g++ -c parte_tua.cpp → parte_tua.o
```

compilazione e link delle diverse parti

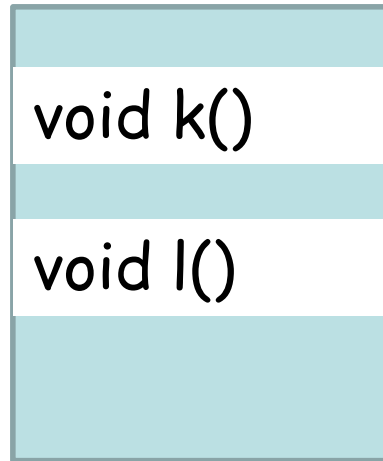
```
g++ parte_mia.o parte_tua.o
```

Makefile

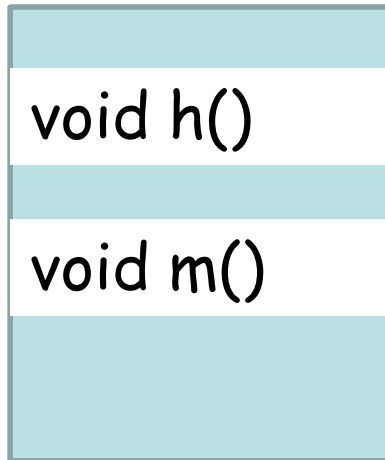
parte 1



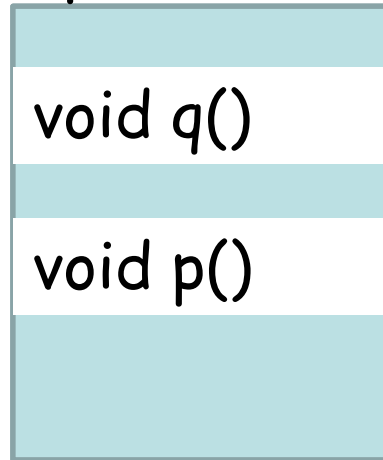
parte 2



parte 3



parte 4



le zone azzurre
formano un
unico blocco

il blocco globale

parte 1

```
void f()  
void g()  
int x
```

parte 2

```
void k()  
void l()
```

parte 3

```
void h()  
void m()
```

parte 4

```
struct T{.....}  
void q()  
void p()
```

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 di
T esattamente

sistematizzare l'inclusione:

file header e file delle definizioni

parte_1.cpp e parte_1.h

.cpp contiene le definizioni

.h le dichiarazioni di tipo, i prototipi
delle funzioni esportate

= interfaccia di parte_1

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

`#include "parte_1.h" #include "parte_1.h"`

`void f()`

`void g()`

`void l()`

`void m()`

`void h()`

`void i()`

`void n()`

`void w()`

se mettessimo
`int x;`
in `parte_1.h`
che è incluso in
parte 2

ERRORE doppia
definizione di `x`

2 zone di memoria per `x`!

int x

void f()

void g()

extern int x;

void l()

void m()

void h()

void i()

struct T{.....}

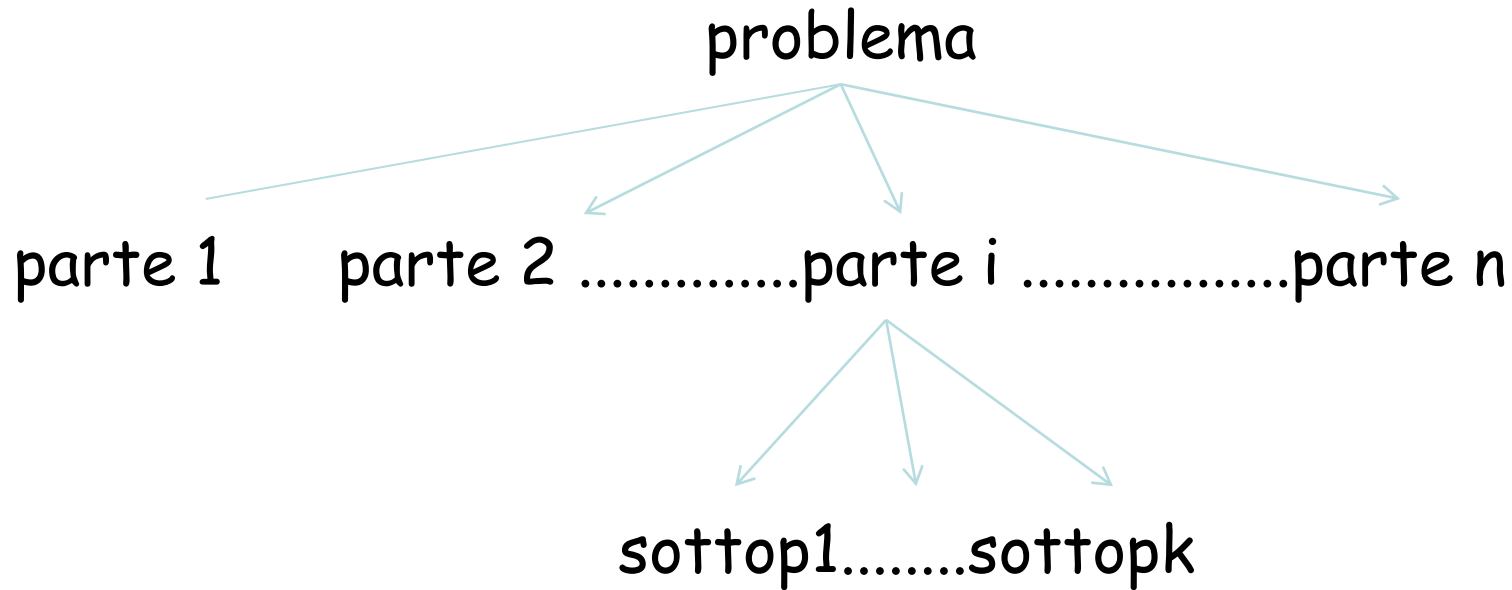
void n()

void w()

si deve includere
extern int x;

e quindi avere 2
file .h

organizzazione logica dei programmi:



facile interferenze di nomi → namespace permette di
"isolare" le diverse parti

come si costruisce un namespace

namespace pippo{prototipi e tipi} == .h

definizioni su altri file == .cpp

necessarie comunicazioni

using namespace pippo; → tutto disponibile

using pippo::f(.....); → f(..) disponibile

pippo::f(...)

2. eccezioni (testo 9.5)

3 istruzioni del C++:

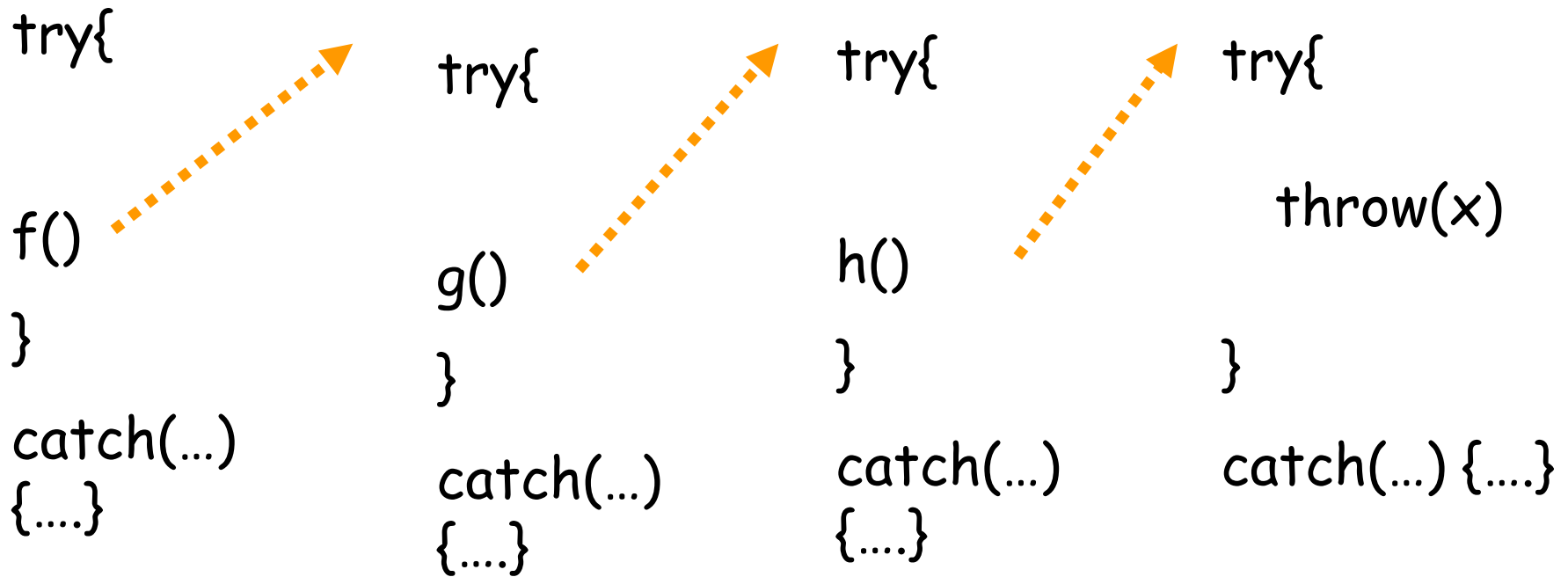
- **try** : evidenzia un blocco in cui possono essere sollevate eccezioni
- **throw(val: Tipo)** : solleva l'eccezione
- **catch(Tipo x)**: la "afferra" e la gestisce

```
main()
{try
{int x;
cin >> x;
f(x);
}
catch(int x)
{switch(x)
{
case 1: cout << "errore 1"<<endl; break;
case 2: cout << "errore 2"<<endl; break;
case 3: cout << "nessun errore"<< endl;
}
}}
}
```

concordanza
di tipo



```
void f(int x)
{
if(x==1)
throw(1);
if(x==2)
throw(2);
throw(3);
}
```



i catch vengono esaminati in ordine e viene preso il primo con parametro di tipo = a quello sollevato dal throw

try e catch possono essere anche dentro
un ciclo

quindi dopo il catch il ciclo continua


```
main()
{
    int c=0; bool controllato=false;
    char A[]="pip4oplu5srom1wst3a12er";
    int s=sizeof(A);
    while(c<10 && !controllato)
    {
        try {
            F(A,s-1);
            cout<<"stringa ok="<<A<<endl;
            controllato=true;
        }
        catch(int i)
        {
            cout<<"trovato carattere strano in posizione="<<i <<endl;
            cout<<"errore n."<<c<<endl;
            A[i]='x';
            cout<<A<<endl;
            c++;
        }
    }
}
```

```
void F(char *A, int dim)
{
    for(int i=0; i<dim; i++)
        if(A[i]<'a' || A[i]>'z')
            throw(i);
}
```

una catch può contenere una throw
che verrà gestita da un'altra catch

```

main()
{try
{....
while(!controllato)
{try { F(A, s - 1);
      cout<<"stringa ok="<<A<<endl;
      controllato=true;
    }
    catch(int i)
    {
      cout<<"trovato carattere strano in posizione="<<i<<endl;
      cout<<"errore n."<<c<<endl;
      A[i]='x';
      c++;
      cout<<A<<endl;
      if(c==4)
        throw(c);
    }
  }
}
catch(int i) {cout<<"trovati "<< i <<" errori: termino"<<endl;}
}

```

```

void F(char *A, int dim)
{
  for(int i=0; i<dim; i++)
    if(A[i]<'a' || A[i]>'z')
      throw(i);
}

```

catch che lancia una throw

tipi struttura lanciati dalle eccezioni

```
struct err{ string mess; int pos;  
    err(string S, int i) // costruttore  
    {mess= S; pos=i;}  
    err(){}  
};
```

```
main()  
try {  
    char A[]="pip4o";  
    F(A,5);  
    cout<<A<<endl;
```

```
}  
catch(err x)  
{cout<<"trovato "<<x.mess<<" in pos=" <<x.pos<<endl;}
```

```
void F(char *A, int dim)  
{  
    for(int i=0; i<dim; i++)  
        if(A[i]<'a' || A[i]>'z')  
            { err E("errore tal dei tali",i);  
              throw(E);  
            }  
        else  
            A[i]=.....;  
}
```

3. overloading o sovraccaricamento (9.8)

```
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");
```

conversioni hanno un costo:

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

invocazione $f(e1,e2)$ molti candidati:

$f(T1,T2) \leftarrow [c1,c2]$

$f(H1,H2) \leftarrow [h1,h2]$ qual è meglio ??

$[c_1, c_2, \dots, c_k]$ è **meglio** di $[h_1, h_2, \dots, h_k]$ se

- per ogni i in $[1..k]$

$c_i \leq h_i$

- ed esiste j in $[1..k]$ t.c.

$c_j < h_j$

importanti le conversioni automatiche che vengono fatte al momento dell'invocazione delle funzioni con passaggio dei parametri per valore

quelle che si possono fare con `static_cast`

non sono ammesse conversioni che coinvolgono puntatori

4. Costanti e puntatori a costanti (9.1)

`const int x=2, *p=&x; // OK`

`int *q=&x; // NO`

`int y=3;`

`p=&y; // OK`

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

`y++; // SI`

aggiungere `const` OK, toglierlo NO

const serve **soprattutto** per proteggere parametri passati alle funzioni

```
void F(const int A[],...) // F non può  
cambiare A
```

```
int X[100];
```

```
....
```

```
F(X,...) // OK anche se X non è const
```

void F(const double & y,...) // F non può
cambiare y

double y=3.14;
F(y,...) //OK

anche
F(3.14,...) // OK, ma solo con const nel
parametro formale

5. cast (9.4)

operazioni per convertire il tipo di un valore

cast alla C: $T =$ Tipo destinazione

$T\ x = (T)\ exp;$

si calcola il valore di exp lo si converte in un valore del tipo T e lo si assegna a x .

Viene fatto e basta (se il compilatore "sa" come farlo)

ma il programmatore sa veramente
cosa sta chiedendo ?
il C lo assume

ma C++ ha altro approccio: meglio
avere diversi cast in modo che il
programmatore dichiari esplicitamente
quale conversione pensa di richiedere

se non fosse come pensa, 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 il C++

`static_cast` per promozioni e il loro opposto

`int` \rightarrow `double` ma anche `double` \rightarrow `int` o

`double` \rightarrow `char`

solo `int` \rightarrow `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; // attenzione !!
```

reinterpret_cast PERICOLO !!

double * \rightarrow int

int* \rightarrow double*

dynamic_cast<T>(exp)

test dinamico se tipo di exp compatibile
con T allora true, altrimenti false