

capitolo 9

strutture di controllo

break

continue

switch

```
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++;  
    }  
}
```

```
enum colore{bianco, giallo, rosso, blu,...} X=giallo;
```

.....

```
switch(X)
```

```
{
```

```
    case bianco: X=giallo; break;
```

```
    case giallo: X=rosso; break;
```

```
    case rosso: case blu: X=bianco; break;
```

```
    default: X=blu;
```

```
}
```

con char

```
char X='a';
```

```
.....
```

```
switch(X)
```

```
{
```

```
    case 'a': ..... break;
```

```
    case 'b': ..... break;
```

```
    .....;
```

```
    default: .....;
```

```
}
```

- tipi strutture
- { collezione di campi con tipi diversi }
- {anche operatori per integrare i nuovi tipi nel linguaggio}
- campi + funzioni per manipolarli (metodi)
- sicurezza
- classi e oggetti

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

cast (9.4)

operazioni per cambiare il tipo di un valore e/o di una variabile

cast alla C: $T_d =$ Tipo destinazione

$T_d \ x = (T_d) \text{exp};$

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

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

ma il programmatore sa veramente
cosa sta chiedendo ?
C....non importa

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, messaggio
d'errore

Il C++ ha 4 operazioni di cast:

- `static_cast<T_d>(exp);`
- `const_cast<T_d>(exp_const);`
- `reinterpret_cast<T_d_punt>(exp_punt)`
- `dynamic_cast<T_d>(exp);`

static_cast

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 PERICOLO !!

double * → int

int* → double*

dynamic_cast<T_dest>(exp)

test dinamico se tipo di exp compatibile
con T_dest allora ok, altrimenti no

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

parametri (passati per valore) con valori di default

```
void f(int x=0, int y, int z=0) // NO
```

valori di default devono essere accostati a destra

```
void f(int x, int y=0, int z=0) // OK
```


f(5,3)

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");
```

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.

$c_j < h_j$

eccezioni

qualche esempio

trattamento delle eccezioni (9.5)

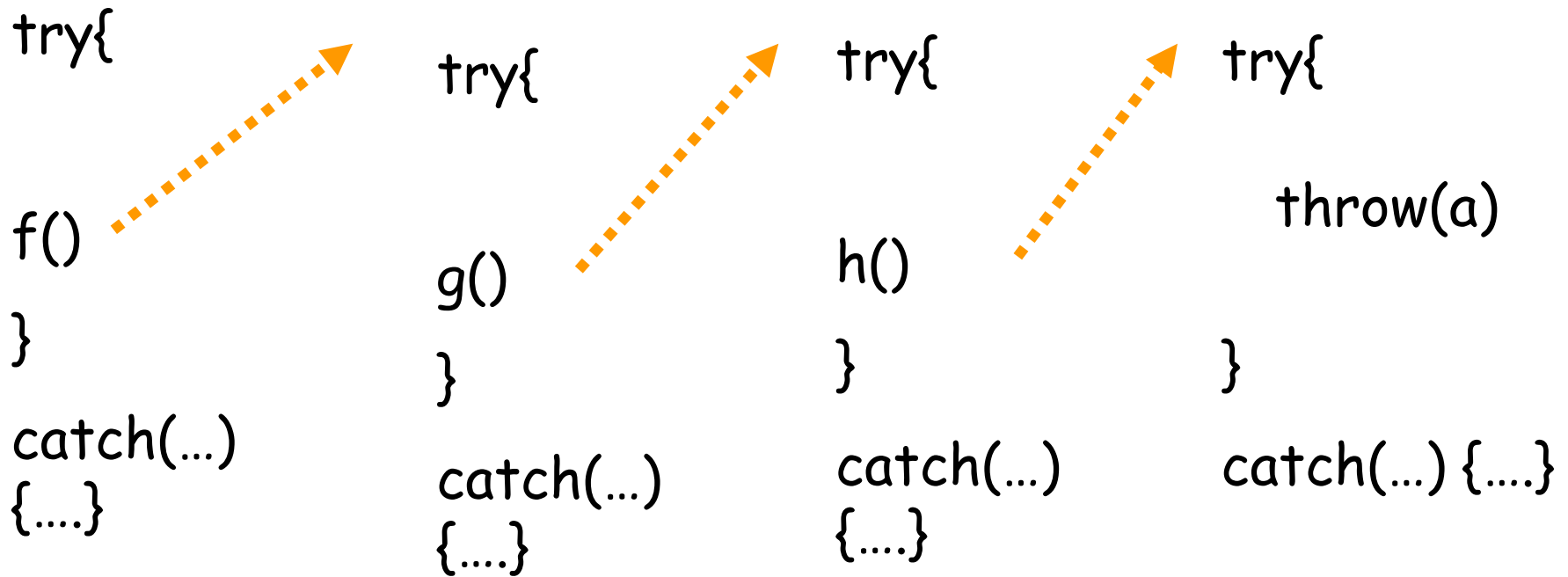
3 costrutti:

- **try** : avverte che nel blocco possono essere sollevate eccezioni
- **throw(...)** : solleva l'eccezione
- **catch(...)** : 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 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);
}

```

un catch può lanciare una throw

tipi struttura come errori

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

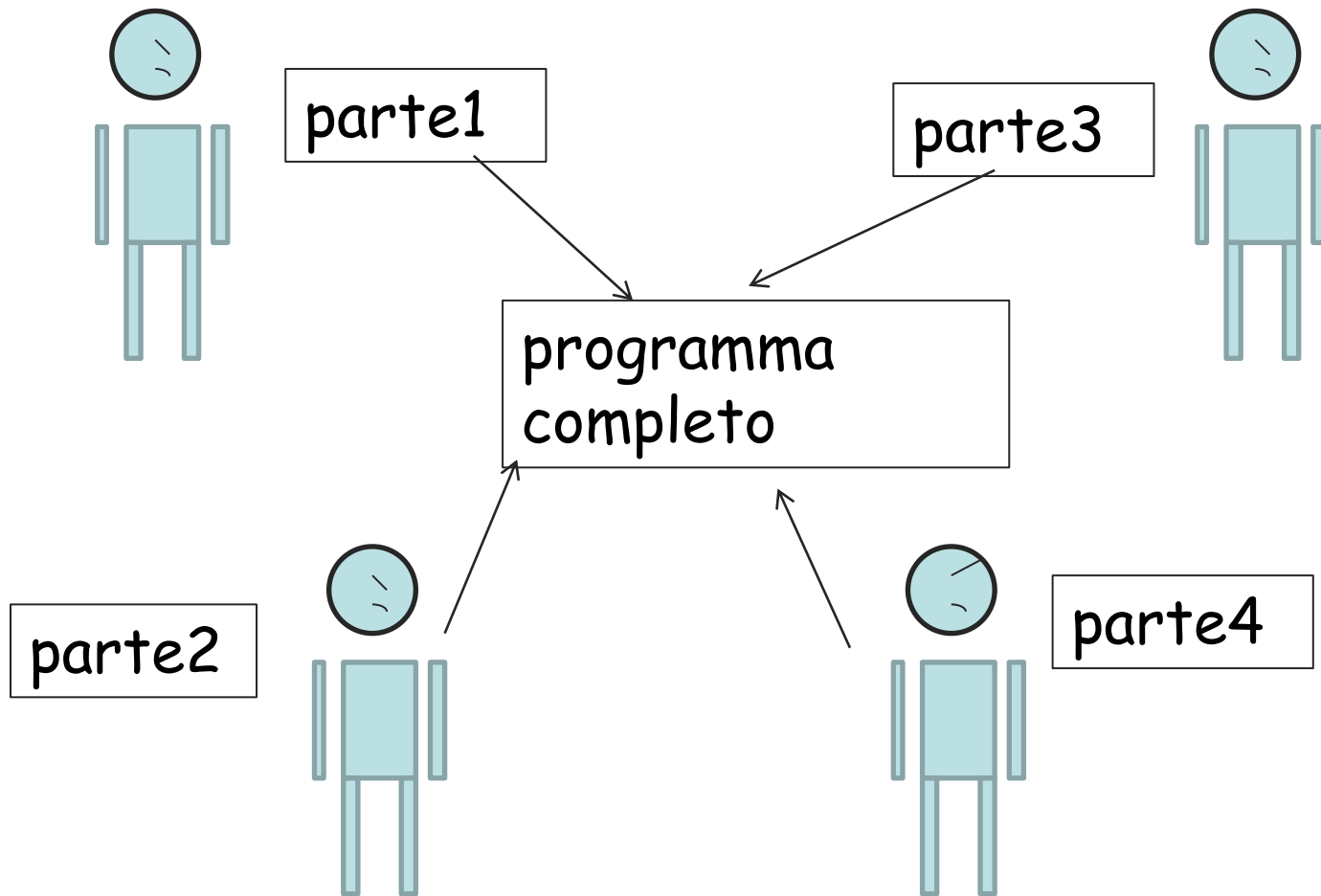
```
main()  
try {  
    char A[]="pip4o";  
    F(A,5);  
    cout<<A<<endl;
```

```
}  
catch(err x)  
{cout<<"dopo "<<x.mess<<" trovato errore 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 tali",i);  
              throw(E);  
            }  
        else  
            A[i]=.....;  
}
```

compilazione separata (9.7)

programmi scritti su più file



ogni programmatore deve però essere in grado di compilare la sua parte
altrimenti come trova gli errori ?

per farlo deve "sapere" qualcosa delle altre parti

cosa si può mettere in ciascuna parte che debba essere "visibile" anche nelle altre parti?

- variabili/costanti globali
- tipi ad hoc
- funzioni



```
void f()
```

```
void g()
```

```
int x
```

```
void k()
```

```
void l()
```

```
void h()
```

```
void m()
```

```
void q()
```

```
void p()
```

```
struct T{.....}
```

le zone azzurre
formano un
unico blocco

il blocco globale

funzioni e tipi ad hoc:
sono definiti nel blocco globale

basta ripeterli dove servono

funzioni: ripetere prototipo

tipi: ripetere la dichiarazione esatta

variabili e costanti (meno importante)
più complicato

se ripetessimo `int x` in tutti i file dove
serve la globale `x` il blocco globale
violerebbe la regola che in un blocco ci
può essere solo 1 dichiarazione di un
certo nome
una sola dichiarazione e poi solo richiami

void f()
void g()
int x

external int x;

void l()
void m()

void h()
void i()

void n()
void w()
struct T{.....}

richiami
preceduti da
external

variabili globali sono inizializzate

sistematizzare l'esportazione:

file header

pippo.cpp e pippo.h

.cpp contiene le definizioni

.h i tipi ad hoc e i prototipi e le
dichiarazioni globali (interfaccia)

pippo.cpp inizia con

```
#include "pippo.h"  
#include "pluto.h"
```

.....

nel file pippo.h:

```
#ifndef PIPPO_H  
#define PIPPO_H
```

....definizioni....

```
#END_IF
```

organizzazione logica dei programmi:

namespace

non è detto che coincida con lo sviluppo di
parti separate

```
using namespace std;  
using std::cin;
```