30. Funzioni ricorsive

In C++ una funzione può invocare se stessa (funzione ricorsiva) oppure due o più funzioni possono chiamarsi a vicenda (funzioni mutualmente ricorsive).
Nella mia funzione posso chiamare un'altra funzione che a sua volta invoca la mia funzione (questo è il caso delle funzioni mutualmente ricorsive).

Molti problemi sono definibili in termini di funzioni ricorsive (in maniera più "naturale"):

- il fattoriale: 0! def = 1; n! def = n · (n − 1)!;
- pari/dispari: even(n) ⟺ odd(n − 1);
- odd(n) ⟺ even(n − 1); espressioni: somma def = numero; somma def = (somma + somma)
- funzione ricorsiva che calcola la lunghezza di una stringa

Due caratteristiche di una funzione ricorsiva:

- una o più condizioni di terminazione (ad esempio in quella per calcolare il fattoriale, quando il numero è nullo)
- una o più chiamate ricorsive

C'è un intrinseco rischio di produrre sequenze infinite (come con i cicli) e soprattutto non sono efficienti a causa di insidie computazionali: stack finito, macchina che va in swap e non usa solo la memoria riservata al programma, ma anche quella del disco fisso.

! La ricorsione è fortemente collegata al principio di induzione matematico **!**

Alcune note sulla ricorsione
La realizzazione ricorsiva di una funzione può richiedere due funzioni:

- una funzione ausiliaria ricorsiva, con un parametro di ricorsione aggiuntivo (simile a contatore in loop)
- una funzione principale (*wrapper*) che chiama la funzione ricorsiva con un valore base del parametro di ricorsione
- situazione molto frequente nell'uso di array

30.1 Ricorsione vs Iterazione

La chiamata ricorsiva occupa ad ogni iterazione un pezzo dello stack, mentre il ciclo non usa una quantità di memoria maggiore di quella che gli è stata riservata.

Dimensione dello stack dipende dalla profondità di ricorsione: notevole overhead e spreco di memoria, **quando possibile, tipicamente iterazione più efficiente,** passando oggetti "grossi", è indispensabile usare passaggio per riferimento o puntatore (faremo sempre passaggio per valore perché si usano strutture poco complesse).

Attenzione a chiamate identiche in rami diversi! (es. Fibonacci): rischio esplosione combinatoria.

30.2 Riscrittura come iterazione

Molte funzioni ricorsive possono essere riscritte in forma iterativa: le funzioni **tail recursion,** in cui la chiamata ricorsiva è operata come ultimo passo prima del *return* (es: somma, pari/dispari, …).
In generale, quando non comporta una "biforcazione". Esempi: fattoriale, Fibonacci (l'ultima istruzione è la somma delle chiamate ricorsive, non la chiamata ricorsiva), …

g++ -O2 effettua una conversione da *tail-recursive* in iterative. Con questo costrutto da scrivere in fase di compilazione, il programma (assembly) analizza la funzione capisce che è ricorsiva e la riscrive usando i cicli. Non usando *-o2* si rischia di andare in segmentation fault (stack saturo).

Da ricorsione in coda a iterazione (caso generale)

```
type F(int x,...) {                    type F(int x,...) {
if (CasoBase(x,...))                   while (!CasoBase(x,...)){
res = IstrBase(...);                   Istr(...);
else {                    →            x=agg(x,...);
Istr(...);                             }
x=agg(x,...);                          res = IstrBase(...);
res = F(x,...);                        return res;
}                                      }
return res;
}
```

Se lo stack si satura (a causa delle troppe chiamate ricorsive) si va a scrivere l'area di memoria di un altro programma e/o del sistema operativo. Di solito però il programma viene abortito.

g++ -g -O0 zero optimise (col debugging disbabilitare ottimizzazioni, con programma per l'utente mettere ottimizzazioni)
g++ -O2 = g++ -O1 -foptimize -sibling – calls (*-o2* include questi due flag e altri)

Parametro *-s* : il programma abortisce se l'*heap* sovrascrive lo *stack*

Non sempre è agevole riscrivere la ricorsione in modo iterativo. Questo nel caso delle funzioni non *tail-recursive* o con chiamate multiple. In generale, convertire una funzione ricorsiva in iterativa richiede l'uso di uno stack. Si usano strutture addizionali simulando lo stack esplicito di un programma. Si opera con la memoria generica del programma che è unbounded.

Da ricorsione in coda a iterazione (caso void)

```
type F(int x,...) {                    void F(int x,...) {
while (!CasoBase(x,...)){              while (!CasoBase(x,...)){
Istr(...);                    →        Istr(...);
x=agg(x,...);                          x=agg(x,...);
}                                      }
res = IstrBase(...);                   {IstrBase(...);}
return res;                            }
}
```

Funzione ricorsiva come un vero e proprio ciclo

```
if(m!=n){      //condizione di terminazione
 ris+=fprec;
 fprec=ris-fprec;      //istruzioni
 fibo2(m,n-1);         //aggiornamento variabile di controllo
 }
```

31. Qualche esempi

! è importante capire che quando c'è una chiamata ricorsiva è come se andassimo in un'altra funzione, modificassimo i parametri e andiamo in un'altra funzione, modifichiamo i parametri e andiamo ancora. Quando la condizione di terminazione è soddisfatta non si "va più in altre funzioni" e si eseguono le istruzioni standard fino al return (si esegue il return nel caso di funzioni tail recursive). Poi dopo il return si torna nella funzione (subito dopo la chiamata ricorsiva) che ci ha chiamato e ancora indietro, indietro indietro, fino alla prima chiamata che ritorna direttamente al *main*.

! Quando termina la ricorsione si ritorna alla riga della chiamata, la quale assume il valore di ritorno dell'ultima ricorsione, che dà un valore di ritorno a quella prima ecc.

Esempio funzione *fattoriale*

```
int fact(int num){
int res;

if(num==0){
res=1;
} else {
res=num*fact(num-1);
}
return res;
}
```

Esempio funzioni mutualmente ricorsive

```
bool pari(int n) {
  bool ris;
  if (n==0)
    ris = true;
  else
    ris = dispari(n-1);
  return ris;
}
```

```
bool dispari(int n) {
  bool ris;
  if (n==0)
    ris = false;
  else
    ris = pari(n-1);
  return ris;
}
```

Evitare se possibile di usare definizioni statiche e variabili globali

Usare: condizioni specifiche (che permettano di assegnare un valore ad una variabile quando la ricorsione termina), wrapper, passare per parametro una variabile.

Usare un wrapper al posto di →

```
static inv=0;

if(x!=0){
  inv*=10;
  inv+=x%10;
 inverti(x/=10);
 }
return inv;
```

31.2 Qualche nota sulla ricorsione

Abbiamo visto che è abbastanza semplice passare da una funzione iterativa ad una ricorsiva. Al posto dell'eventuale ciclo *for* metto la chiamata ricorsiva (sia in coda, caso più semplice, che all'inizio, in questo caso prima si chiama *n* volte la funzione e poi si fanno le operazioni).

N.B! Il contatore *i* che viene passato per parametro (e inizializzato a zero) può essere passato mediante l'aiuto di un wrapper oppure semplicemente mettendo argomento di default=0.

Inoltre per aumentare il contatore è meglio non fare il *++i* direttamente nella chiamata a funzione, ma farlo prima. Si può altrimenti fare *funz(a,b,i+1);*

La chiamata ricorsiva può essere fatta anche prima di fare tutte le operazioni. Questo di solito si fa se si necessita di una variabile aggiuntiva che può essere dichiarata nell'*else.*

```
int funz(int a, int b){
for(int i=0;i<b;i++){
a = … ;
}
return a;
}
```

```
int funz(int a, int b, int i){

if(i<b){
a = … ;
++i;
a = funz(a,b,i);
} else {}

return a;

}
```

```
int funz(int a, int b, int i){

if(i<b){
i++;
a = funz(a,b,i);
a = a +4 ;
} else {
caso base;
(int var=0);
}

return a;

}
```

Anche nel caso di *for* annidati è possibile arrivare ad una semplice funzione ricorsiva.

Qua è consigliabile sempre mettere la chiamata ricorsiva in coda e dopo ogni ciclo fare una chiamata ricorsiva.

N.B.! È importante ricordarsi di inizializzare a 0 e/o con i valori giusta i vari contatori dei cicli.

```
void funz(int v, int n){
int i,k;
for(k=n-1;k>0;k--){
  for(i=0;i<k;i++){
    istr;
}
}
}
```

```
void funz(int v, int n, int k, int j){

if(k>0){
   if(i<k){
      istr;
  }
  funz(v,n,k,j+1);
}
 funz(v,n,k-1,0);
}
```

Infine è opportuno dare un occhio a una funzione ricorsiva in cui è necessario non solo fare il ciclo ma anche altre funzioni. Un esempio banale è il caso in cui sia necessario passare a una funzione dei puntatori (che puntano a aree di memoria della heap) allocare le aree di memoria in un altra funzione e poi fare il un ciclo all'interno della stessa funzione chiamante. Sarebbe più comodo farlo in due funzioni diverse e creare una funzione solamente per fare il ciclo ma potrebbe subentrare la possibilità di farlo nella stessa funzione. Vediamo questo esempio:

N.B.! Prestare attenzione a passare array dinamici per riferimento, cosa che non viene fatta standard. Ricordarsi inoltre che a seguito di tutte le chiamate ricorsive i contatori vengono

modificati quindi se per qualche motivo abbiamo bisogno di contatori “pieni” anche dopo tante chiamate occorre salvare il valore iniziale in qualche variabile o passare altre variabili per riferimento.

```
void funzrec(…,int casobase,int contatore){

  if(contatore<casobase){
    contatore--;
     //eventuali modifiche ad altre variabili che si passano nella funzione per maneggiare i dati nel corpo dell'if
    funzrec(,contatore,casobase);

    istr del ciclo;

  } else {
  altrafunz(…);
  //eventuali allocazioni dinamiche usate nel ciclo
  }

}
```

Mettendo la chiamata ricorsiva all’inizio chiamiamo la funzione ricorsiva *n* volte (dove *n* è il numero di volte che deve essere eseguito il ciclo), alla fine entriamo nell’*else* dove facciamo la chiamata all’altra funzione (come prima vera e propria istruzione effettiva della nostra funzione, per questo motivo nell’*else* possono essere fatte cose che servono nell’*if,* tra cui allocare memoria e inizializzare variabili).

Nel corpo dell’*if* vengono poi fatte le semplici istruzioni necessarie.

N.B.! Se servono parametri in *altrafunz* che vengono modificati dal contatore valutare la possibilità di passarli per parametro o trovare un modo per tenerli uguali.
Fare sempre molta attenzione al passaggio per riferimento e per valore.