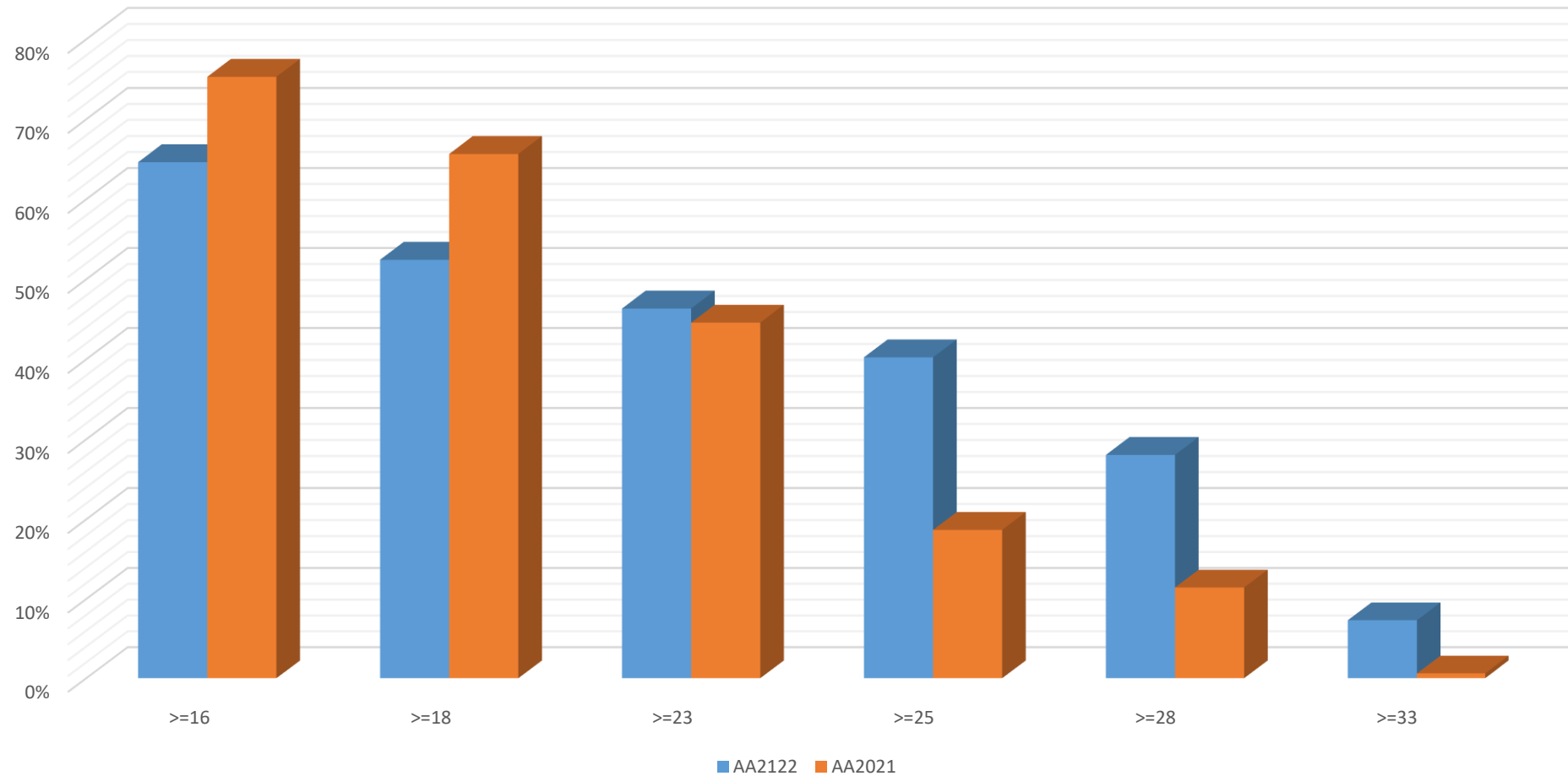


Report prima prova intermedia





Università degli Studi di Cagliari
Corso di Laurea in Ingegneria Biomedica



25 novembre 2021

ELEMENTI DI INFORMATICA

https://www.unica.it/unica/page/it/gianluca_marcialis

A.A. 2021/2022

Docente: **Gian Luca Marcialis**

LINGUAGGIO C
Funzioni e procedure

Sommario

- Introduzione
- Funzioni in C
 - Intestazione
 - Variabili interne ed esterne
 - Regole di visibilità
- Procedure in C

Introduzione

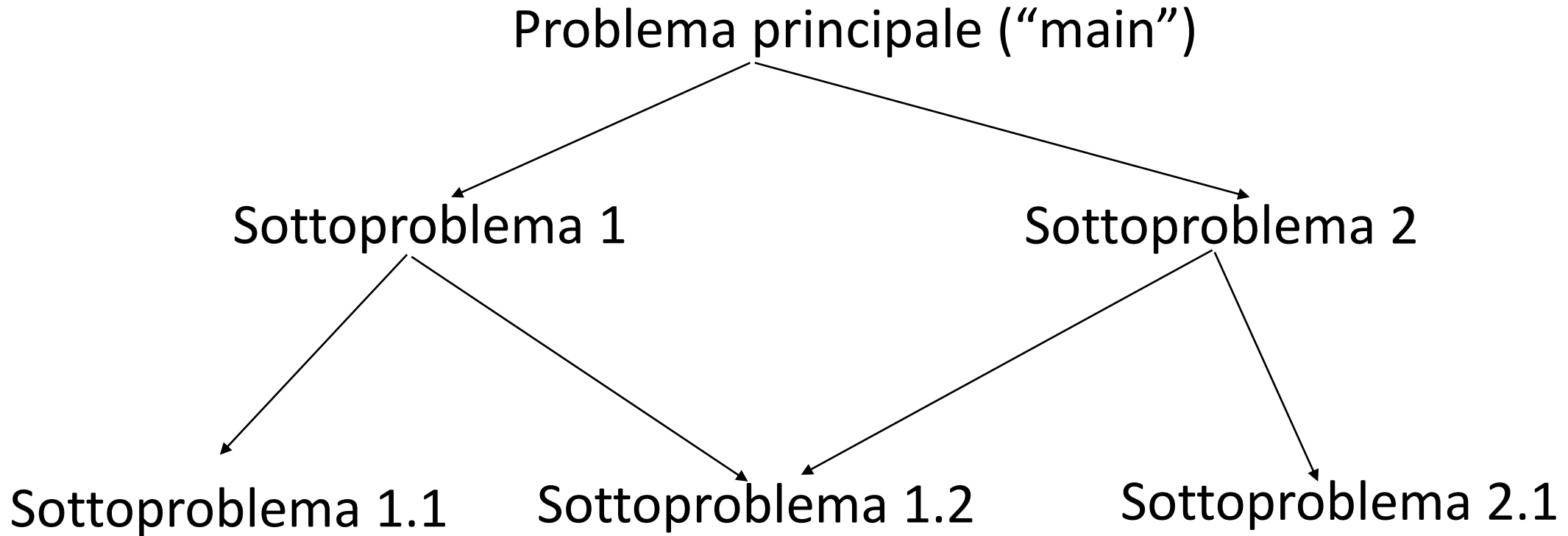
- Finora abbiamo visto le strutture di controllo di base che il C mette a disposizione per alterare il flusso di esecuzione delle istruzioni
- Da questo punto di vista, le funzioni e le procedure costituiscono un metodo particolarmente raffinato per il controllo del flusso di esecuzione
- Possiamo definire una **funzione** come un **raggruppamento di istruzioni volte a risolvere un determinato sottoproblema** entro il problema principale
 - Si pensi per esempio alle “funzioni di libreria” usate in precedenza
- La conoscenza e l’uso di tali strumenti fa la differenza fra un progettista di algoritmi (nonché programmatore) mediocre ed un vero “ingegnere del software”

Approcci di programmazione

- In generale, in problemi complessi possono essere “individuati” sottoproblemi, la cui soluzione ad esempio è necessario calcolare molte volte all'interno dell'algoritmo, ma con “ingressi” differenti
- Questo conduce a due sostanziali approcci (strategie) per la risoluzione dei problemi:
 - Top-Down
 - Bottom-Up

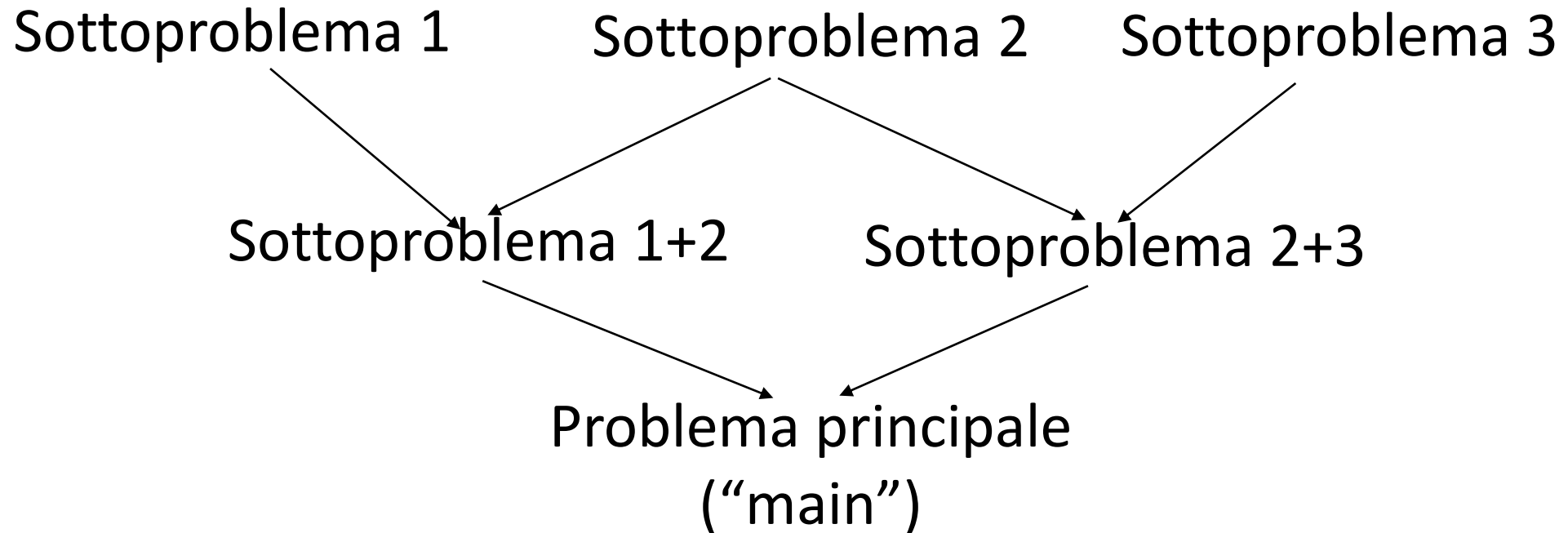
L'approccio Top-Down

- Si parte dal problema principale e via via si individuano i sottoproblemi che lo compongono



L'approccio Bottom-Up

- Si individuano problemi essenziali, il più semplici possibile, che, integrati gradualmente, permettono la soluzione del problema principale



Modularità del codice

- In generale è difficile seguire il Top-Down od il Bottom-Up in modo rigoroso, in quanto
 - certe funzionalità possono risultare utili in un secondo momento
 - non si ha un'idea chiara di come il problema principale vada risolto nella sua integrità ma si è riusciti ad individuare alcune funzionalità di base
- Si preferisce un approccio “ibrido”, chiamato anche “sandwich”, in cui le due strategie vengono condotte in parallelo
 - Ad es. su una parte del problema si segue la Top-Down, su altre la Bottom-Up
- La convergenza delle due strategie porta alla soluzione del problema
- **MODULARITA' DEL SOFTWARE**
 - si migliora la chiarezza del programma
 - ne si attenua la “rigidità”

Esempio di funzioni a noi già note in forma **evocata/chiamata**

- `r=sqrt(x);`
 - Riceve un `double x` e calcola la sua radice quadrata assegnandola ad `r`
- `a=pow(x,y);`
 - Riceve due `double x` e `y` e calcola x^y assegnandolo ad `a`
- `n=strlen(s);`
 - Riceve una stringa `s` e calcola la sua lunghezza assegnandola a `n`
- `c=strcmp(s1,s2);`
 - Confronta le stringhe `s1` ed `s2` assegnando a `c` un valore pari 0 se sono identiche
- `strcpy(s1,s2);`
 - Copia la stringa `s2` nella stringa `s1`

Funzioni in C

- Una funzione in C è caratterizzata dalla seguente sintassi:

```
Tipo_in_uscita NOME_FUNZIONE(Lista_parametri)
{
    [Parte dichiarativa;]
    Corpo della funzione;
    return Variabile di tipo Tipo_in_uscita;
}
```

Funzioni in C

- Tipo_in_uscita corrisponde al tipo di dato identificato come valore calcolato dalla funzione == rappresentazione della soluzione del sottoproblema

```
Tipo_in_uscita NOME_FUNZIONE (Li  
{  
    [Parte dichiarativa;  
    Corpo della funzione;  
    return Variabile di tipo Tipo_in_uscita;  
}
```

```
double pow(double x, double y)  
{  
    double p;  
    ...  
    /*Calcolo di  $x^y$  */  
    p=  $x^y$ ;  
    ...  
    return p;  
}
```

Funzioni in C

- **NOME_FUNZIONE** identifica il sottoproblema che essa risolve (es. Calcola_Fattoriale):

```
Tipo_in_uscita NOME_FUNZIONE (Li  
{  
    [Parte dichiarativa;]  
    Corpo della funzione;  
    return Variabile di tipo Tipo_  
}
```

```
double pow(double x, double y)  
{  
    double p;  
    ...  
    /*Calcolo di  $x^y$  */  
    p=  $x^y$ ;  
    ...  
    return p;  
}
```

Funzioni in C

- **Lista_parametri** è una lista di variabili che vengono fornite alla funzione (parametri di ingresso) ed eventualmente altre che la funzione fornisce (parametri di uscita)

```
Tipo_in_uscita NOME_FUNZIONE (Li
{
    [Parte dichiarativa;]
    Corpo della funzione;
    return Variabile di tipo Tipo_in_uscita;
}
```

```
double pow(double x, double y)
{
    double p;
    ...
    /*Calcolo di  $x^y$  */
    p=  $x^y$ ;
    ...
    return p;
}
```

Parametri di ingresso e uscita

- `Lista_parametri` presenta questa forma
 - `Lista_parametri = Tipo1[*] NomeVariabile1, Tipo2[*] NomeVariabile2, ..., TipoN[*] NomeVariabileN`
- Alcune di queste variabili vengono “prestate” alla funzione dall'esterno (es. dalla parte “principale” del programma) e vengono usate per calcolare l'uscita senza essere modificate → parametri di ingresso
- Altre possono fare “parte” della soluzione del sottoproblema e quindi essere modificate dopo l'esecuzione dell'ultima istruzione della funzione (**return**) → parametri di uscita
- Le parentesi quadre `[*]` indicano un elemento, l'asterisco, *eventualmente* aggiunto dopo la dichiarazione di tipo
- La `Lista_parametri` può anche essere vuota, in tal caso, tra parentesi si scrive semplicemente il tipo “non definito” **void**:
 - `Lista_parametri = void`

Passaggio per valore o per variabile

- Un parametro di ingresso viene tipicamente passato “per valore”: il C fa una copia del contenuto della variabile e all’interno usa quella, senza alterare quindi il contenuto del parametro passato quando la funzione termina
- Un parametro di uscita viene invece passato “per variabile”: il contenuto della variabile è accessibile alla funzione, e può essere modificato, in quanto viene passato *l’indirizzo* della funzione
 - Ciò si compie aggiungendo **un asterisco** dopo la dichiarazione di tipo:
 - Es. `Tipo1* NomeVariabile1`

Funzioni in C

- Dopo la graffa aperta, vi è eventualmente una **Parte dichiarativa** in cui possono essere dichiarate appunto delle **variabili locali** della funzione.

- Variabili locali: esse non sono “visibili” all'esterno in nessun modo
- La Parte dichiarativa comprende la c
restituita in uscita attraverso l'istruzione **return**

```
Tipo_in_uscita NOME_FUNZIONE (Li  
{
```

[Parte dichiarativa;]

Corpo della funzione;

return Variabile di tipo Tipo_in_uscita;

```
double pow(double x, double y)  
{  
    double p;  
    ...  
    /*Calcolo di  $x^y$  */  
    p=  $x^y$ ;  
    ...  
    return p;  
}
```

Funzioni in C

- Il **Corpo della funzione** è il codice che serve per risolvere il sottoproblema, che elabora le variabili nella `Lista_parametri` servendosi delle variabili dichiarate nella `Parte dichiarativa`

```
Tipo_in_uscita NOME_FUNZIONE (Li
```

```
{
```

```
    Parte dichiarativa;
```

```
    Corpo della funzione;
```

```
    return Variabile di tipo Tipo_in_uscita;
```

```
}
```

```
double pow(double x, double y)
{
    double p;
    ...
    /*Calcolo di  $x^y$  */
    p=  $x^y$ ;
    ...
    return p;
}
```

Procedure in C

- Sono esattamente equivalenti alle funzioni, con l'unica differenza che Tipo_in_uscita è sempre **void**
- La soluzione del sottoproblema è rappresentata dall'insieme dei parametri di uscita
- L'ultima istruzione **return** può anche essere omessa

```
void NOME_PROCEDURA(Lista_parametri)
{
    Parte dichiarativa;
    Corpo della procedura;
    [return;] /*opzionale*/
}
```

Chiamata di una funzione

- Una funzione in C viene **evocata (chiamata)** da un'altra funzione
- Esempio:

```
int main() /* Funzione chiamante/evocante */
{
    /* Funzione chiamata/evocata */
    printf("Ciao\n");
    return 0;
}
```

La funzione «main»

- E' la prima funzione chiamata direttamente dal «sistema operativo», ovvero il «chiamante»
- Buona programmazione impone che tutte le funzioni restituiscano un valore al chiamante, per **informarlo dell'esito delle loro operazioni**

```
int main()  
{  
    printf("Ciao\n");  
    return 0; /* info sull'esito */  
}
```

Prototipi di una funzione

- Il prototipo di una funzione corrisponde alla sua intestazione
 - Descrive l'interfaccia con l'utilizzatore
 - Non serve sapere com'è fatta «dentro» la funzione
 - Basta sapere cosa fa, cosa prende in ingresso e cosa restituisce in uscita
- Viene riportato prima della funzione main, a meno che non si tratti di una funzione di libreria
 - In tal caso, i prototipi sono già presenti nel file di libreria (stdio.h, string.h, math.h etc)
- Esempio:

```
int stampaStringa(char* stringaDaStampare) ;
```

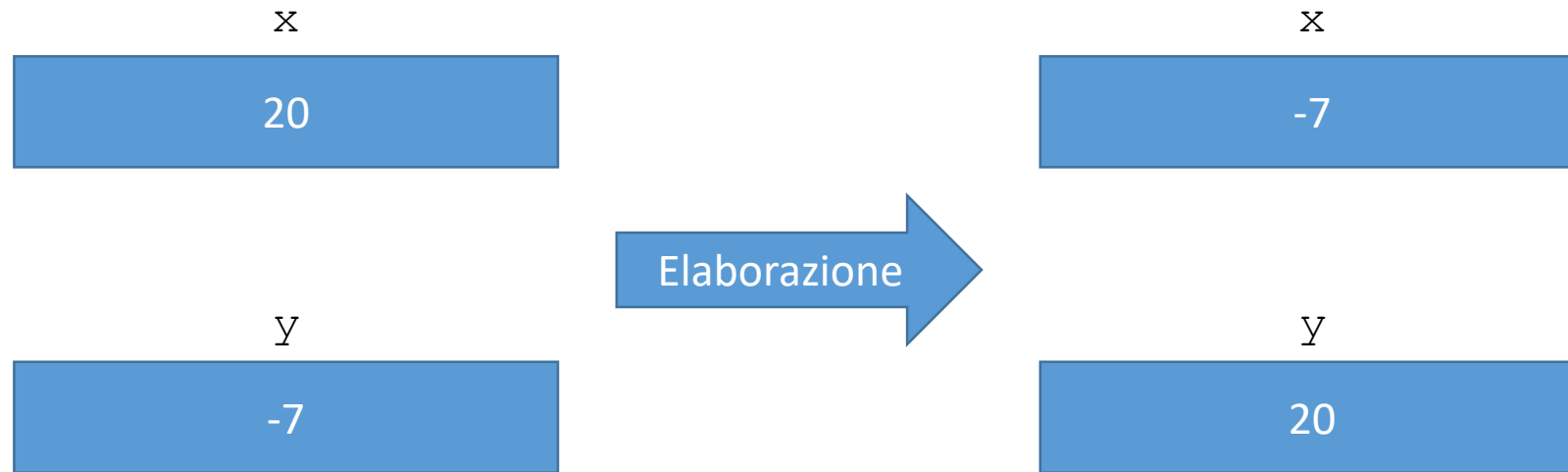
Prototipi completi di funzioni note

- `double sqrt(double x);`
 - Riceve un float `x` e calcola la sua radice quadrata restituendola in uscita
- `double pow(double x, double y);`
 - Riceve due float `x` e `y` e calcola x^y restituendolo in uscita
- `unsigned int strlen(char *s);`
 - Riceve una stringa `s` e calcola la sua lunghezza restituendola in uscita
- `unsigned int strcmp(char *s1, char *s2);`
 - Confronta le stringhe `s1` ed `s2` restituendo un intero pari a zero se le due stringhe sono identiche
- `char* strcpy(char* d, char *s);`
 - Copia la stringa `s` nella stringa `d` e restituisce `d`

Esercizio

- Scrivere un programma C che, letti due interi da tastiera x e y, li stampi a video, poi scambi i loro valori ristampandoli a video
- Per risolvere il problema è richiesto in particolare di:
 - scrivere una funzione `scambia1` che, ricevendo in ingresso due interi a e b passati **per valore**, scambi il valore di a con quello di b
 - scrivere una funzione `scambia2` che, ricevendo in ingresso due interi a e b passati **per variabile**, scambi il valore di a con quello di b
- Quale delle due funzioni sarà efficace?

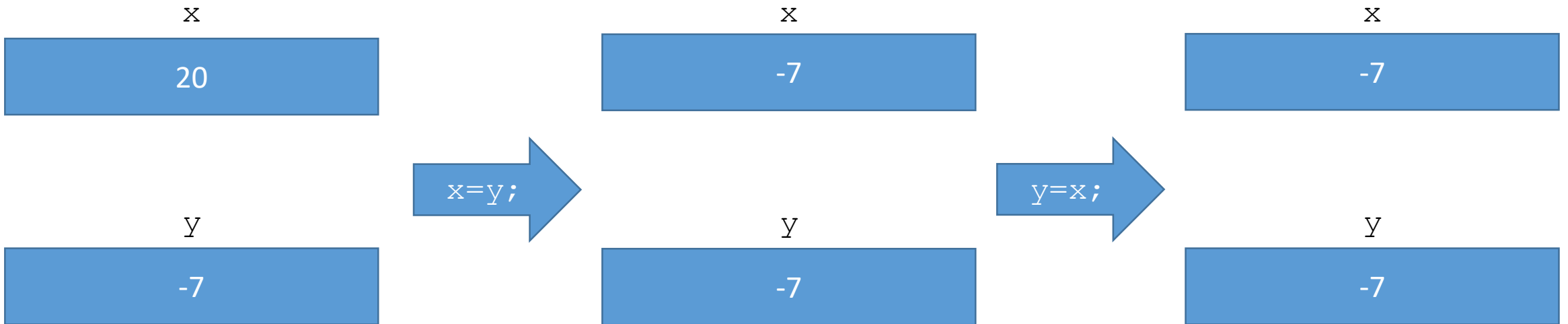
Algoritmo di scambio



Soluzione banale

$x=y;$

$y=x;$



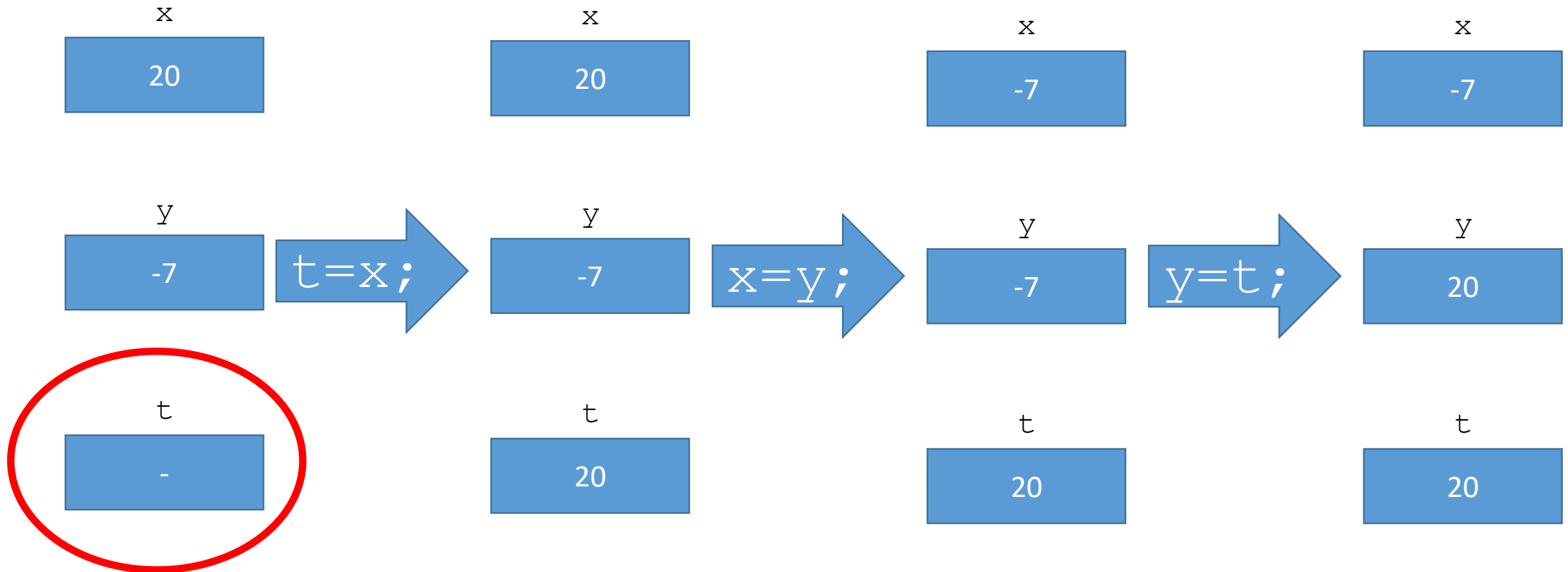
Soluzione banale

$x=y;$

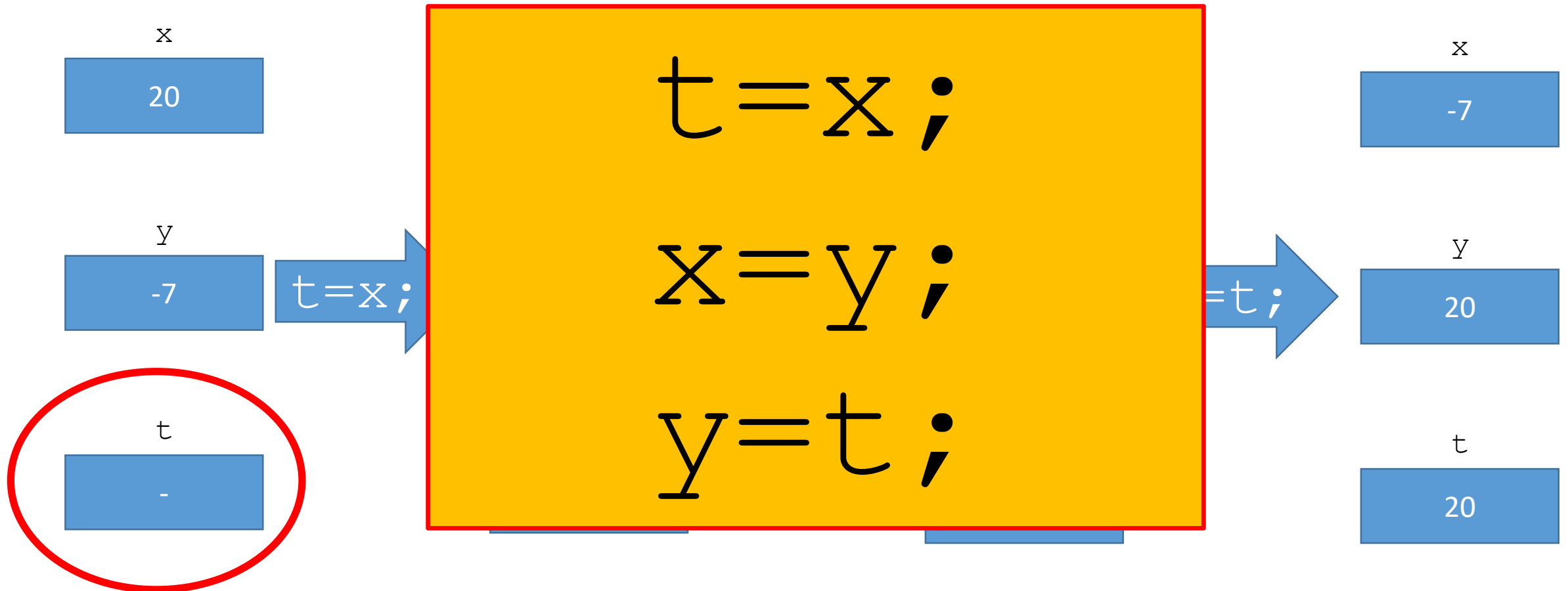
$y=x;$



Algoritmo di scambio



Algoritmo di scambio



Prima di conoscere le funzioni...

```
#include <stdio.h>

int main()
{
    int x, y; /*input/output di sistema*/
    int t; /*variabile di supporto*/

    /* Parte di lettura */
    printf("Inserisci un numero reale:\n");
    scanf("%d",&x);

    printf("Inserisci un numero realte:\n");
    scanf("%d",&y); //Occhio

    /* Prima stampa dei dati */
    printf("x=%d e y=%d\n", x, y);

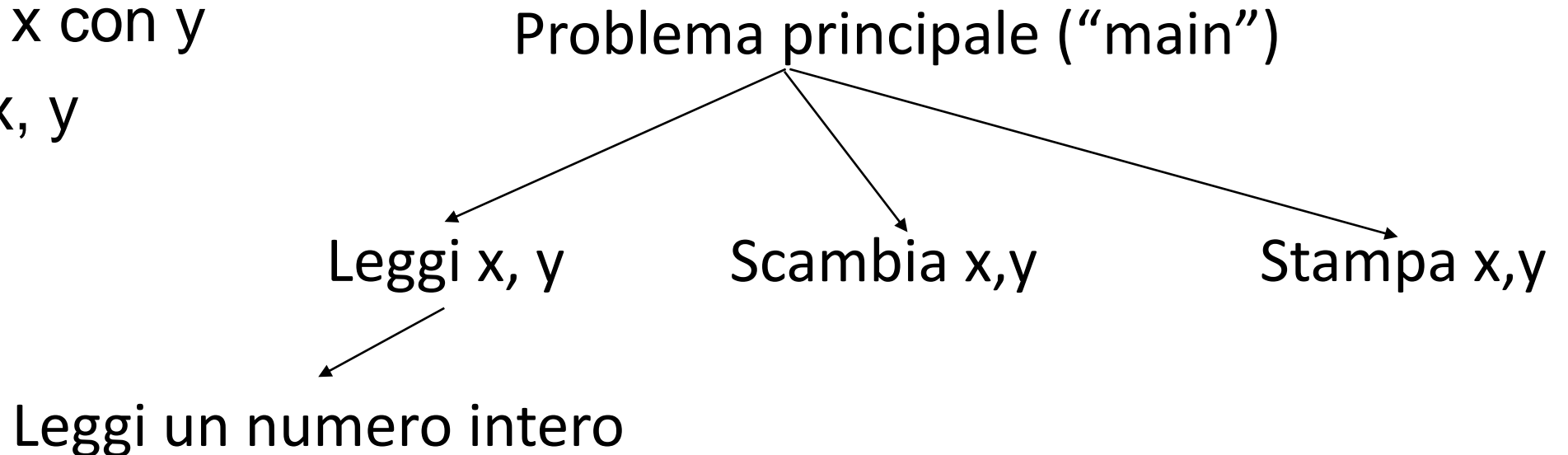
    /* Elaborazione:
    algoritmo di scambio */
    t=x;
    x=y;
    y=t;

    /* Seconda stampa dei dati */
    printf("x=%d e y=%d\n", x, y);

    return 0;
}
```

...ma ora le cose si possono fare diversamente

- Leggi x, y
- Stampa x, y
- Scambia x con y
- Stampa x, y



Il codice

```
#include <stdio.h>
```

```
/* Prototipi delle funzioni */  
int leggiIntero(void);  
void scambia1(int a, int b);  
void scambia2(int* a, int *b);  
void stampaInteri(int a, int b);
```

```
int main()  
{  
    int x, y; /*input/output di sistema*/  
  
    /* Parte di lettura */  
    x=leggiIntero();  
    y=leggiIntero();  
  
    /* Prima stampa dei dati */  
    stampaInteri(x,y);  
  
    /* Parte di elaborazione: passaggio per...  
    scambia2(&x, &y); /* ...variabile */  
  
    /*Seconda stampa dei dati*/  
    stampaInteri(x,y);  
  
    return 0;  
}
```

```

#include <stdio.h>

int main()
{
    int x, y; /*input/output di sistema*/
    int t; /*variabile di supporto*/

    /* Parte di lettura */
    printf("Inserisci un numero reale:\n");
    scanf("%d",&x);

    printf("Inserisci un numero realte:\n");
    scanf("%d",&y); //è giusta?

    /* Prima stampa dei dati */
    printf("x=%d e y=%d\n", x, y);

    /*Elaborazione: algoritmo di scambio */
    t=x;
    x=y;
    y=t;

    /*Seconda stampa dei dati*/
    printf("x=%d e y=%d\n", x, y);
    return 0;
}

```

```

#include <stdio.h>

int main()
{
    int x, y; /*input/output di sistema*/

    /* Parte di lettura */
    x=leggiIntero();

    y=leggiIntero();

    /* Prima stampa dei dati */
    stampaInteri(x,y);

    /* Elaborazione: algoritmo di scambio */
    scambia2(&x, &y); /* ...variabile */

    /*Seconda stampa dei dati*/
    stampaInteri(x,y);
    return 0;
}

```

Implementazione di `scambia1` e `scambia2`

Passaggio per valore

```
void scambia1(int a, int b)
{
    int t;
    t=a;
    a=b;
    b=t;
}
```

Passaggio per variabile

```
void scambia2(int *a, int *b)
{
    int t;
    t=*a;
    *a=*b;
    *b=t;
}
```

Le altre funzioni

```
int leggiIntero(void)
{
    int numero;
    printf("Inserire un intero.\n");
    scanf("%d",&numero);
    return numero;
}
```

```
void stampaInteri(int a, int b)
{
    printf("Primo numero: %d\nSecondo numero: %d\n",a,b);
}
```

Vantaggi nell'uso delle funzioni

- Scomposizione di un problema in sottoproblemi
- Sottoproblemi risolti usati per problemi diversi da quello iniziale
- All'utilizzatore sono trasparenti connessi al «dettaglio» della funzione
- Riduzione di errori dovuti alla ripetizione delle stesse linee di codice
- Risparmio di tempo su problemi complessi
- Codice più leggibile

Un altro esempio semplice

- Scrivere un programma C che, leggendo da tastiera due valori interi senza segno N e K, con $K \leq N$, calcoli il numero M di possibili combinazioni di N oggetti a gruppi di K e lo stampi a video
- La formula per il calcolo di M è la seguente:

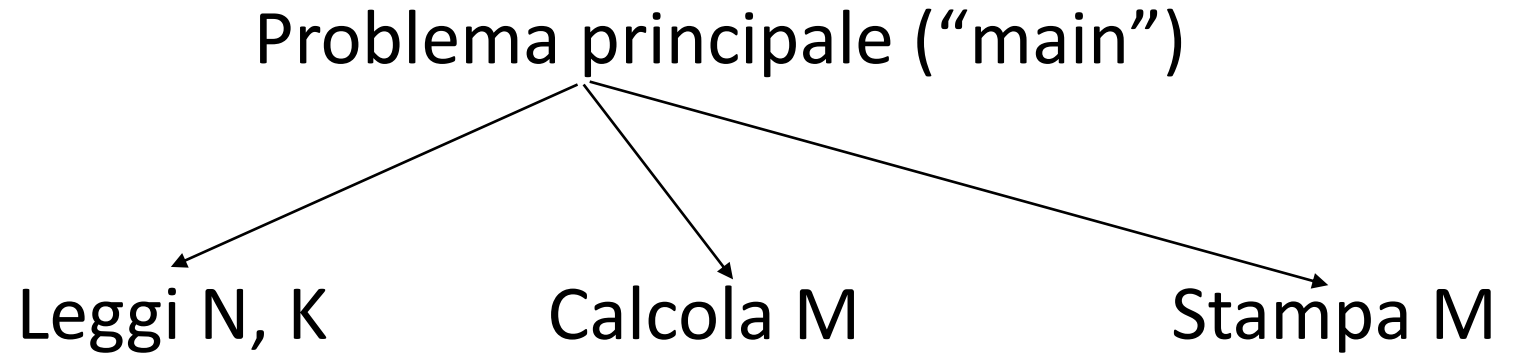
$$M = \binom{N}{K} = \frac{N!}{K!(N-K)!}$$

- Dove l'espressione $j!$ significa "fattoriale di j" ed è data a sua volta dalla seguente ($0!=1$):

$$j! = \prod_{i=1}^j i$$

Algoritmo: soluzione di “alto livello”

- Leggi due valori da tastiera N e K
- Calcola M
- Stampa a video M



Algoritmo: individuazione delle funzionalità

➤ Leggi due valori da tastiera N e K

- Uso scanf

➤ Calcola M

- Calcola N!
- Calcola (N-K)! ➡
- Calcola K!

$$M = \binom{N}{K} = \frac{N!}{K!(N-K)!}$$

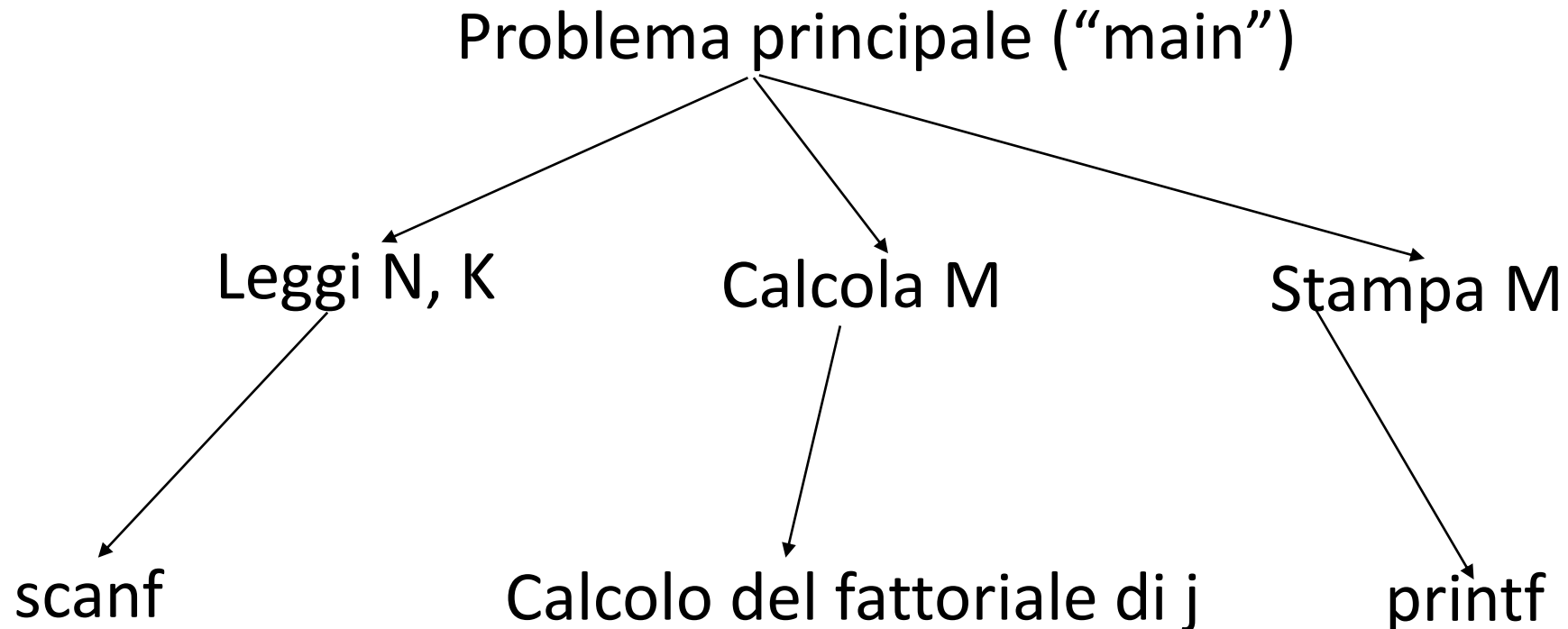
il calcolo del fattoriale è un'operazione ripetitiva

➤ Stampa a video M

- Uso printf

Quindi...

- Siamo riusciti a scomporre il problema in problemi più elementari
- Resta da capire come scrivere “Calcolo del fattoriale di j”



Torniamo al problema

- Procediamo in modo top-down, e scriviamo una prima versione del programma

```
/*Programma per il calcolo combinatorio*/  
#include <stdio.h>  
int main(void)  
{  
    unsigned int N, K, M;  
    /*Leggi N, K da tastiera*/  
    /*Calcola M*/  
    /*Stampa M a video*/  
    return 0;  
}
```

Leggi N, K da tastiera: procedura leggi

- Scriviamo una procedura che legga da tastiera due interi e li memorizzi in N e K, passati in ingresso come parametri

```
void leggi(unsigned int* N, unsigned int* K)
{
    printf("Inserisci i valori non negativi N e K:\n");
    scanf("%d %d", N, K);
    return;
}
```

Passaggio per variabile

Stampa M a video: procedura

stampa

- Soluzione banalissima:

```
void stampa(unsigned int M)
{
    printf("Il valore richiesto è pari a: %d\n",M) ;
    return;
}
```

Calcola M: funzione calcola

- Scriviamo una funzione che restituisca appunto il valore richiesto:

```
unsigned int calcola(unsigned int N, unsigned int K)
{
    unsigned int fattN, fattK, fattD, M;
    /*Calcola il fattoriale di N*/
    /*Calcola il fattoriale di K*/
    /*Calcola il fattoriale di D=N-K*/
    M=fattN/(fattK*fattD); /*formula del calcolo (N K)*/
    return M;
}
```

Calcolo del fattoriale: funzione fattoriale

- A questo punto scriviamo la funzione del fattoriale parametrizzata su un generico intero senza segno j :

```
unsigned int fattoriale(unsigned int j)
{
    unsigned int i, fattj;      /*parte dichiarativa*/

    fattj=1;
    if (j>1)
        for (i=2; i<=j; i++)
            fattj=fattj*i;
    return fattj;
}
```

} /*corpo funzione*/

Ritorniamo a calcola

- Sostituiamo ai commenti le **chiamate** alla funzione con i relativi parametri

```
unsigned int calcola(unsigned int N, unsigned int K)
{
    unsigned int fattN, fattK, fattD, M;
    fattN=fattoriale(N);
    fattK=fattoriale(K);
    fattD=fattoriale(N-K);
    M=fattN/(fattK*fattD); /*formula del calcolo (N K)*/
    return M;
}
```

Completamento del programma 1

- Completiamo il programma sostituendo ai commenti le opportune chiamate

```
/*Programma per il calcolo combinatorio*/  
#include <stdio.h>  
int main(void)  
{  
    unsigned int N, K, M;  
    leggi (&N, &K) ;  
    M=calcola (N, K) ;  
    stampa (M) ;  
    return 0 ;  
}
```

Completamento del programma 2

- Se però scrivessimo su un editor il programma alla slide precedente, e tentassimo di eseguirlo, avremmo subito un errore in fase di *compilazione*:
 - il compilatore ci chiederebbe “dove sono le funzioni leggi, calcola, stampa perché io possa tradurle?”
- Le funzioni che abbiamo scritto possono essere inserite in tre modi:
 - Nello stesso file del programma **prima** della funzione `main`
 - Nello stesso file del programma **dopo** la funzione `main`, ma facendo precedere a questa le intestazioni di tutte le funzioni, chiamate **prototipi di funzione**
 - In un file a parte

Soluzione 1: prima della main

```
/*Programma per il calcolo combinatorio*/
#include <stdio.h>

void leggi(unsigned int* N, unsigned int* K)
{
    printf("Inserisci i valori non negativi N e K:\n");
    scanf("%d %d",N,K);
    return;
}

void stampa(unsigned int M)
{
    printf("Il valore richiesto è pari a: %d\n",M);
    return;
}

unsigned int fattoriale(unsigned int j)
{
    unsigned int i, fattj;      /*parte dichiarativa*/
    fattj=1;
    if (j>1)
        for (i=2; i<=j; i++)    /*corpo
funzione*/
            fattj=fattj*i;
    return fattj;
}
```

```
unsigned int calcola(unsigned int N, unsigned int K)
{
    unsigned int fattN, fattK, fattD, M;
    fattN=fattoriale(N);
    fattK=fattoriale(K);
    fattD=fattoriale(N-K);
    M=fattN/(fattK*fattD); /*formula del calcolo (N K)*/
    return M;
}

int main(void)
{
    unsigned int N, K, M;
    leggi(&N,&K);
    M=calcola(N,K);
    stampa(M);
    return 0;
}
```

Soluzione 2: dopo la main

```
/*Programma per il calcolo combinatorio*/
#include <stdio.h>

void leggi(unsigned int* N, unsigned int* K);
void stampa(unsigned int M);
unsigned int fattoriale(unsigned int j);
unsigned int calcola(unsigned int N, unsigned int K);

int main(void)
{
    unsigned int N, K, M;
    leggi(&N, &K);
    M=calcola(N, K);
    stampa(M);
    return 0;
}

void leggi(unsigned int* N, unsigned int* K)
{
    printf("Inserisci i valori non negativi N e K:\n");
    scanf("%d %d", N, K);
    return;
}

void stampa(unsigned int M)
{
    printf("Il valore richiesto è pari a: %d\n", M);
    return;
}
```

Elementi di Informatica - A.A. 2021/22 - Prof. Gian Luca Marcialis

```
unsigned int fattoriale(unsigned int j)
{
    unsigned int i, fattj;    /*parte dichiarativa*/
    fattj=1;
    if (j>1)
        for (i=2; i<=j; i++)    /*corpo funzione*/
            fattj=fattj*i;
    return fattj;
}

unsigned int calcola(unsigned int N, unsigned int K)
{
    unsigned int fattN, fattK, fattD, M;
    fattN=fattoriale(N);
    fattK=fattoriale(K);
    fattD=fattoriale(N-K);
    M=fattN/(fattK*fattD); /*formula del calcolo (N K)*/
    return M;
}
```

Soluzione 3: in un file a parte

```
FILE PRINCIPALE "calcolo_combinatorio.c"  
/*Programma per il calcolo combinatorio*/  
#include "funzioni.c"
```

```
int main(void)  
{  
    unsigned int N, K, M;  
    leggi(&N,&K);  
    M=calcola(N,K);  
    stampa(M);  
    return 0;  
}
```

```
FILE "funzioni.c"  
#include <stdio.h>
```

```
void leggi(unsigned int* N, unsigned int* K);  
void stampa(unsigned int M);  
unsigned int fattoriale(unsigned int j);  
unsigned int calcola(unsigned int N, unsigned int K);
```

```
void leggi(unsigned int* N, unsigned int* K)  
{  
    printf("Inserisci i valori non negativi N e K:\n");  
    scanf("%d %d",N,K);  
    return;  
}
```

```
void stampa(unsigned int M)  
{  
    printf("Il valore richiesto è pari a: %d\n",M);  
    return;  
}
```

```
unsigned int fattoriale(unsigned int j)  
{  
    unsigned int i, fattj;    /*parte dichiarativa*/  
    fattj=1;  
    if (j>1)  
        for (i=2; i<=j; i++)    /*corpo funzione*/  
            fattj=fattj*i;  
    return fattj;  
}
```

```
unsigned int calcola(unsigned int N, unsigned int K)  
{  
    unsigned int fattN, fattK, fattD, M;  
    fattN=fattoriale(N);  
    fattK=fattoriale(K);  
    fattD=fattoriale(N-K);  
    M=fattN/(fattK*fattD); /*formula del calcolo (N K)*/  
    return M;  
}
```

Passaggio di vettori come parametri

- Il passaggio di vettori è l'unico caso in cui si è obbligati ad usare il passaggio **per variabile**
- Il C passa alla funzione invocata solo **l'indirizzo del primo elemento** del vettore, quindi la manipolazione dei valori all'interno **ne determina l'alterazione permanente visibile all'esterno della funzione stessa**

- Esempio di prototipo con passaggio di vettore intero:

```
void elaboraVettore(int* v);  
/* non c'è differenza tra variabile intera singola passata per  
variabile e vettore, visto come sequenza di variabili intere.  
Il vettore è sempre passato per variabile.  
*/
```

Esercizio 1

- Scrivere una procedura C che, ricevendo in ingresso un vettore di interi v e due valori i e j , permuti il valore in posizione i con quello in posizione j

```
/*Procedura per lo scambio di due valori in un vettore*/  
void scambia(int* v, int i, int j)  
{  
    int temp;  
    temp=v[i];  
    v[i]=v[j];  
    v[j]=temp;  
    return;  
}
```

Il vettore v viene passato antepoendo al suo identificatore simbolico il carattere ‘*’

Lato chiamante:

```
int v[10];  
/*Assegno i vari valori di v (per esempio con un'altra funzione)*/  
/*Invoco scambia su due elementi di v, diciamo il terzo e il  
quinto*/  
scambia(v,2,4); /*altra possibile scrittura: scambia(&v[0],2,4);*/
```

Esercizio 2

- Scrivere una funzione C che, ricevendo un vettore v di N elementi (N definito tramite `#define`), scriva in un secondo vettore di pari dimensioni la seguente corrispondenza: $w[N-1] == v[0]$, $w[N-2] == v[1]$, ..., $w[N-i] == v[i-1]$, ..., $w[0] == v[N-1]$

```
void inverti(int*v, int*w)
{
    int i;
    for (i=0; i<N; i++)
        w[i]=v[N-(i+1)];
}
```

Esercizio 3

- Scrivere una funzione `media_mobile` C che, ricevendo in ingresso un vettore `v` di `N` interi, scriva su un secondo vettore `w`, sempre di `N` interi:
 - $w[i] = (v[i] + v[i+1]) / 2;$
 - Per $i=0, \dots, N-2$
 - $w[N-1] = (v[N-1] + v[0]) / 2$

Soluzione

```
void media_mobile (int*v, int *w)
{
    int i;

    for (i=0; i<=N-2; i++)
        w[i] = (v[i] + v[i+1]) / 2;

    w[N-1] = (v[N-1] + v[0]) / 2;
}
```

Esercizio 4

- Scrivere una funzione `C_somma` che, ricevendo in ingresso tre vettori di interi `v`, `w` e `z` di dimensione `N` predefinita (es. attraverso `#define`) scriva in `z` :
 - `z[0] = v[0] + w[N-1]`
 - `z[1] = v[1] + w[N-2]`
 - ...
 - `z[N-1] = v[N-1] + w[0]`

Classroom work

- Scrivere una funzione `stampa` che, ricevendo in ingresso un vettore di interi di dimensione N , stampa a video tutti i suoi elementi
- Scrivere la funzione `main` con una chiamata a ciascuna delle funzioni studiate in queste slide, inclusa la precedente
- Strutturare la `main` in modo tale da:
 - Leggere i dati di input
 - Sottomettere i dati ad elaborazione mediante le funzioni scritte
 - Stampare a video i dati di output mediante la funzione `stampa`

Suggerimento

```
#include <stdio.h>
#define N 50 /*dimensione di tutti i vettori*/
/* Qua mettiamo tutte le definizioni di funzione viste in precedenza */
int main()
{
    /*Parte dichiarativa di vettori e variabili*/

    /*Leggo da tastiera tanti valori interi quanti sono necessari e li
memorizzo nei vettori*/

    /*Invoco le funzioni precedentemente definite per elaborare i
vettori*/

    /*Utilizzo una funzione che mi faccia vedere se il calcolo è
corretto, stampando a video per esempio il contenuto degli elementi di
un vettore passato come parametro */

    return 0;
}
```

Calcolo del Massimo Comune Divisore

- Il Massimo Comune Divisore tra due interi non negativi x e y è definito come il massimo intero per il quale è possibile dividere con resto nullo sia x che y
- Per calcolarlo si usa l'algoritmo di Euclide
 - Input: x, y , con $x \geq y$
 - Output: $z \leftarrow \text{MCD}(x, y)$
 - Finché $y \neq 0$, ripeti:
 - $r = \text{resto di } x/y$;
 - $x = y$;
 - $y = r$;
 - $Z = X$;

Esercizio 5

- Scrivere una funzione che, ricevendo in ingresso due interi x e y , restituisca il loro massimo comune divisore calcolato secondo l'algoritmo di Euclide
- Scrivere un programma C, organizzato modularmente tramite funzioni di lettura, stampa ed elaborazione, che letti due valori interi da tastiera, stampi a video i valori stessi ed il loro massimo comune divisore.

Soluzione (parziale)

```
int MCD(int x, int y)
{
    int r;

    while (y!=0)
    {
        r=x%y;
        /*l'operatore % permette di calcolare subito il resto*/
        x=y;
        y=r;
    }
    return x; /* il chiamante: z=MCD(x,y); */
}
```

Altri esercizi

- Scrivere una funzione `prodotto` che restituisce il prodotto di un intero `x` per un intero `y` forniti in ingresso
 - Implementazione del metodo: $x*y = x + \dots + x$ (`y` volte)
- Scrivere una funzione `potenza` che restituisce la potenza di una base intera non nulla `b` per un esponente non negativo `e` forniti in ingresso
 - Senza usare `pow`
- Scrivere una funzione `conta_minuscole` che restituisce il numero di caratteri minuscoli in una stringa ricevuta in ingresso

Il concetto di blocco

- Definizione:
un **blocco** consiste di due parti sintattiche racchiuse tra parentesi graffe:
 - Una parte dichiarativa (facoltativa)
 - Una sequenza di istruzioni
- Diversi blocchi possono comparire internamente al `main` o alle *funzioni* che compongono un programma

Esempi di blocchi

```
#include <stdio.h>

int main()
{
    int a,b;
    ...
    /* blocco1 */
    {
        char a,c;
        ...
    }
    ...
}
```

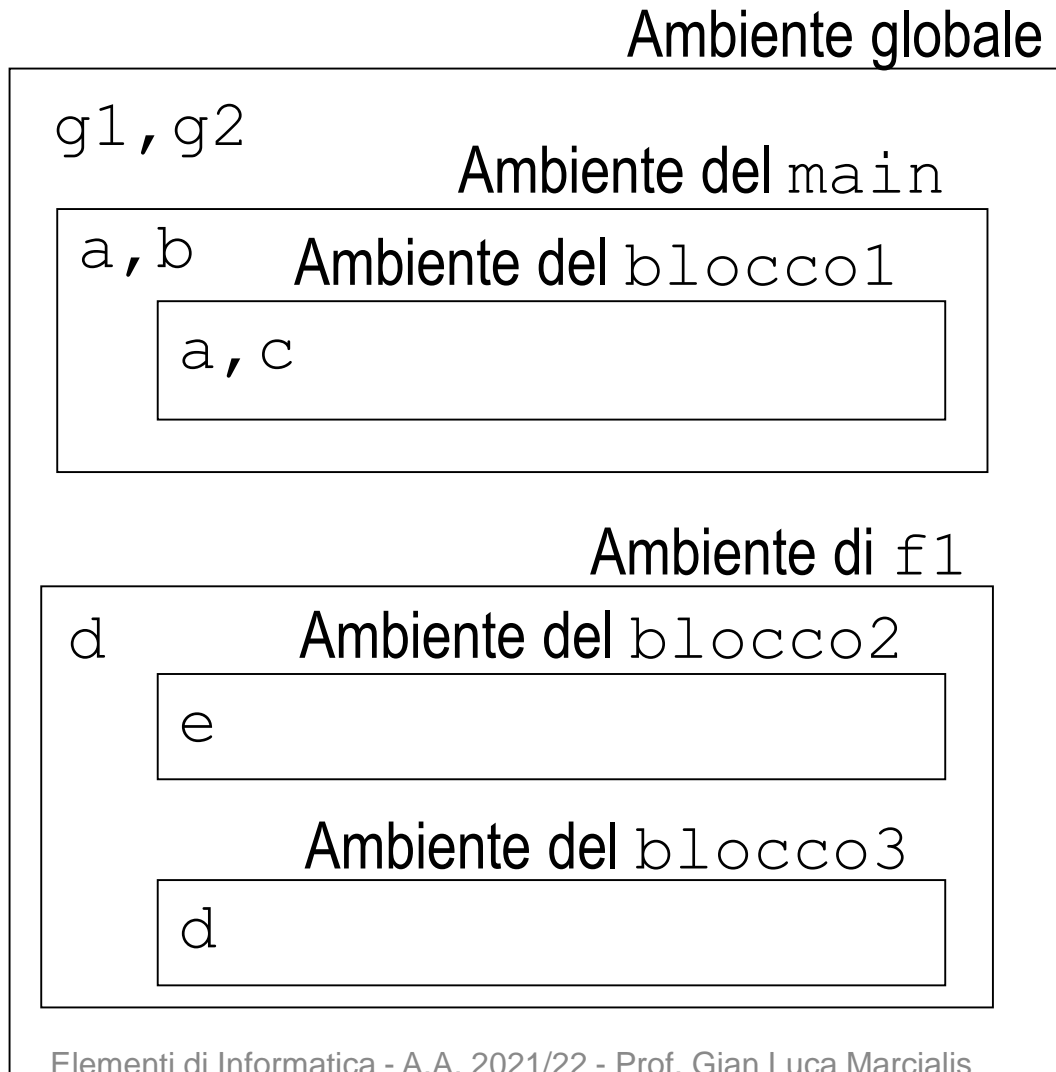
continua⇒

```
int f1(int par1, int par2)
{
    int d;
    ...
    {
        /* blocco2 */
        int e;
        ...
    }
    {
        /* blocco3 */
        int d;
        ...
    }
    ...
}
```

Visibilità delle variabili

- Vediamo alcune definizioni:
 - Si dice **ambiente globale di un programma** l'insieme di tutti gli elementi dichiarati nella sua parte dichiarativa globale
 - Si dice **ambiente locale di una funzione** l'insieme di tutti gli elementi dichiarati nella sua parte dichiarativa e nella sua testata
 - Si dice **ambiente di un blocco** l'insieme di tutti gli elementi dichiarati nella sua parte dichiarativa
- Il concetto di ambiente permette di dichiarare più volte lo stesso identificatore anche con significati diversi, purché in ambienti diversi

Visibilità delle variabili – modello a contorni



Elementi nell'**ambiente globale** possono essere “visti” da tutte le funzioni e i blocchi

Elementi nell'**ambiente locale** di una funzione possono essere “visti” da tutti i blocchi contenuti nella funzione

Gli elementi nell'**ambiente di un blocco** possono essere “visti” da tutte le istruzioni nel blocco e dai blocchi in esso contenuti

In caso di più definizioni dello stesso identificatore:

è valida la definizione dell'ambiente più vicino al punto di utilizzo

Tempo di vita delle variabili

- Dichiarazione di una variabile:
allocazione in memoria dello spazio per la sua rappresentazione
- Due categorie di variabili:
 - **Variabili fisse o statiche:**
allocate una sola volta e vengono distrutte solo quando termina l'esecuzione del programma.
Sono le variabili «globali» del programma e quelle della funzione `main`.
 - **Variabili automatiche:**
create ogni volta che si entra nel loro ambito di visibilità e che vengono distrutte all'uscita da tale ambiente.
Sono le variabili dichiarate a livello di funzione, procedura e blocco.

Per saperne di più

- Ceri, Mandriola, Sbattella, *Informatica – arte e mestiere*, Capo. 7-8, McGraw-Hill
- Kernighan, Ritchie, *Il linguaggio C*, Cap. 4, Pearson-Prentice Hall