

Funzioni

introduzione alla programmazione

Introduzione

- abbiamo visto che in C++ ogni programma deve comprendere *almeno una funzione* (`main`)
- abbiamo già parlato di estensioni procedurali, di programmazione strutturata...
- ...l'idea è legata all'intuizione di **scomporre un problema in parti più piccole** (più gestibili e anche più comprensibili) che possano essere riusate quando necessario
- In C++
 - funzioni predefinite
 - funzioni definite dal programmatore

Funzioni

- in algebra una funzione è una regola che mette in corrispondenza valori (*argomenti*) e un unico valore di ritorno

$f(x) = 2x+5$ **definisco la funzione**

così $f(1) = 7$, $f(2)=9$, ... **uso la funzione (con valori specifici)**

- In programmazione il concetto è analogo
- Il linguaggio C++ mette a disposizione varie funzioni predefinite organizzate in librerie separate
- Inoltre dà la possibilità di definire nuove funzioni

Funzioni predefinite: esempi già visti

- libreria **cmath**
 - funzioni `ceil(x)`, `cos(x)`, `exp(x)`, `fabs(x)`, `floor(x)`, `pow(x,y)`, `sqrt(x)` ,
...
- libreria **cstdlib**
 - funzioni `abs(x)`, ...
- per usarle bisogna includere il file di intestazione associato alla libreria

```
#include<cmath>
#include<cstdlib>
```

Funzioni definite dal programmatore

- **funzioni che restituiscono un valore** - ritornano un dato di un certo tipo (enunciato return) e quindi possono essere usate all'interno di espressioni (a seconda del tipo)
- **funzioni void** - non restituiscono nulla e quindi non necessitano di un return

in questi casi l'enunciato return può essere usato semplicemente per uscire dalla funzione

Funzioni definite dal programmatore

- Come per le funzioni algebriche dobbiamo distinguere due momenti:
 - La definizione della funzione
 - Il suo uso

In C++ questi due momenti sono formalizzati con due diverse sintassi:

- Definizione della funzione
- Invocazione o chiamata

Funzioni definite dal programmatore

- **Quando** progettare una function a partire da un modulo:
 - se il modulo è di una sola riga non ha senso (quasi mai)
 - se il modulo è formato da varie righe allora conviene

1. Migliora la leggibilità
2. Riduce la possibilità di errori
3. Semplifica interventi futuri
4. Facilita il riuso

modulo = un blocco di codice separato dal resto

Modularità del codice

- Un modulo è un blocco di codice separato dal resto, i cui dettagli implementativi possono essere nascosti (encapsulation)
- Interfaccia - specifica come esso debba essere invocato
 - in questo modo possiamo per esempio modificare il modulo senza preoccuparci del resto (l'importante è non cambiare l'interfaccia)
 - parametri del modulo - un insieme di variabili che contengono valori di ingresso e/o uscita al modulo

Dove inserire le funzioni

- le funzioni possono essere ovunque nel programma
- una scelta classica è inserirle dopo il main, in questo caso prima del main dovremmo inserire i *prototipi* delle funzioni
 - in C++ dobbiamo dichiarare un identificatore prima di usarlo
- un'altra possibilità è scrivere le funzioni prima del main

Funzioni void - sintassi

- DEFINIZIONE DELLA FUNZIONE

```
void NomeDellaFunzione(elenco parametri formali)
{
...;
}
```

- INVOCAZIONE DELLA FUNZIONE

```
NomeDellaFunzione(elenco parametri attuali);
```

Esempio

esempio semplice:
funzioni senza parametri!

```
#include <iostream>

using namespace std;

// Prototipi delle funzioni

void Stampa2Righe();
void Stampa4Righe();

// Funzione main

int main()
{
    Stampa2Righe();    // invocazione o chiamata di funzione
    cout << "CIAO\n";
    Stampa4Righe();    // invocazione o chiamata di funzione
    return 0;
}

// Dichiarazioni di funzione

void Stampa2Righe()
{
    cout << "*****" << endl;
    cout << "*****" << endl;
}

void Stampa4Righe()
{
    Stampa2Righe();
    Stampa2Righe();
}
```

Esempio

```
#include <iostream>

using namespace std;

// Prototipi delle funzioni
void StampaRighe(int);

// Funzione main
int main()
{
    StampaRighe(2);
    cout << "CIAO\n";
    StampaRighe(4);
    return 0;
}

// Dichiarazioni di funzione

void StampaRighe(int n)
{
    for (int i=0; i<n; ++i)
        cout << "*****" << endl;
}
```

parametro attuale

parametro formale

// header o intestazione
// qui inizia il corpo della funzione

Parametri delle funzioni

- **parametro formale** - una variabile dichiarata nell'intestazione di una funzione
- **parametro attuale** - una variabile o espressione presente nell'invocazione di una funzione

Istruzione return

- una funzione di tipo void non ha valore di ritorno, però può essere utile avere uno strumento per uscire dalla funzione quando necessario

```
void StampaRighe(int n)
{
    if (n < 1)
        return;
    for (int i=0; i<n; ++i)
        cout << "*****" << endl;
}
```

Problem Solving

- Scrivere un programma che visualizzi uno schema (triangolo di asterischi) come questo

```
      *
     * *
    * * *
   * * * *
  * * * * *
 * * * * * *
* * * * * * *
 * * * * * * *
  * * * * * * *
    * * * * * *
      * * * * *
```

- l'altezza del triangolo deve poter essere impostata dall'utente

Le funzioni che restituiscono un valore

- sqrt, abs, pow, ... sono esempi di funzioni predefinite che restituiscono un valore
- Una differenza fondamentale rispetto alle funzioni void è *il modo in cui vengono chiamate*
- Le funzioni che restituiscono un valore sono spesso chiamate come **parte di un'espressione**
- Le funzioni void sono chiamate come **istruzioni indipendenti**

Le funzioni che restituiscono un valore

- In altre parole possiamo dire che ha senso progettare una funzione che restituisce un valore quando
 - progettiamo una funzione che abbia senso ritorni solo un valore
 - riteniamo sia sensato utilizzare tale funzione all'interno di espressioni

Esempio

- **Dichiarazione:**

```
int abs(int number)
{
    if (number < 0 )
        number = -number;
    return number;
}
```

- **Chiamata:**

```
int n;
int p = -8;
n=abs(p) ;
```

Funzioni che restituiscono un valore: sintassi

- DEFINIZIONE DELLA FUNZIONE

```
tipoDellaFunzione NomeDellaFunzione (elenco  
parametri formali)  
{  
...;  
return espressione;  
}
```

- INVOCAZIONE DELLA FUNZIONE

```
var=NomeDellaFunzione (elenco parametri attuali);
```

Enunciato return

- Quando l'enunciato return viene eseguito all'interno di una funzione, questa termina immediatamente e il controllo torna alla *funzione invocante*
- L'enunciato che ha invocato la funzione viene sostituito dal valore restituito da return

Esempio

- ```
double larger(double x, double y)
{
 double max;
 if (x>=y)
 max=x;
 else
 max = y;
 return max;
}
```

- ```
double larger(double x, double y)
{
    if (x>=y)
        return x;
    else
        return y;
}
```

- ```
double larger(double x, double y)
{
 if (x>=y)
 return x;
 return y;
}
```

# funzioni booleane

- funzioni il cui valore di ritorno è booleano
- sono molto utili per semplificare condizioni complesse all'interno di loop o selezioni
- Esempio

```
bool isTriangle(float angle1, float angle2, float angle3)
{
 return (fabs(angle1+angle2+angle3 - 180.0) < 0.00001)
}
```

# funzioni booleane

```
bool is_even(int n) {
 if ((n%2) == 0) return true;
 return false;
}
```

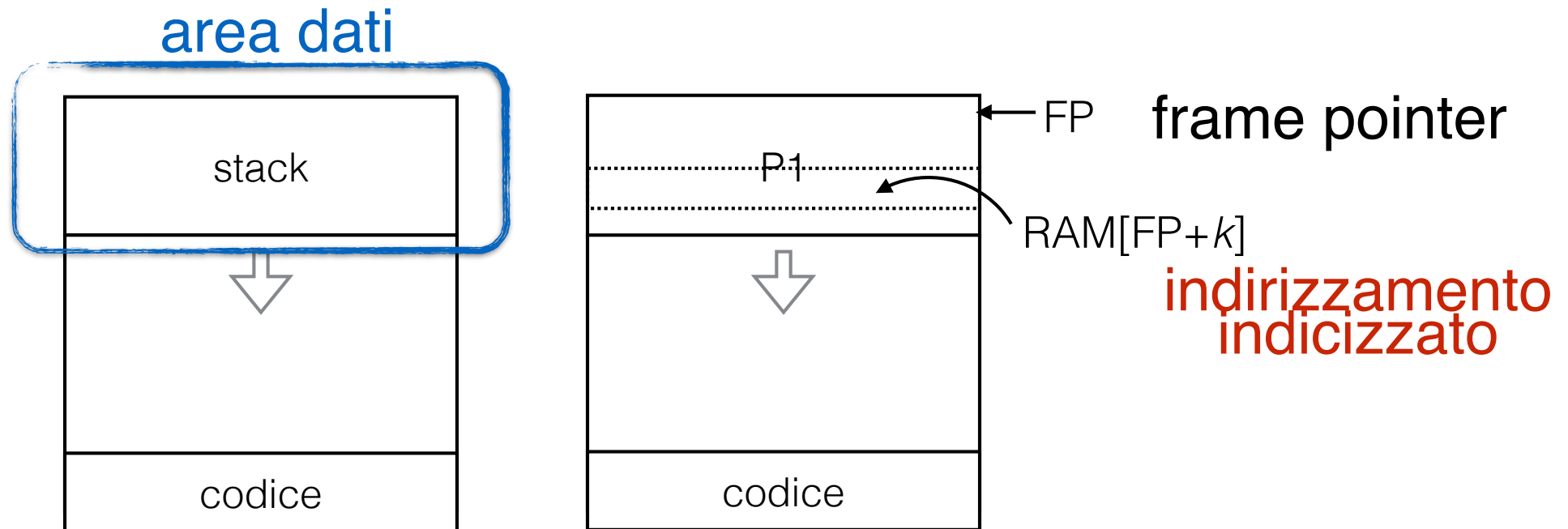
sintatticamente corretto,  
ma non è una buona pratica

```
bool is_even(int n) {
 return ((n%2) == 0);
}
```

molto meglio così!!

# Parentesi su allocazione della memoria: lo **stack**

- Ricordiamo:
  - separazione netta tra codice e dati
  - separazione tra codice e/o dati di programmi diversi

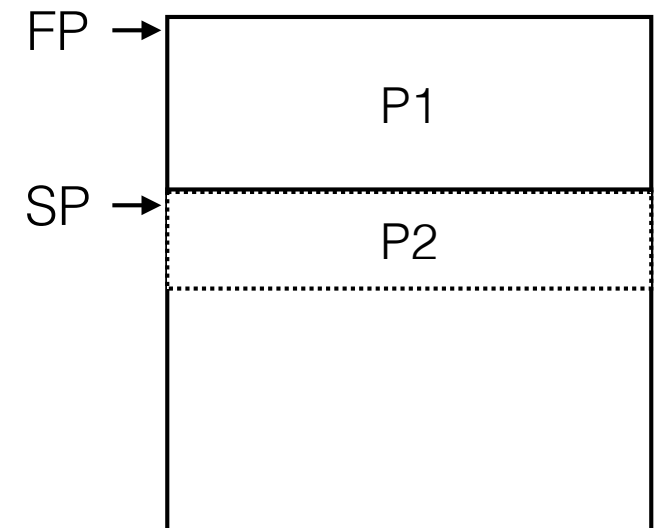


il FP viene assegnato ad un programma P1 dal sistema operativo



# Parentesi su allocazione della memoria: lo **stack**

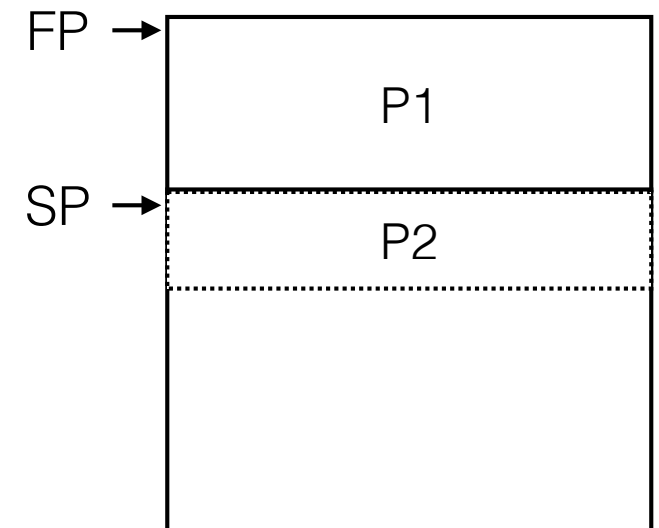
- in alcuni momenti il programma P1 può essere sospeso per attivare un altro programma P2 (che potrebbe essere un'estensione procedurale / chiamata di funzione)
- L'area dati di P2 può essere allocata nello spazio di memoria libero dopo quella di P1
- Il registro *stack pointer* (*SP*) fa sì che l'area di P1 non venga intaccata



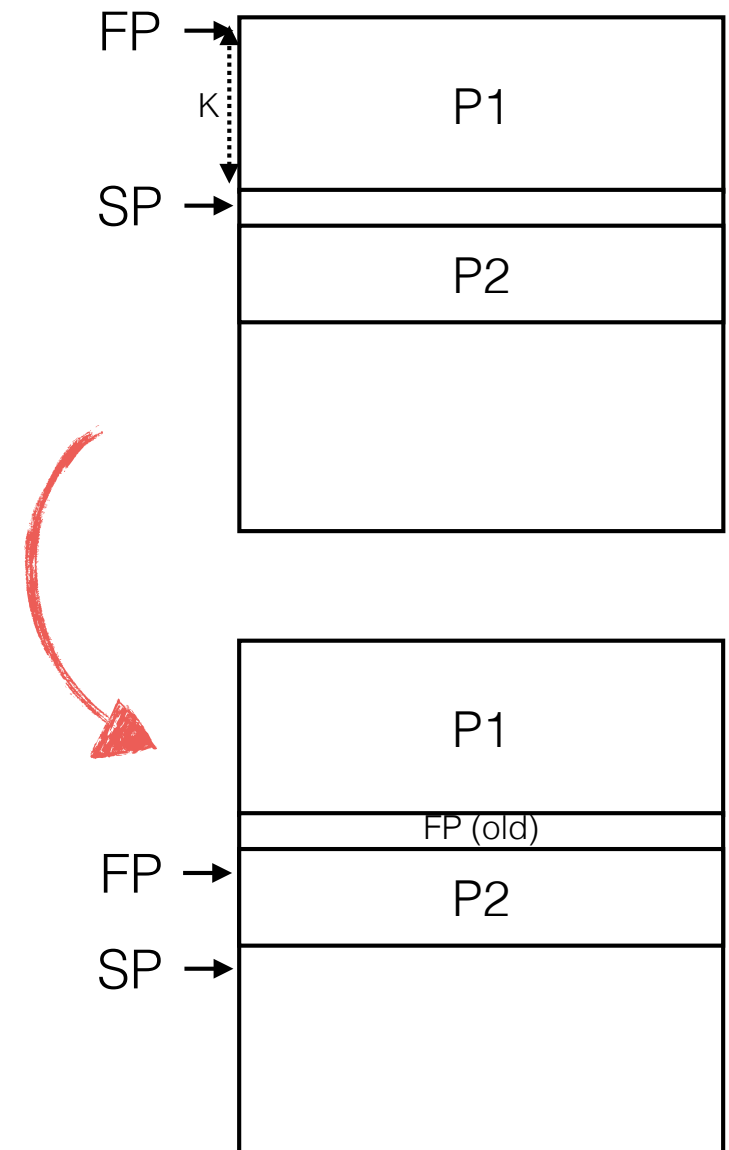
il valore dello SP può essere calcolato da P1

# Parentesi su allocazione della memoria: lo **stack**

- Il programma P1 può accedere ai dati attraverso l'indirizzamento indicizzato  
 $\text{RAM}[\text{FP}+k]$
- $\text{FP} \leq k < \text{SP}$

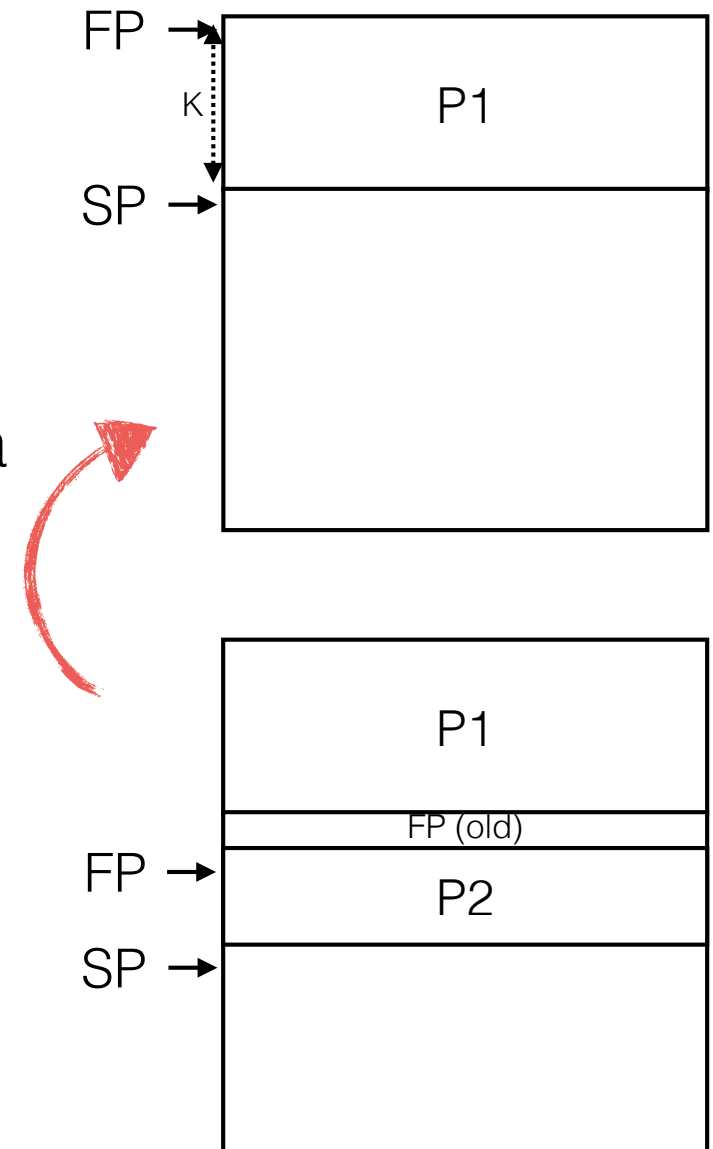


# Parentesi su allocazione della memoria: lo **stack**



# Parentesi su allocazione della memoria: lo **stack**

- Eliminazione della fetta di memoria di una sottoprocedura appena conclusa



# Passaggio dei parametri

- parametro passato per valore - un parametro formale che riceve **una copia** del contenuto del corrispondente parametro attuale

```
void esempio(int param)
```

- parametro passato per riferimento - un parametro formale che riceve **l'indirizzo di memoria** (la locazione) del parametro attuale corrispondente

```
void esempio (int& param)
```

# Passaggio per valore

Nel passaggio per valore:

- i parametri attuali sono valutati
- il loro valore è memorizzato in variabili locali alla funzione che corrispondono ai parametri formali
- ogni modifica all'interno del corpo della funzione riguarderà le variabili locali

# Passaggio per riferimento

Nel passaggio per riferimento:

- i parametri attuali devono essere variabili
- ogni modifica all'interno del corpo della funzione riguarderà i parametri attuali

il passaggio per riferimento:

- (ottimizza l'uso della memoria) consente di passare dati molto grandi senza doverli copiare nelle variabili locali delle funzioni
- ha lo svantaggio di rendere i programmi incomprensibili (va usato con cautela)

# Esempio “giocattolo” sul passaggio dei parametri

```
#include <iostream>

using namespace std;

// Prototipi delle funzioni
void SommoUno(int);
void SommoUnoRif(int&);

int main()
{
 int a=0;
 cout << "All'inizio " << a << endl;
 SommoUno(a);
 cout << "Dopo il passaggio per valore " << a << endl;

 SommoUnoRif(a);
 cout << "Dopo il passaggio per riferimento " << a << endl;
 return 0;
}

// Dichiarazioni di funzione

void SommoUno(int i)
{
 i=i+1;
 cout << "Sono dentro a SommoUno " << i << endl;
}

void SommoUnoRif(int& i)
{
 i=i+1;
 cout << "Sono dentro a SommoUnoRif " << i << endl;
}
```

```
skin:codice odone$./a.out
All'inizio 0
Sono dentro a SommoUno 1
Dopo il passaggio per valore 0
Sono dentro a SommoUnoRif 1
Dopo il passaggio per
riferimento 1
skin:codice odone$
```



# Passaggio dei parametri

- Nel **passaggio per riferimento** *il parametro attuale può essere solo una variabile*
- Nel **passaggio per valore** *il parametro attuale può anche essere un'espressione arbitrariamente complessa*

**vi è chiaro il perché?**

- `void DoThis(int& count, float val)`  
....

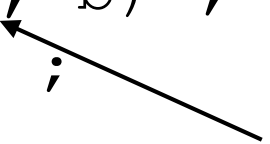
- Chiamate valide:

- `DoThis(someint, somefloat);`
- `DoThis(someint, 9.78);`
- `DoThis(someint, 3.*sqrt(v));`

# Esempio

```
void swap (int& x, int& y) {
 int tmp ;
 tmp = x ; x = y ; y = tmp ;
}
```

```
int main () {
 int a = 1, b = 2 ;
 swap (a+b, b) ;
 return(0) ;
}
```



errore!

# Ambito di visibilità: le variabili locali alle funzioni

- una funzione è un blocco indipendente
- al suo interno possiamo definire variabili locali
- tali variabili saranno visibili solo nell'intervallo di visibilità della funzione

# Ancora sull'ambito di visibilità

- **Esempio:**

```
void SomeFunc(float);
const int A=17;
int b,c;

int main()
{
 b=4;
 c=6;
 SomeFunc(42.8);
 return 0;
}
```

Regole di precedenza: se una funzione dichiara un id locale con lo stesso nome di un id globale, all'interno della funzione l'id locale ha la precedenza

- ```
void SomeFunc(float c)  
{  
    float b;  
    b=2.3;  
    cout << A << b << c<<end;  
}
```

impedisce l'accesso alla globale int c



impedisce l'accesso alla globale int b



Variabili globali - side effect

- se più funzioni accedono alla stessa *variabile globale* e qualcosa va storto è più difficile tracciare il problema e risolverlo
- se una stessa variabile deve essere acceduta da più funzioni è meglio aggiungerla come parametro a tutte le funzioni (se necessario, passandola per riferimento)

Variabili globali - side effect

```
#include <iostream>

using namespace std;

int t;
void funct(int&);

int main ()
{
    t=15;
    cout << "in main: t = " << t << endl;
    funct(t);
    cout << "in main after funct: t = " << t << endl;
    return 0;
}

void funct(int& a)
{
    cout << "1. in funct: a = " << a << " and t " << t << endl;
    a=a+12;
    cout << "2. in funct: a = " << a << " and t " << t << endl;
    t=t+13;
    cout << "3. in funct: a = " << a << " and t " << t << endl;
}
```

in main: t = 15
in funct: a = 15 and t 15
in funct: a = 27 and t 27
in funct: a = 40 and t 40
in main after funct: t = 40

funzioni che restituiscono più di un valore

- Talvolta possiamo aver bisogno di progettare una funzione che produca (calcoli o aggiorni) più valori
- **In questo caso si consiglia di definirla come funzione void con più parametri passati per riferimento**

Funzioni parziali

$$f : X \rightarrow Y$$

- Una funzione parziale non è definita su tutti gli elementi del dominio X
- Esempio

```
int safe_divide(int x, int y) {  
    if (y != 0) return x/y;  
}... altrimenti?
```


Funzioni parziali

- Altro esempio:

Definiamo la funzione

```
int area(int length, int width){  
    // length > 0 e width > 0 sono i lati di un rettangolo  
    // la funzione ne calcola l'area  
    return length*width;  
}
```

che cosa succede se qualcuno chiama area su argomenti negativi o nulli?

```
area(0,6)==0//ok, rettangolo degenero, ma può aver senso  
area(-2,3)==-6//orrore: un'area negativa! va beh, mi accorgo dell'errore  
area(-2,-23)==46//da qui in poi funziona tutto anche se non ha senso  
area(0,-56)==0//questo non è degenero è proprio senza senso
```

Come gestire la parzialità

- Tecnica “vecchia” : restituire uno specifico valore

```
int area(int length, int width){  
    // length > 0 e width > 0 sono i lati di un rettangolo  
    // la funzione ne calcola l'area  
    // su input sbagliato restituisce -1  
    if (length <= 0 || width <= 0) return -1;  
    return length*width;  
}
```

In questo caso la gestione errore scaricata sulla funzione chiamante

```
int z = area(x, y);  
if (z<0) {...};
```

Problemi

- Che succede se il chiamante si dimentica di fare il controllo sul risultato?
- Per alcune funzioni non esiste un valore *cattivo* da usare per rappresentare l'errore (per esempio nel caso della divisione)

Un utile meccanismo linguistico (cenni)

In C++ (come in Java, C#...in generale nei linguaggi moderni) esiste un apposito meccanismo linguistico

In casi anomali (o sbagliati) si *solleva un'***eccezione**
–termina l'esecuzione della funzione corrente

- un'eccezione viene **automaticamente** individuata durante l'esecuzione
–non ci si può *dimenticare* di verificare se c'è stato un errore

- l'eccezione può essere **catturata** e **gestita** dal chiamante
–stesse potenzialità del codice d'errore o del valore *sbagliato*

- un'eccezione non gestita *galleggia* al livello di chiamata più alto
–in caso di chiamate di funzioni annidate
(e.g. main, che chiama read_set, che chiama read_number..)

- se nessuno la gestisce il programma termina con errore

Un utile meccanismo linguistico (cenni)

- **Sollevare un'eccezione (throw an exception)**

```
int area(int length, int width) {  
    // length > 0 e width > 0 sono i lati di un rettangolo  
    // la funzione ne calcola l'area  
    // su input sbagliato solleva una stringa  
    string err = "parametro con valore non positivo";  
    if (0 >= length || 0 >= width) throw err; // esce da area  
    return length*width;  
}
```

- Tutto quello che segue un throw (fino alla fine della funzione) **non** viene eseguito
 - come se ci fosse un return
 - non serve else

Il **tipo** sollevato indica che **tipo** di errore c'è stato

Il **valore** sollevato serve per passare dei **dati**

Un utile meccanismo linguistico (cenni)

Catturare un'eccezione (catch an exception)

qualsiasi tipo
ma lo stesso
quando si solleva
e si cattura

```
int area(int length, int width) {  
    // length > 0 e width > 0 sono i lati di un rettangolo  
    // la funzione ne calcola l'area  
    // su input sbagliato solleva una stringa  
    string err = "parametro con valore non positivo";  
    if (0 >= length || 0 >= width) throw err;  
    return length*width;  
}
```

```
int main() {  
    int x, y;  
    try{  
        cout << "inserisci lunghezza e altezza del rettangolo";  
        cin >> x >> y;  
        cout << "L'area del rettangolo e': " << area(x, y);  
    }  
    catch (string& err) {  
        cout << err;  
        cerr << "oops! Bad area calculation - fix program\n";  
    }  
}
```

per riferimento per
evitare copie inutili

Un utile meccanismo linguistico (cenni)

- Il meccanismo delle eccezioni si applica a tutte le situazioni anomale (compresi errori, gestione dei casi di out of range...)
- Vantaggi
 - meccanismo generale adatto in ogni caso
 - meccanismo automatico
 - meccanismo flessibile
 - si separano logicamente
 - codice da eseguire se tutto va bene
 - individuazione dell'errore (eventualmente diversificato)
 - gestione dell'errore
- Non è una panacea
 - bisogna sempre capire quali situazioni sono *strane*
 - bisogna sempre capire che cosa fare per risolvere situazioni strane

Gestire errori

Quando si chiama una funzione che può generare errori

1. si inizia un blocco try{...}
2. nel blocco si fa la chiamata
3. nel blocco va tutta la logica di esecuzione di quando **non** ci sono errori
4. dopo il blocco, per ciascun *tipo* possibile di errore
 - a) un blocco catch(<tipo di eccezione>& <nome>)
{codice per gestire l'errore}
 - b) gestione può voler dire
 - risolvere e riuscire ad ottenere il risultato (es. chiedere un altro input all'utente e riprovare)
 - segnalare l'errore all'utente e chiudere il programma con codice d'errore
 - sollevare un'eccezione di altro tipo (lo vedrete nei prossimi corsi, a IP non vi capiterà)
5. Se non volete gestire un certo errore, non mettete il blocco catch
 - automaticamente passerà a chi vi sta chiamando e sarà sua responsabilità decidere se/come gestirlo

Nota finale (tanti tipi di eccezione)

Scrivere una funzione con argomenti interi n e d , con d compreso fra 0 e 9, che restituisce il più grande numero compreso fra $-|n|$ e $|n|$ che nella sua rappresentazione in base 10 usa la cifra d .

Che cosa può andare male in una chiamata di questa funzione?

- Possibili errori

- d troppo piccolo (minore di 0)
- d troppo grande (maggiore di 9)
- non esiste il valore di ritorno (esempio $d==8$ e $n==3$)

- A seconda di come cataloghiamo gli errori potremmo sollevare eccezioni diverse.