Funzioni: passaggio parametri e moduli

Alessandra Giordani
agiordani@disi.unitn.it
Lunedì 29 aprile 2013
http://disi.unitn.it/~agiordani/

Abbiamo già visto…

- Calcolo delle radici di equazioni di 2°grado
  - Implementazione nel main()
  - Utilizzo della libreria *math.h* per *sqrt()*

```
#include <math.h>
```

  - in compilazione dobbiamo aggiungere il flag -lm alla riga di comando

```
gcc -lm radici.c -o radici
./radici
```

Oggi faremo (ovvero farete)

→ Calcolo delle radici di equazioni di 2°grado utilizzando una funzione *delta(a,b,c)* dichiarata e definita separatamente

- Introdurremo le funzioni, i parametri attuali e formali e la visibilità delle variabili
- Funzioni ricorsive
- Ordinamento di un vettore

Breve ripasso su GCC

GCC - GNU C Compiler: esempi di invocazione

- `$ gcc miofile.c`
  - Compila il codice sorgente in `miofile.c`
  - Se il file non contiene errori, crea il file `a.out`
  - `a.out` contiene il codice eseguibile (macchina) del codice sorgente `miofile.c`
- `$ gcc -o fileout miofile.c`
  oppure
  `$ gcc miofile.c -o fileout`
  - Compila il codice sorgente in `miofile.c`
  - Se il file non contiene errori, crea il file `fileout`
  - `fileout` contiene il codice eseguibile (macchina) del codice sorgente `miofile.c`
- Path del file di input e di output assoluti o relativi
- Il nome del file di input deve terminare con `.c`

Fasi della compilazione

- Preprocessing (opzione "-E" di GCC):
  - Rimozione dei commenti
  - Interpretazione di speciali "direttive", ad es:
    - `#include <file.h>`
      include il contenuto di `file.h` nel codice sorgente
    - `#define x y`
      sostituisce nel codice seguente tutte le occorrenze di `x` con `y`
- Compilazione: traduce C in assembly (opzione "-S" di GCC)
- Assemblaggio: traduce assembly in codice oggetto (i.e. linguaggio macchina, opzione "-c" di GCC)
- Linking: combina codice oggetto da piú file per creare un eseguibile. Questo é necessario quando:
  - si usano funzioni di libreria;
  - il codice sorgente é suddiviso in piú file

Per saperne di più: `$ man gcc`

Le funzioni

Una funzione é un insieme di istruzioni che realizzano un algoritmo. È caratterizzata dalla sua signature:

- parametri formali: il tipo dei parametri dell'algoritmo (es: funzione che calcola il delta di un'eq di secondo grado, deve conoscere i coefficienti dell'equazione)
- nome: un nome arbitrario per identificarla (es: `delta`)
- ritorno: il tipo del risultato ritornato dalla funzione (es: `double`)

Una funzione senza argomenti é detta procedura

Ciclo di vita di una funzione: dichiarazione, implementazione, invocazione

Dichiarazione di una fuzione

Informa il compilatore che esiste una funzione con una certa signature:
`<tipo di ritorno> <nome> ( <parametri formali> );`

Le regole per la validità dei nomi di funzioni sono le stesse che nel caso delle variabili. Alcuni esempi:

```
double delta(double a, double b, double c);
int max(int x, int y);
double radice_quadrata(double x);
```

- La dichiarazione di una funzione non dice nulla sulla sua semantica (cosa fa) né sulla sua implementazione (come lo fa)
- black-box reuse: se conosciamo la signature di una funzione e sappiamo cosa fa (documentazione), possiamo usarla anche ignorandone l'implementazione

Invocazione di una funzione

L'invocazione di una funzione avviene attraverso la sostituzione dei parametri formali con una tupla compatibile di parametri attuali. Esempi:

```
double delta(double a, double b, double c);

... /* da qualche parte l'implementazione della funzione
       delta deve essere definita */

double coeff_x2 = 8, coeff_x1 = 4, coeff_x0 = 2;
double d1 = delta(coeff_x2, coeff_x1, coeff_x0);
double d2 = delta(8, 4, 2); /* stesso risultato */
double d3 = delta(8/2, 5.6, 2.56);
double d4 = delta(coeff_x2*coeff_x1, coeff_x3, 2*coeff_x2)

char x = delta(8, 4, 2); /* errore: ritorno non compatibile */

/* errore, argomenti non compat. */
double y = delta("ciao", 3, 8);

/* se non serve, é ok ignorare il valore di ritorno */
delta(6, 10, 567.0);
```

Implementazione di funzione

Specifica le istruzioni associate ad un nome di funzione:

```
<tipo di ritorno> <nome> ( <parametri formali> )
{ // inizio delle istruzioni associate alla funzione
   <istruzione1>;
   <istruzione2>;
   ...
   return <valore compatibile con tipo di ritorno>;
} // fine delle istruzioni associate alla funzione
```

Le istruzioni possono essere: dichiarazioni di variabili, assegnamenti, espressioni, chiamate di funzione, etc.
Esempio:

```
/* Restituisce la somma dei primi n numeri interi */
int somma_primi_n_interi(int n)
{
   int result = n * (n+1) / 2;
   return result;
}
```

Esempio di una procedura

- Dichiarazione
- Utilizzo
- Definizione

```c
#include <stdio.h>

void stampamsg (int num) ;
int num = 3; //variabile globale

int main()
{
    int num = 6; //variabile locale
    stampamsg (12);
    printf("La variabile num vale: %d\n" , num);
    return 0;
}

void stampamsg (int num)
{
 printf("Il parametro num vale: %d\n" , num);
}
```

Esempio di una funzione (1/3)

- Abbiamo visto un algoritmo per il calcolo delle radici di un equazione di 2°grado.
- Questo programma è una possibile implementazione

```
#include <stdio.h>
#include <math.h>
int main()
{
float x1,x2,a=2,b=3,c=1,delta;

delta=b*b-4*a*c;
if (delta<0)
      printf("Non esistono radici reali\n");
      else
      {
      if (delta==0)
                x1=x2=-b/(2*a);
           else
                {
                x1=(-b+sqrt(delta))/(2*a);
                x2=(-b-sqrt(delta))/(2*a);
                }
      printf("x1=%f,x2=%f\n",x1,x2);
      }
return 0;
}
```

Esempio (2/3)

- Dichiarazione
- Utilizzo
- Definizione

```c
#include <stdio.h>
#include <math.h>
double delta (double a,double b,double c);

int main()
{
float x1,x2,a=2,b=3,c=1,d;
d=delta(a,b,c);
if (d<0)
      printf("Non esistono radici reali\n");
      else
      {
      if (d==0)
                x1=x2=-b/(2*a);
           else
                {
                x1=(-b+sqrt(d))/(2*a);
                x2=(-b-sqrt(d))/(2*a);
                }
      printf("x1=%f,x2=%f\n",x1,x2);
      }
return 0;
}

double delta (double a,double b,double c)
{
 double res;
 res=b*b-4*a*c;
 return res;
}
```

Esempio (3/3)

Definizione in delta.c

```c
#include "delta.h"

double delta (double a,double b,double c)
{
 double res;
 res=b*b-4*a*c;
 return res;
}
```

Compilazione sorgenti crea file oggetto:

```
gcc -c delta.c -o delta.o
gcc -c radici3.c -o radici3.o
```

Linker di file oggetto crea eseguibile

```
gcc -lm radici3.o delta.o -o
   radici3
./radici3
```

Dichiarazione in delta.h

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

double delta (double a,double b,double c);
```

Utilizzo in radici3.c

```c
#include "delta.h"

int main()
{
float x1,x2,a=2,b=3,c=1,d;

d=delta(a,b,c);
if (d<0)
      printf("Non esistono radici reali\n");
      else
      {
      if (d==0)
                x1=x2=-b/(2*a);
           else
                {
                x1=(-b+sqrt(d))/(2*a);
                x2=(-b-sqrt(d))/(2*a);
                }
      printf("x1=%f,x2=%f\n",x1,x2);
      }
return 0;
}
```

Fattoriale di n nel main

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

main()
{
        int n;
        unsigned long fatt;

        printf("Introdurre un intero positivo:\n");
        scanf("%d", &n);
        fatt = 1;
        printf("%d! = ", n);
        while (n > 1)
        {
                fatt = fatt * n;
                n--;
        }
        printf("%ld\n", fatt);
        exit(0);
}
```

Funzione fattoriale

```
int fatt (n)
int n;
{
        int t, result;

        result = 1;
        for (t=2; t<=n; t++)
              result *= t;
        return result;
}
```

Funzione non ricorsiva

```
int rfatt (n)
int n;
{
        if  (n==0)
              return 1;
        else
              return (n*rfatt(n–1));
}
```

Funzione ricorsiva

Passaggio per Valore

- Viene effettuata una copia della variabile passata per parametro
- Non modifica il valore della variabile passata
- Come posso modificare **a**?

```c
#include <stdio.h>

int func(int a, int b) {
    printf("&a = %p\n", &a);
    a = a + b;
    return a;
}

int main() {

    int value = 10;
    int result;

    printf("prima vale: %d\n", value);
    printf("indirizzo = %p\n", &value);
    result = func(value, 10);
    printf("dopo vale: %d\n", value);
    printf("risultato: %d\n", result);

    printf("prima vale: %d\n", value);
    result = func(value, 15);
    printf("dopo vale: %d\n", value);
    printf("risultato: %d\n", result);

    return(0);
}
```

Passaggio per Riferimento

- Modifica il valore puntato dalla variabile “indirizzo” passato come parametro
- Vediamo come vengono utilizzati i puntatori…

```c
#include <stdio.h>

int func(int *a, int b) {
    printf("a: %p\n", a);
    *a = *a + b;
    return *a;
}

int main() {

    int value = 10;
    int result;

    printf("prima vale: %d\n", value);
    printf("indirizzo: %p\n", &value);
    result = func(&value, 10);
    printf("dopo vale: %d\n", value);
    printf("risultato: %d\n", result);

    printf("prima vale: %d\n", value);
    result = func(&value, 15);
    printf("dopo vale: %d\n", value);
    printf("risultato: %d\n", result);

    return(0);
}
```

Puntatori - 1

- Un puntatore é una variabile che rappresenta un indirizzo di memoria
- Usati principalmente per: memorizzare indirizzi di variabili, passaggio per riferimento, manipolazione di array
- I puntatori in C sono "tipati": portano con sé informazione relativa al tipo di dato a cui puntano
- Dichiarazione di un puntatore: operatore unario "*" (solo in dichiarazione!!)

```
int a, b, c; /* dichiaro tre variabili intere */
int * x;     /* dichiaro un puntatore a intero */
double * y;  /* dichiaro un puntatore a double */
```

- Una variabile puntatore occupa in memoria un numero di byte sufficiente a rappresentare tutti gli indirizzi di memoria indirizzabili dal bus $\Rightarrow$ 4 byte in una architettura a 32 bit
- Lo spazio occupato da un puntatore non dipende dal tipo!

Puntatori - 2

- N.B.: * nella dichiarazione é associato a destra (al nome di variabile) e non a sinistra (al tipo). Se voglio dichiarare due puntatori ad intero sulla stessa riga devo scrivere

```
int * j, * k;
```

- Se invece scrivessi:

```
int * j,k;
```

  dichiarerei:

  - Un puntatore ad intero (j)
  - Una variabile intera (k)

- Domanda: quale é l'effetto della seguente istruzione?

```
double a, * b, c, d;
```

Manipolazione dei puntatori

- Operatore indirizzo di (unario): &
  - Variabile ⇒ Puntatore
  - Resituisce l'indirizzo (reference) di una variabile (e ci permette di salvarla in un puntatore!)
  - Da non confondere con l'AND logico (binario, &&) e l'AND bit a bit (binario, &)
- Operatore di dereferenziazione (unario, nelle espressioni): *
  - Puntatore ⇒ Valore
  - Restituisce il valore della variabile che si trova ad un certo indirizzo (e ci permette di manipolare il valore di una variabile di cui conosciamo l'indirizzo!)
  - Da non confondere con l'operatore di moltiplicazione (binario) o con l'operatore * nella dichiarazione di una variabile di tipo puntatore

Attenzione:

- La dichiarazione:

```
int * a;
```

- Significa: dichiaro un puntatore ad intero e lo chiamo a
- Il nome del puntatore é solo a, senza * !!!

- a rappresenta un indirizzo

- *a é il valore memorizzato all'indirizzo a

Esempio:

```
int a = 100, b = 50, *p;

p = &a;          // ora p punta ad a

int c = *p;      // assegno a c il valore puntato da p: 100

p = &b;          // ora p punta a b

c = *p + 10;     // assegno a c il valore puntato da p
                 // aumentato di 10, cioé 60

*p = *p + 5;     // equivalente a: b = b + 5, b vale 55

int * q = &a;

int d = *p * *q; // cioé: d = (*p) * (*q)
                 // d value 55*100 = 5500
```

Puntatori e printf

- La funzione printf permette di stampare un indirizzo mediante l'indicatore di formato %p:

```
int a = 100;
printf("a: %d, indirizzo: %p\n", a, &a);

int *b = &a;
printf("b: %p, indirizzo: %p\n", b, &b);
return 0;
```

- L'indirizzo viene stampato in formato esadecimale

L'accesso a una variabile puntata – 1

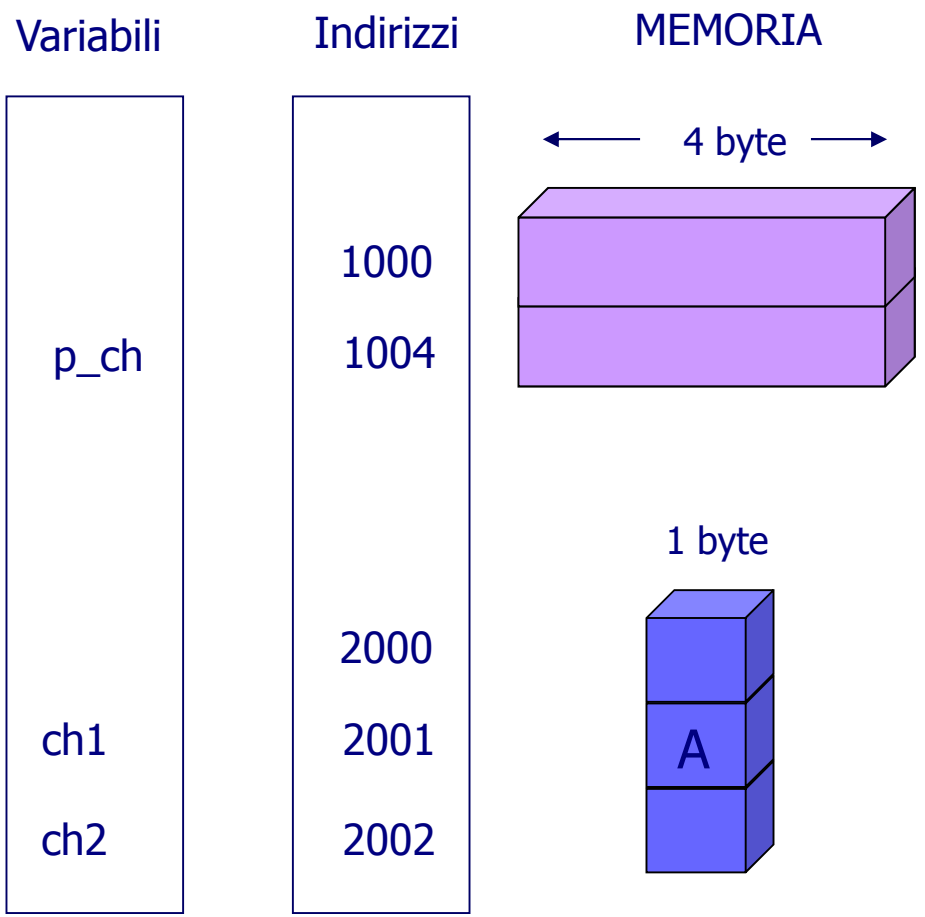
- Si usa l'asterisco * anche per accedere al valore che è memorizzato all'indirizzo di memoria contenuto in una variabile puntatore

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

main()
{
    char *p_ch;
    char ch1='A', ch2;

    printf("L'indirizzo di p_ch è: %p\n", &p_ch);
    p_ch = &ch1;
    printf("Il valore contenuto in p_ch è %p\n, p_ch);
    printf("Il valore contenuto all'indirizzo \
            memorizzato in p_ch è: %c\n", *p_ch);
    ch2 = *p_ch;
}
```

L'accesso a una variabile puntata – 2

CODICE

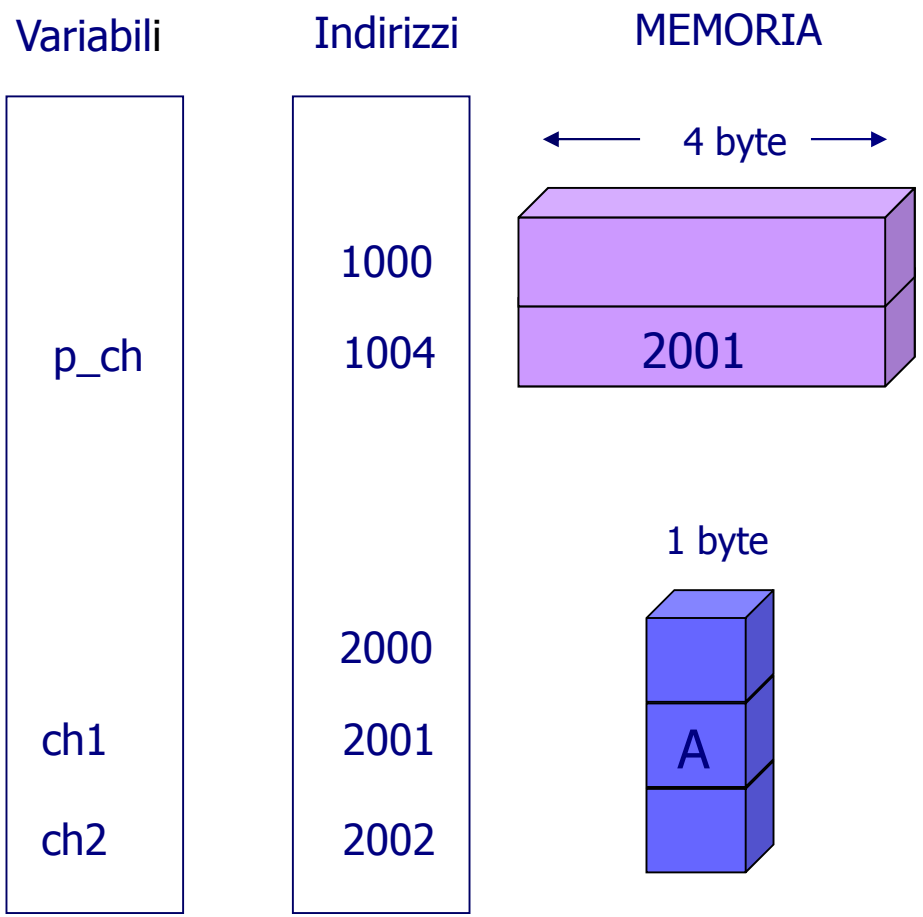
```
char *p_ch;
char ch1='A', ch2;
```



L'accesso a una variabile puntata – 3

CODICE

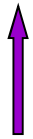
```
p_ch = &ch1;
```



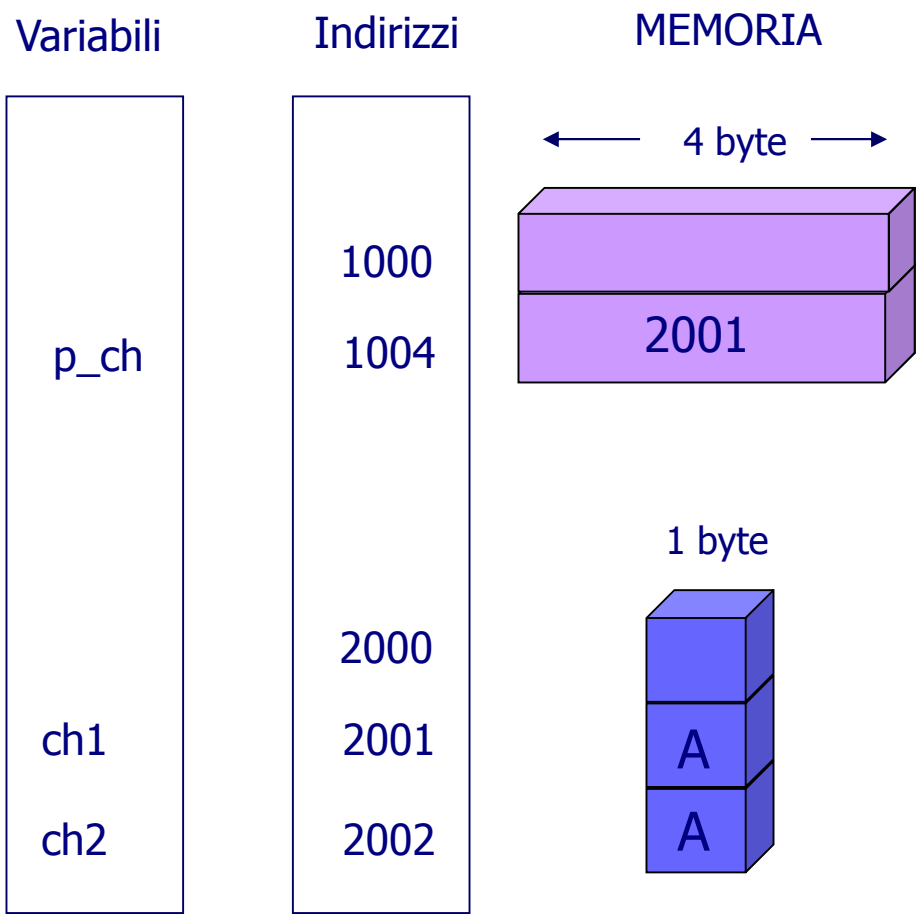
L'accesso a una variabile puntata – 4

CODICE

```
ch2 = *p_ch;
```



Prendi l'indirizzo memorizzato in **`p_ch`** e leggi il valore contenuto nella locazione di memoria identificata da tale indirizzo; copialo in **`ch2`**



Passaggio di parametri per indirizzo

- Le stringhe, ed in generale i vettori e matrici, non possono essere passate per valore come parametro di una funzione!
- Bisogna passarle per riferimento, utilizzando il puntatore al primo elemento dell'array (nome array!)

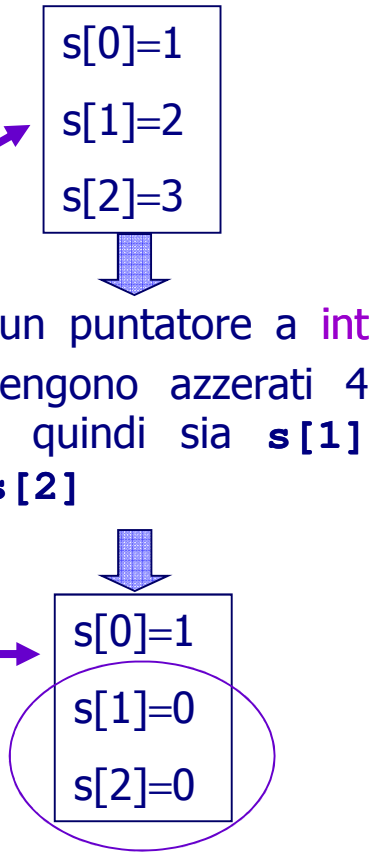
```
char  stringa[] = "prova";
int lunghezza = conta(stringa);
```

- La funzione sarà così dichiarata

```
int conta(char *stringa)
{
   …
}
```

Il passaggio di puntatori come argomenti di funzione - 1

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

void clr(p)
int *p;
{
   *p = 0  /* Memorizza 0 alla locazione p */
}
main()
{
   static short s[3] = {1,2,3};

   clr(&s[1]);  /* Azzera l'elemento 1 di s[] */
   printf("s[0]=%d\ns[1]=%d\ns[2]=%d\n", s[0],s[1],s[2]);
   exit(0);
}
```



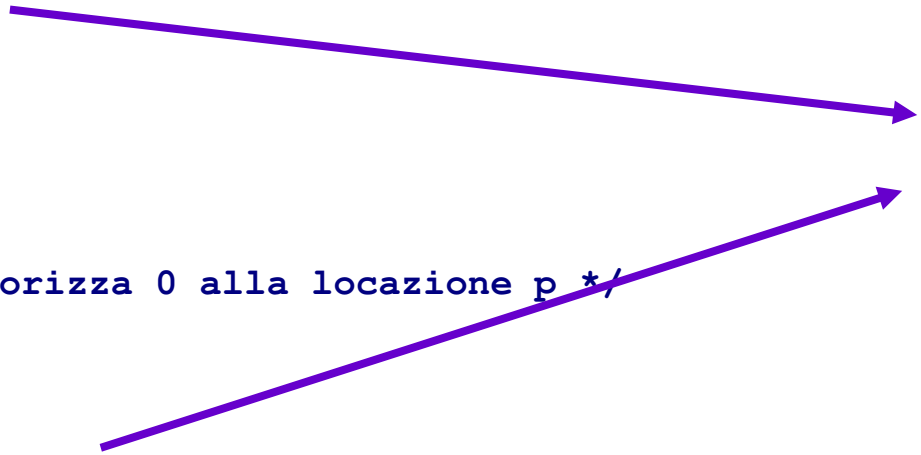
Il passaggio di puntatori come argomenti di funzione – 2

- Il compilatore segnala i tentativi di utilizzo congiunto di puntatori di tipi diversi
- Un'eccezione alla regola è costituita dall'uso di puntatori come argomenti di funzione: in mancanza di un prototipo, il compilatore non effettua controlli per verificare la corrispondenza di tipo fra parametro attuale e parametro formale

  ⇨ si possono produrre risultati inconsistenti nel caso di parametri di tipo disomogeneo
- Il prototipo di una funzione è una dichiarazione di funzione antecedente alla sua definizione: permette al compilatore di compiere il controllo sui tipi degli argomenti che vengono passati alla funzione

Prototipo di una funzione

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

void clr(int *);

void clr(p)
int *p;
{
   *p = 0  /* Memorizza 0 alla locazione p */
}
main()
{
   static short s[3] = {1,2,3};

   clr(&s[1]);  /* Azzera l'elemento 1 di s[] */
   printf("s[0]=%d\ns[1]=%d\ns[2]=%d\n", s[0],s[1],s[2]);
   exit(0);
}
```



p è un puntatore a int

incompatibile con un puntatore a short int!

Implementare la funzione conta()!

- Completare l'implementazione della funzione che conta la lunghezza della stringa

```
int conta(char *stringa)
{
    int l = 0;
   …
   return
}
```

Le funzioni di libreria per le stringhe `strlen()`

- La funzione **`strlen()`**, restituisce il numero di caratteri che compongono una stringa (escluso il carattere nullo)
- Poiché nell'espressione **`*str++`** i due operatori hanno la stessa precedenza ed associatività destra, l'espressione viene analizzata dal compilatore nel modo seguente:
  - Valutazione dell'operatore di incremento postfisso; il compilatore passa **`str`** all'operatore successivo e lo incrementa solo al termine della valutazione dell'espressione
  - Valutazione dell'operatore *, applicato a **`str`**
  - Completamento dell'espressione, con l'incremento di **`str`**

```
int strlen(str)
char *str;
{
  int i;

  for (i=0; *str++; i++)
       ; /* istruzione vuota */
  return i;
}
```

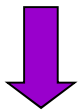
printf() di stringhe

- All'interno della stringa template dobbiamo usare %s.
- Come altro parametro dobbiamo passare il nome della variabile, che usato da solo è il puntatore al primo elemento della stringa
- printf() stamperà quel elemento fino al carattere `'\0'`
  - `char str[] = "stringa";`
  - `printf("%s",str); //stampa` **`stringa`**
  - `printf("%s",str+2); // stampa` **`ringa`**
  - `printf("%s",str[2]); // non corretto!`
  - `printf("%c",str[2]); // stampa` **`r`**

La dichiarazione di tipo: typedef

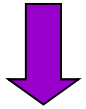
- La dichiarazione di tipo deve apparire in un programma prima che il tipo venga adoperato per la dichiarazione di variabili
- Le dichiarazioni di tipo sono particolarmente utili nella definizione di tipi composti
- Avvertenza: typedef e #define non sono equivalenti…

```
#define PT_TO_INT  int *

PT_TO_INT p1, p2;
```



```
int  *p1, p2;
```

```
typedef  int *  PT_TO_INT;

PT_TO_INT p1, p2;
```



```
int  *p1, *p2;
```