SOTTOPROGRAMMI FUNZIONI IN C

Fondamenti di Programmazione 2021/2022

Francesco Tortorella

Nuove operazioni ?

- In qualunque linguaggio di programmazione il tipo di dati non specifica solo l'insieme dei valori che a questo appartengono, ma anche le operazioni su questi definite.
- Solo alcune di tali operazioni sono però rese disponibili dal linguaggio; le altre devono essere implementate dal programmatore

Come inserirle ?

- Definito l'algoritmo che realizza una particolare operazione, i costrutti visti finora ne permettono la codifica.
- Il problema è che il codice va replicato ogni qual volta quell'operazione è richiesta nel programma.
- Conseguenze:
  - **Minore leggibilità del codice**
  - **Maggiore probabilità di errori**
  - **Minore manutenibilità del codice**

Sottoprogrammi

- L'ideale sarebbe un meccanismo che permetta di arricchire il linguaggio con una istruzione che realizzi quell'operazione.
- Tale meccanismo dovrebbe dare al programmatore la possibilità di:
  - **specificare le istruzioni che realizzano l'operazione richiesta;**
  - **specificare i dati coinvolti;**
  - **specificare il nome con cui identificare l'operazione**.
- Questo meccanismo è realizzato tramite i **sottoprogrammi**.

Sottoprogrammi

- Un sottoprogramma è una particolare unità di codice che non può essere eseguita autonomamente, ma soltanto su richiesta del programma principale o di un altro sottoprogramma.
- Un sottoprogramma viene realizzato per svolgere un compito specifico (p.es. leggere o stampare gli elementi di un array, calcolare il valore di una particolare funzione matematica, ecc.) per il quale implementa un opportuno algoritmo.

Sottoprogrammi

- Per questo scopo, il sottoprogramma utilizza variabili proprie, alcune delle quali sono impiegate per scambiare dati con il programma dal quale viene attivato.
- Un sottoprogramma può essere attivato più volte in uno stesso programma o anche utilizzato da un programma diverso da quello per cui era stato inizialmente progettato.

Sottoprogrammi: definizione

- Nel definire un sottoprogramma è quindi necessario precisare operazione e flusso di dati
- Quale operazione il sottoprogramma realizza
- Qual è il flusso di dati tra il sottoprogramma ed il codice che lo ha attivato. In particolare:
  - Quali sono i dati in ingresso al sottoprogramma
  - Quali sono i dati in uscita dal sottoprogramma

Sottoprogrammi in C: Funzioni

- Una **funzione** è un particolare sottoprogramma che riceve in ingresso dei dati e produce in uscita un valore il quale non è assegnato ad uno dei parametri, ma viene attribuito al nome stesso della funzione.
- Il valore fornito in uscita è calcolato tramite le istruzioni presenti nella funzione eseguite sui valori dei dati forniti in ingresso.

Struttura di una funzione

Tipo del valore restituito

Parametri di ingresso

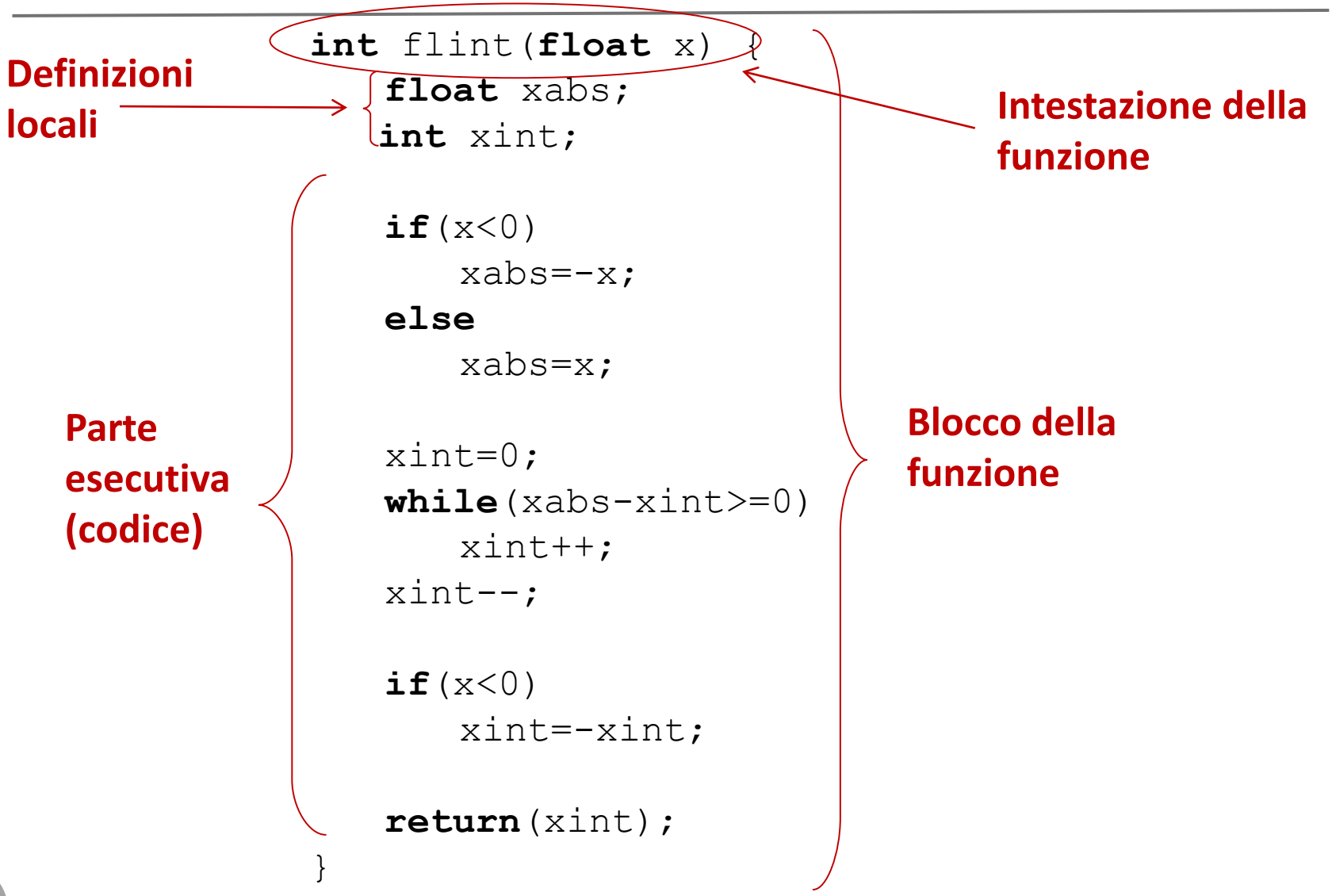
```
int flint(float x) {
    float xabs;
    int xint;

    if(x<0)
        xabs=-x;
    else
        xabs=x;

    xint=0;
    while(xabs-xint>=0)
        xint++;
    xint--;

    if(x<0)
        xint=-xint;

    return(xint);
}
```

Istruzione di ritorno

Struttura di una funzione

- Sono riconoscibili due parti:
  - **l'intestazione**
  - **il blocco**
- L'intestazione della funzione riporta le informazioni principali relative alla funzione: nome, tipo restituito, parametri di ingresso.
- Il blocco è costituito da:
  - una parte dichiarativa (variabili locali)
  - una parte esecutiva (istruzioni)

Struttura di una funzione




```
int flint(float x) {
    float xabs;
    int xint;

    if(x<0)
        xabs=-x;
    else
        xabs=x;

    xint=0;
    while(xabs-xint>=0)
        xint++;
    xint--;

    if(x<0)
        xint=-xint;

    return(xint);
}
```

Nome della funzione

- Il nome identifica univocamente la funzione
- I nomi delle funzioni hanno gli stessi vincoli dei nomi delle variabili. Il nome deve cominciare con una lettera che può essere seguita da una combinazione di lettere, cifre, underscore.

Parte esecutiva

- La parte esecutiva contiene l'insieme di istruzioni che implementa l'operazione che la funzione deve realizzare.
- Le istruzioni lavorano sull'insieme formato dai parametri di ingresso e dalle variabili definite all'interno.
- Le istruzioni possono essere costrutti di qualunque tipo (calcolo e assegnazione, I/O, selezioni, cicli, commenti, linee vuote, chiamate di altre funzioni).
- Al termine c'è una istruzione di **return** il cui scopo è di:
  - terminare l'esecuzione della funzione;
  - restituire il valore tra parentesi come valore della funzione.

```
int flint(float x) {
    float xabs;
    int xint;

    if(x<0)
        xabs=-x;
    else
        xabs=x;

    xint=0;
    while(xabs-xint>=0)
        xint++;
    xint--;

    if(x<0)
        xint=-xint;

    return(xint);
}
```

Funzioni: attivazione

- L'esecuzione delle istruzioni di una funzione è provocata da una particolare istruzione del programma che lo attiva (istruzione di *chiamata*, per cui il programma è anche detto *chiamante*).
- Ciò determina la sospensione dell'esecuzione delle istruzioni del programma chiamante, che riprenderà dopo l'esecuzione dell'ultima istruzione del sottoprogramma (tipicamente, un'istruzione di *ritorno*).

Funzioni: flusso dei dati

- Il passaggio dei dati in ingresso dal programma chiamante alla funzione avviene attraverso una lista di variabili elencate nell'intestazione della funzione, dette **argomenti** o **parametri formali** della funzione. Esse sono destinate ad ospitare i dati di ingresso della funzione.
- Con la istruzione di chiamata, il programma chiamante fornisce alla funzione una lista di **parametri effettivi**, costituiti dai valori su cui la funzione deve *effettivamente* operare.
- La corrispondenza tra parametri effettivi e formali è fissata per ordine.

Struttura di una funzione

Tipo del valore restituito

Parametri di ingresso

```
int flint(float x) {
    float xabs;
    int xint;

    if(x<0)
        xabs=-x;
    else
        xabs=x;

    xint=0;
    while(xabs-xint>=0)
        xint++;
    xint--;

    if(x<0)
        xint=-xint;

    return(xint);
}
```

Istruzione di ritorno

Chiamata di una funzione

- La chiamata di una funzione avviene all'interno di una espressione in cui compare il nome della funzione seguito dai parametri effettivi tra parentesi tonde ().
- Nell'espressione la funzione partecipa fornendo un valore del tipo restituito.
- La valutazione dell'espressione avvia l'esecuzione della funzione.
- Come parametri effettivi possono essere presenti anche delle espressioni (ovviamente, del tipo assegnato al parametro formale corrispondente).

Chiamata di una funzione

```
#include <stdio.h>
int main() {
    float a,b;
    int ai,bi;

    printf("a: "); scanf("%f",&a);
    printf("b: "); scanf("%f",&b);

    ai=flint(a);
    bi=flint(b);

    printf( "La parte intera di %f e' %d\n", a, ai);
    printf( "La parte intera di %f e' %d\n", b, bi);

    printf( "Somma delle parti intere: %d\n ", ai+bi);
    printf( "Parte intera della somma: %d\n ", flint(a+b));

    return(0);
}
```

Esecuzione di una funzione

1. Nel programma chiamante, la valutazione di un'espressione attiva la chiamata della funzione;
2. All'atto della chiamata, i parametri effettivi vengono valutati ed assegnati ai rispettivi parametri formali;
3. L'esecuzione del programma chiamante viene sospesa e il controllo viene ceduto alla funzione;
4. Inizia l'esecuzione della funzione: i parametri formali sono inizializzati con i valori dei parametri effettivi;
5. Le istruzioni della funzione sono eseguite;
6. Come ultima istruzione viene eseguito un `return` che fa terminare l'esecuzione della funzione e restituire il controllo al programma chiamante;
7. Continua la valutazione dell'espressione nel programma chiamante sostituendo al nome della funzione il valore restituito.

Librerie di funzioni

- Alcune funzioni sono già disponibili all'interno di librerie fornite con il compilatore e quindi non richiedono una definizione esplicita da parte dell'utente. Es.: **`sqrt(x)`**

Organizzazione del programma

```c
#include <stdio.h>

int flint(float x) {
     float xabs;
     int xint;

     if(x<0)
          xabs=-x;
     else
          xabs=x;
     xint=0;
     while(xabs-xint>=0)
          xint++;
     xint--;
     if(x<0)
          xint=-xint;
     return(xint);
}

int main() {
     float a,b;
     int ai,bi;

     printf("a: "); scanf("%f",&a);
     printf("b: "); scanf("%f",&b);
     ai=flint(a);
     bi=flint(b);

     printf( "La parte intera di %f e' %d\n", a, ai);
     printf( "La parte intera di %f e' %d\n", b, bi);
     printf( "Somma delle parti intere: %d\n ", ai+bi);
     printf( "Parte intera della somma: %d\n ", flint(a+b));

     return(0);
```

La definizione della funzione `flint` deve precedere il `main`

La funzione **`main`**

- Anche il blocco identificato da **`main`** è una funzione a tutti gli effetti.
- Particolarità di **`main`**:
  - viene chiamata dal Sistema Operativo all'atto dell'esecuzione del programma;
  - il flusso di dati avviene con il S.O., sia per i parametri effettivi in ingresso, sia per il valore restituito da **`return`**

Prototipo di una funzione

- Come per le variabili, anche le funzioni devono essere definite prima di essere usate.
- Nel caso ci siano più funzioni, il **`main`** andrebbe in fondo al file sorgente, rendendo meno leggibile il codice.
- In effetti, per poterle gestire correttamente, il compilatore ha bisogno solo delle informazioni presenti nell'intestazione della funzione.
- E' quindi possibile anticipare al **`main`** solo le intestazioni delle funzioni (prototipi) e inserire dopo il **`main`** le definizioni delle funzioni.
- Il prototipo è formato dall'intestazione della funzione terminato con ';':

```
int flint(float x);
```

Organizzazione del programma con i prototipi

```c
#include <stdio.h>
int flint(float x);

int main() {
      float a,b;
      int ai,bi;

      printf("a: "); scanf("%f",&a);
      printf("b: "); scanf("%f",&b);
      ai=flint(a);
      bi=flint(b);

      printf( "La parte intera di %f e' %d\n", a, ai);
      printf( "La parte intera di %f e' %d\n", b, bi);
      printf( "Somma delle parti intere: %d\n ", ai+bi);
      printf( "Parte intera della somma: %d\n ", flint(a+b));

      return(0);
}

int flint(float x) {
      float xabs;
      int xint;

      if(x<0)
            xabs=-x;
      else
            xabs=x;
      xint=0;
      while(xabs-xint>=0)
            xint++;
      xint--;
      if(x<0)
            xint=-xint;
      return(xint);
}
```

Prototipo della funzione

```c
int flint(float x);
```

Definizione della funzione

Procedure

- A volte le operazioni da implementare *non richiedono la produzione di un valore, ma l'esecuzione di un'azione*, come la stampa di valori o la modifica di variabili.
- In questi casi si può utilizzare un tipo diverso di sottoprogramma: la **procedura**.
- La chiamata della procedura avviene mediante una esplicita istruzione di chiamata, costituita dal nome della procedura seguito dalla lista dei parametri effettivi tra ().

Funzioni che restituiscono **`void`**

- In C una procedura viene definita come una funzione che non restituisce valori.
- Questo si realizza tramite il tipo **`void`**.
- Può essere presente l'istruzione **`return`**, che in questo caso ha solo la funzione di terminare l'esecuzione della funzione.

```
void stampa3int(int a,int b,int c) {
    int s;

    printf("Primo valore: %d\n",a);
    printf("Secondo valore: %d\n",b);
    printf("Terzo valore: %d\n",c);

    s=a+b+c;
    printf("Somma: %d\n",s);
    return;
}
```

Chiamata di una procedura

- La chiamata di una procedura avviene con un'istruzione apposita costituita dal nome della procedura seguito dalla lista dei parametri effettivi tra parentesi tonde ().
- L'attivazione viene realizzata nelle stesse modalità viste per la funzione.

```
void stampa3int(int,int,int);

int main() {
    int p,q,r;

    p=2; q=12; r=6;
    stampa3int(p,q,r);

    return(0);
}
```

Passaggio per valore

- La tecnica di corrispondenza tra parametri formali ed effettivi vista finora è detta passaggio per valore (o by value): il valore del parametro effettivo viene copiato nel parametro formale.
- Il parametro formale costituisce quindi una copia locale del parametro effettivo.
- Ogni modifica fatta sul parametro formale non si riflette sul parametro effettivo.

Passaggio per valore

Tutti gli esempi visti finora hanno utilizzato il passaggio per valore

```
void stampa3int(int a,int b,int c) {
     int s;

     printf("Primo valore: %d\n",a);
     printf("Secondo valore: %d\n",b);
     printf("Terzo valore: %d\n",c);

     s=a+b+c;
     printf("Somma: %d\n",s);
     return;
}
```

```
int flint(float x) {
     float xabs;
    int xint;

     if(x<0)
          xabs=-x;
     else
          xabs=x;

     xint=0;
     while(xabs-xint>=0)
          xint++;
     xint--;

     if(x<0)
          xint=-xint;

     return(xint);
}
```

Siamo soddisfatti?

Immaginiamo di voler realizzare una funzione che realizzi lo scambio tra due variabili definite nel programma chiamante

```
#include <stdio.h>

void swap(int a, int b) {
   int temp;

   temp = a;
   a = b;
   b = temp;
}

int main()
{
          int x, y;

          printf("Valore di x: "); scanf("%d", &x);
          printf("Valore di y: "); scanf("%d", &y);

          swap(x,y);

          printf("Nuovo valore di x: %d\n", x);
          printf("Nuovo valore di y: %d\n", y);
          return 0;
}
```

Che cosa succede? Perché?

Variabili, registri ed indirizzi

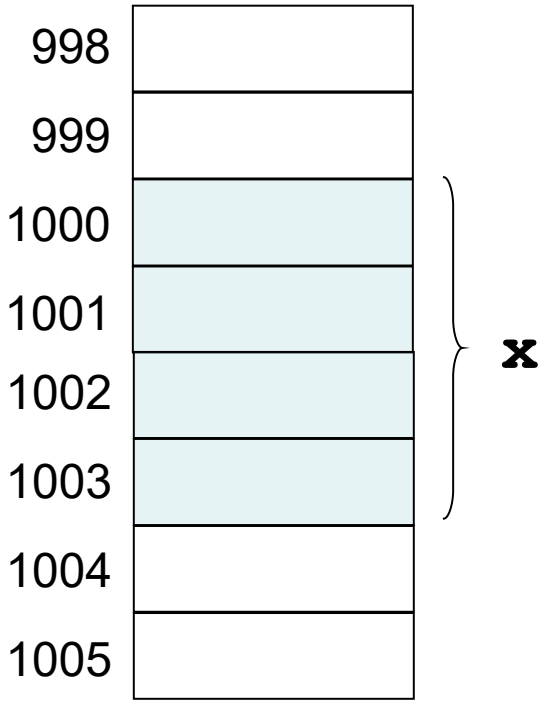
- La definizione di una variabile implica l'allocazione (da parte del compilatore) di registri di memoria. Il numero di registri allocati dipende dal tipo della variabile.
- Alla porzione di memoria allocata si accede tramite l'identificatore della variabile. Questo ci risparmia di preoccuparci in quale particolare locazione la variabile sia realmente allocata.
- È il compilatore a creare e gestire la corrispondenza tra identificatore della variabile e indirizzo della locazione in memoria.

Variabili, registri ed indirizzi

Esempio:

```
int x;
```

Con l'istruzione viene definita una variabile intera **x** che occupa 4 registri da 1 byte a partire dall'indirizzo 1000.



Variabili, registri ed indirizzi

- Il C dà la possibilità di accedere esplicitamente all'indirizzo di una variabile tramite l'operatore **`&`** (operatore di riferimento o di *reference*) prefisso all'identificatore della variabile.

`int x;` variabile **`x`**

`&x` indirizzo della variabile **`x`**

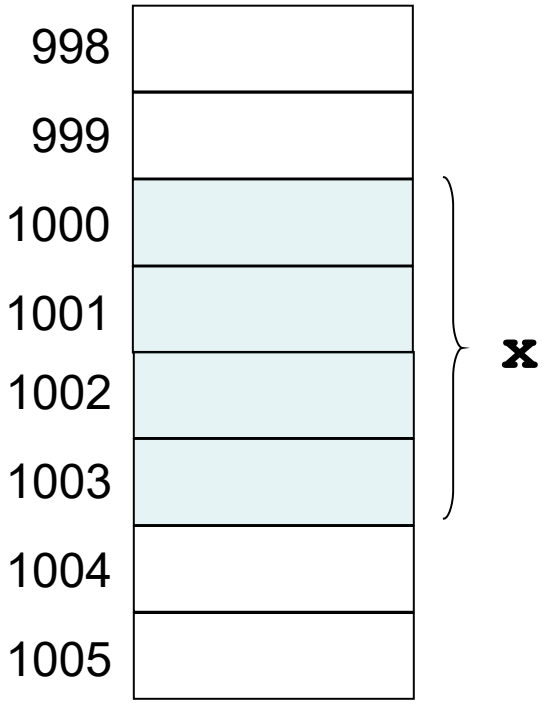
Variabili, registri ed indirizzi

- Esempio:
  `int x;`

- In questo caso `&x` sarà uguale a 1000.

ACHTUNG !

- L'operatore `&` si può applicare solo alle variabili (o, più precisamente, a *l-value*).



Variabili, registri ed indirizzi

```
#include <stdio.h>
using namespace std;

int main(){
    int x=3;

    printf("Valore di x: %d\n", x);
    printf("Indirizzo di x: %p\n", &x);
    return (0);
}
```

```
Valore di x: 3

Indirizzo di x: 0x7ffe94169bcc
```

indirizzo esadecimale

Puntatori

- Il C permette di definire delle variabili di tipo *puntatore* cui si possono assegnare gli indirizzi di variabili di un particolare tipo.
- La definizione di tali variabili (dette *puntatori*) richiede la specificazione del tipo "puntato", seguito da un '**`*`**'.
- Es.: definizione di un puntatore a **`int`**

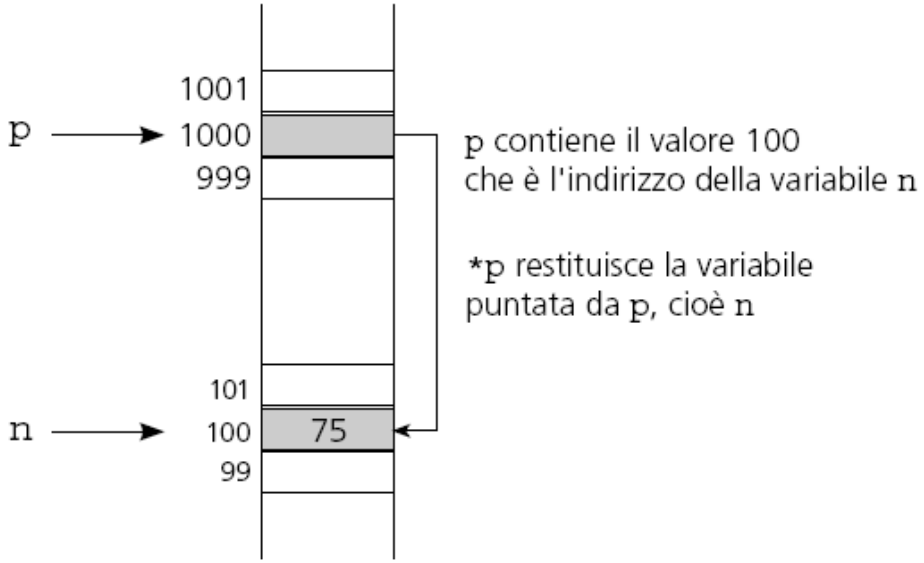
```
int* p;
```

Puntatori

- Di fatto una variabile di tipo puntatore al tipo T contiene l'indirizzo di memoria di una variabile di tipo T.

- Esempio:

```
int n;
int* p;


n = 75;
p = &n;
```

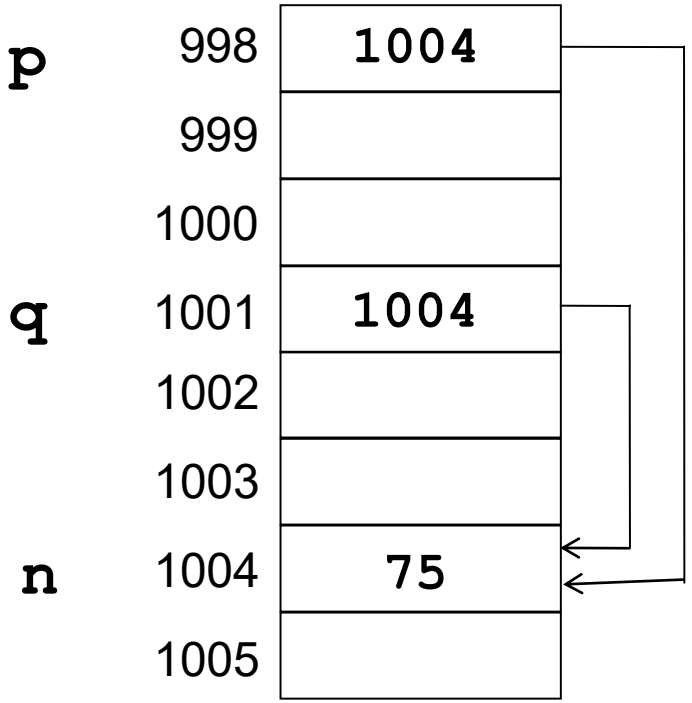


Puntatori

- È possibile fare assegnazioni tra puntatori.

- Esempio:

```
int n=75;
int *p,*q;

 p = &n;
 q = p;
```



- In questo modo due puntatori puntano alla stessa variabile.

Puntatori

- Tramite il puntatore è possibile accedere alla variabile puntata.
- Con l'operatore '*****' (operatore di indirezione o di *dereference*) prefisso all'identificatore della variabile puntatore è possibile accedere direttamente alla variabile puntata, sia in lettura che in scrittura.
- In questo modo si crea un alias della variabile che può essere modificata tramite il puntatore.

Puntatori

```
#include <stdio.h>

int main() {
    int x=3,y=5;
    int *p;

    p = &x;
    *p = 10;
    printf("Valore di x: %d\n", x);

    p = &y;
    *p = 20;
    printf("Valore di y: %d\n", y);

    return 0;
}
```

```
Valore di x: 10
Valore di y: 20
```

Puntatori

```
#include <stdio.h>

int main() {
    int x=3,y=5;
    int *p;

    p = &x;
    *p = 10;
    printf("Valore di x: %d\n", x);

    p = &y;
    *p = 20;
    printf("Valore di y: %d\n", y);

    return 0;
}
```

```
Valore di x: 10
Valore di y: 20
```

Passaggio per riferimento

- Nel passaggio per riferimento, al parametro formale viene assegnato l'indirizzo del parametro effettivo.
- In questo modo, al sottoprogramma è possibile accedere al registro che ospita il parametro effettivo e fare delle modifiche che saranno poi visibili al programma chiamante.
- In altre parole, qualunque modifica effettuata sul parametro formale avrà effetto sul parametro effettivo corrispondente.

Passaggio per riferimento

Il passaggio per riferimento (o by reference) lo si realizza tramite puntatori

… già visto da qualche parte ?

```
#include <stdio.h>

void swap(int* a, int* b) {
   int temp;

   temp = *a;
   *a = *b;
   *b = temp;
}

int main()
{
          int x, y;

          printf("Valore di x: "); scanf("%d", &x);
          printf("Valore di y: "); scanf("%d", &y);

          swap(&x,&y);

          printf("Nuovo valore di x: %d\n", x);
          printf("Nuovo valore di y: %d\n", y);
          return 0;
}
```

Passaggio per riferimento

Il passaggio per riferimento (o by reference) lo si realizza tramite puntatori

```
#include <stdio.h>

void incrementa(int* a, int* b, int* c){
    *a = *a+1;
    *b = *b+1;
    *c = *c+1;
}

int main() {
    int x,y,z;

    x=0; y=1; z=2;
    incrementa(&x, &y, &z);
    printf("%d %d %d\n", x, y, z);
    return(0);
}
```

Passaggio per riferimento

- Il passaggio per riferimento fornisce un modo efficace per realizzare una funzione che deve restituire più di un valore

```
void precsucc(int x, int* prec, int* succ){
    *prec=x-1;
    *succ=x+1;
}
```

Array come parametri

- Gli array sono passati unicamente per riferimento (**perché ?**).
- Come parametro formale, un array si indica con tipo, identificatore e parentesi quadre: **`int vet[]`**.
- Come parametro effettivo va fornito il nome dell'array, senza specificare altro.

Array come parametri

```
void stamparray (int vet[], int num) {
  int i;
  for (i=0; i<num; i++)
    printf("%d  ", vet[i]);
  printf("\n");
}

int main ()
{
  int vet1[] = {5, 10, 15};
  int vet2[] = {2, 4, 6, 8, 10};

  stamparray (vet1, 3);
  stamparray (vet2, 5);

  return (EXIT_SUCCESS);
}
```

Array come parametri: errori frequenti

```
void stamparray (int vet[], int num) {
  int i;
  for (i=0; i<num; i++)
    printf("%d  ", vet[i]);
  printf("\n");
}

int main ()
{
  int vet1[] = {5, 10, 15};
  int vet2[] = {2, 4, 6, 8, 10};

  stamparray(vet1[3],3);
  stamparray(vet2,5);

  return (EXIT_SUCCESS);
}
```

ACHTUNG !!

Si passa un intero e non un array

korrekt !!

Array bidimensionali come parametri

- Nella definizione come parametro formale, va necessariamente definita la **cardinalità** del secondo indice dell'array.
- Possibile aggiungere la cardinalità del primo indice, ma il compilatore non ne tiene conto.

- Come parametro effettivo va fornito il nome dell'array, senza specificare altro.

Array bidimensionali come parametri

```
void stampamat (int mat[][5], int numrig, int numcol) {
  int i,j;
  for (i=0; i<numrig; i++){
     for (j=0; j<numcol; j++)
      printf("%2d  ", mat[i][j]);
    printf("\n");
  }
}

int main ()
{
  int mat1[5][5] = {{5, 10, 15},{7, 11, 44}};
  int mat2[5][5] = {{2, 4, 0}, {6, 8, 4}, {10, 12, 8}};
  int rig1 = 2, col1 = 3;

  stampamat(mat1,rig1,col1);
  printf("\n");
  stampamat(mat2,3,3);

  return (EXIT_SUCCESS);
}
```

Array multidimensionali come parametri

- Nella definizione come parametro formale, vanno necessariamente definite le **cardinalità** di tutti gli indici dell'array, tranne il primo.
- Possibile aggiungere la cardinalità del primo indice, ma il compilatore non ne tiene conto.

- Come parametro effettivo va fornito il nome dell'array, senza specificare altro.

Array multidimensionali come parametri

```
void stampamat3(int mat[][5][5], int dim1, int dim2, int dim3) {
  int i,j,k;
  for(i=0; i<dim1; i++){
     for(j=0; j<dim2; j++){
        for(k=0; k<dim3; k++)
          printf("%2d  ", mat[i][j][k]);
        printf("\n");
     }
     printf("\n\n");
  }
}

int main ()
{
  int mat[5][5][5] = {{{5, 10, 15},{7, 11, 44}},
                      {{4,  9, 14},{6, 10, 43}}};
  int d1 = 2, d2 = 2, d3 =3;

  stampamat3(mat,d1,d2,d3);
  printf("\n");

  return (EXIT_SUCCESS);
}
```

Il concetto di visibilità

- Un programma C può assumere una struttura complessa grazie all'uso dei sottoprogrammi.
- Questo rende necessario definire delle regole per l'uso degli oggetti (variabili, costanti, funzioni) definite nel programma.
- In particolare, tali regole precisano in quali parti del programma è possibile usare un certo identificatore (**visibilità**).

Visibilità delle variabili. Ambiente

- L'insieme delle variabili definite in una funzione può dividersi in due insiemi:
  - Parametri formali: utilizzati per gestire il flusso di dati con il chiamante
  - Altre variabili utilizzate per implementare l'algoritmo nel sottoprogramma (es. indici, variabili di appoggio, ecc.
- L'insieme di queste variabili viene definito *ambiente* della funzione.
- Analogamente, l'insieme delle variabili definite nel chiamante costituisce l'ambiente del chiamante.

Ambienti e visibilità

- Quale relazione esiste tra ambiente del chiamante e ambiente della funzione?
- In altre parole, il sottoprogramma ha la **visibilità** (cioè, può fare uso) delle variabili del chiamante e viceversa ?

Ambienti e visibilità

- Sono **due ambienti distinti** per cui:
  - Le variabili del chiamante non sono visibili dal sottoprogramma e viceversa.
  - Nei due ambienti possono quindi esistere variabili con lo stesso nome, ma sono due variabili distinte e separate.
  - L'unico canale per scambiarsi dati è quindi fornito dallo scambio di parametri.

Visibilità delle variabili

- Le variabili sono **locali** alle funzioni: sono cioè utilizzabili solo all'interno della funzione in cui sono definite.
- Più precisamente, la visibilità si riferisce al **blocco** in cui sono definite.

```c
#include <stdio.h>

int doppia(int x);

int main() {
        int a,b,c;

        printf("a: ");
        scanf("%d", &a);
        b=doppia(a);
        c=doppia(b);
        printf("a: %d\n", a);
        printf("b: %d\n", b);
        printf("c: %d\n", c);

        return(0);
}

int doppia(int x) {
        int a;

        a=x;
        {
          int c;

          c=a;
          a=2*c;
        }
        return(a);
}
```

Visibili `a`, `b` e `c` della funzione `main`

Visibili `a` e `x` della funzione `doppia`

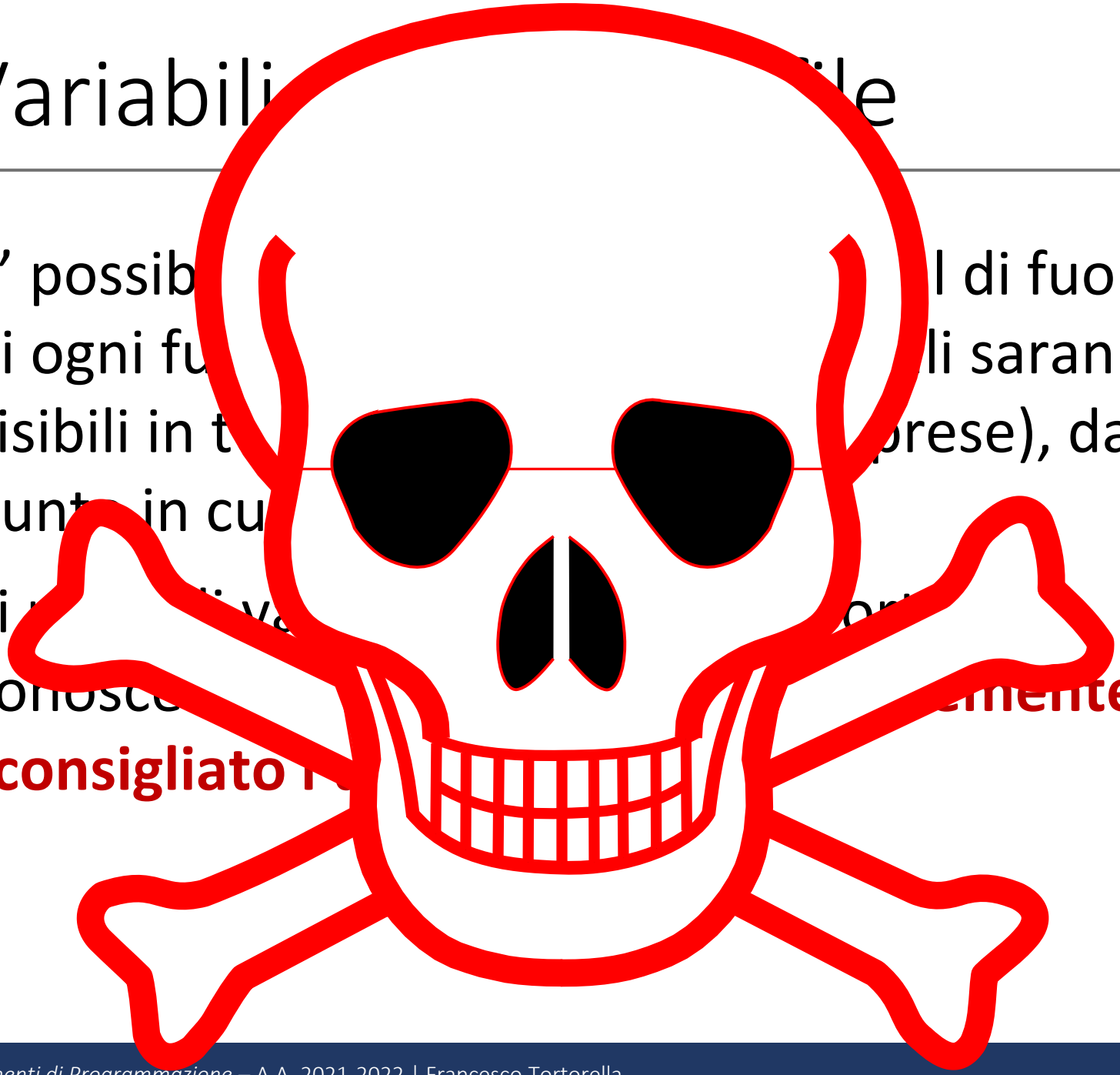
Visibili `a`, `x` e `c` della funzione `doppia`

Visibilità delle variabili

- All'interno di ogni blocco sono visibili:
  - le variabili definite al suo interno
  - le variabili definite nel blocco che eventualmente lo contiene
- La definizione di una variabile in un blocco annulla la visibilità di eventuali variabili con lo stesso nome, ma definite in blocchi esterni.

Variabili

- E’ possib l di fuori di ogni fu li saranno visibili in t prese), dal punt in cu
- Si conosce **sconsigliato**



Perché non si usano le variabili globali

- **Tramite le variabili globali, funzioni differenti possono realizzare uno scambio di dati che non è visibile e chiaramente definito come invece accade tramite il passaggio di parametri**
- Conseguenze:
  - Eventuali errori legati all'uso delle variabili globali (errata inizializzazione o aggiornamento da parte di qualche funzione) difficilmente individuabili (se tutte le funzioni possono accedere ad una variabile globale, come faccio a capire dov'è l'errore?)
  - Le funzioni non sarebbero facilmente riutilizzabili in altri programmi. Richiederebbero, infatti, che nei nuovi programmi fossero necessariamente definite variabili globali con lo stesso nome e dello stesso tipo (vincolo esagerato ed inutile)

Visibilità delle variabili

```
#include <stdio.h>

int doppia(int x);
int tripla(int x);
int z;

int main() {
        int a,b,c;

        printf("a: "); scanf("%d",&a);
        b=doppia(a);
        c=tripla(a);
        printf("a: %d\n", a);
        printf("b: %d\n", b);
        printf("c: %d\n", c);

        return(0);
}

int doppia(int x) {
        int a;
        a=2*x;
        return(a);
}

int tripla(int x) {
        int z;
        z=3*x;
        return(z);
}
```

Visibilità di `z`

Visibilità di `z` della funzione `tripla`

Progettare con i sottoprogrammi

- L'uso dei sottoprogrammi permette di organizzare in modo particolarmente efficace la progettazione di un programma. Infatti, con l'uso dei sottoprogrammi è possibile:
  - articolare il programma complessivo in più sottoprogrammi, ognuno dei quali realizza un compito preciso e limitato, rendendo più semplice la comprensione e la manutenzione del programma
  - progettare, codificare e verificare ad uno ad uno i singoli sottoprogrammi
  - riutilizzare in un programma diverso un sottoprogramma già codificato e verificato
  - limitare al minimo gli errori dovuti ad interazioni non previste tra parti diverse del programma (effetti collaterali)