# LE ASTRAZIONI FUNZIONALI

**AA 2006-2007**

**BR 16 nov 06), BA 17 nov 06: sino a slide 17**

**BR 21 nov 06), BA 22 nov 06: Br 18-37 + 51-55 – Ba sino a 36**

**BR 23 nov 06, BA 24 nov 06**

**ANNO ACCADEMICO 2005-2006**

tecniche di legame, procedure in C: void, Slide 48-61 (1h)
visibilità, classificazione delle risorse, shadowing, effetti collaterali, C: blocchi

# ASTRAZIONI FUNZIONALI

- I linguaggi ad alto livello consentono di creare unità di programma e, a tal scopo, offrono:
  - Costrutti per la definizione
    - Nome unico che individua un gruppo di istruzioni
  - Specifica della modalità di comunicazione con l'ambiente in cui l'unità viene inserita
    - Parametri e varie modalità di sostituzione
  - Meccanismi per l'utilizzo
    - Chiamata dei sottoprogrammi

# ASTRAZIONI FUNZIONALI

Unità distinte dal programma principale

- Sottoprogrammi: programmi non autonomi, ma "asserviti" ad altri programmi
- Diverse denominazioni
  - FORTRAN: *Subroutine*
  - Pascal: *Procedure* e *Function*
  - Linguaggio C: formalmente solo funzioni (anche il ***main*** è una funzione), le procedure possono essere realizzate come funzioni che non restituiscono alcun valore (***void***)

# ASTRAZIONI FUNZIONALI

Tecnica di programmazione che permette di ampliare il repertorio di operatori ed istruzioni nei linguaggi di programmazione.

$$f : D_i \longrightarrow D_o$$

e aumentare la potenza del linguaggio.

# Esempio 1

```
main () {
    int x, n, n_multipli;
    scanf (“%d %d”, &x, &n);
    n_multipli =  calcola_multipli(x,n);
    printf (“%d”, n_multipli);
}
```

3 funzioni nel main: scanf, printf e calcola_multipli

Suggerimenti

Usare commenti, prompting, controllare che x non sia 0,…

# Esempi 2 e 3

```
main () {
    int n, minimo;
    scanf ("%d", &n);
    minimo=
    ricerca_minimo (n);
    printf ("%d", minimo);
}
```

```
main () {
    int v[10];
    leggi (v, 10);
    ordina (v, 10);
    stampa (v, 10);
}
```

(NB: *v* è una variabile array globale)

# VANTAGGI: Stile e qualità

- Leggibilità = maggiore capacità di astrazione
- Manutenibilità = un programma più sintetico è più facilmente correggibile e adattabile
- Riusabilità = invocazione ripetuta in uno stesso programma o in più programmi
- Modularità = codice indipendente sviluppato separatamente
- Consente di rispettare la decomposizione concettuale ottenuta con il metodo di progettazione (struttura logica)
- Rende evidente l'architettura del programma (struttura fisica)

# DEFINIZIONE DI UN SOTTOPROGRAMMA

- Specifica della intestazione (header) del sottoprogramma
  - tipo di sottoprogramma + nome del sottoprogramma + parametri

- Specifica del corpo del sottoprogramma
  - insieme delle istruzioni che verranno eseguite ad ogni chiamata

- Specifica delle modalità di comunicazione tra il sottoprogramma e l'unità che lo vuole usare
  - definizione dei parametri formali

# USO

- Per invocare il sottoprogramma occorre conoscere la sua intestazione
- Chiamata o invocazione
  (nel punto in cui si vuole eseguire il corpo del sottoprogramma)
  - FORTRAN: *CALL identificatore*
  - Pascal e C: *identificatore (parametri effettivi)*

# STRUTTURA DEL MAIN

- Al variare del linguaggio di programmazione cambia anche il posto dove inserire la definizione dei sottoprogrammi
- Il C offre solo funzioni: tipicamente un programma C è composto da numerose piccole funzioni
- In C anche *main* è una funzione, è proprio da essa che inizia la esecuzione del programma
- In C tutte le funzioni possono essere dichiarate solo al livello del *main* (non è ammesso decomporre le funzioni)
- Nel file sorgente C la definizione delle funzioni necessarie deve precedere il *main*

# STRUTTURA DI UN MODULO FISICO C (FILE)

*#include …vari file…*

*dichiarazioni globali*

*definizioni di funzione*

*void main (parametri)*

*{*

*dichiarazioni locali al main e corpo del main*

*}*

# UNA REGOLA GENERALE

- La dichiarazione di una qualsiasi risorsa (costante, tipo, variabile, funzione) deve sempre precedere il suo utilizzo.
- Il compilatore segnala errore quando nel sorgente un identificatore compare prima della sua dichiarazione.

# LE RISORSE DEI SOTTOPROGRAMMI

- I sottoprogrammi possono avere risorse proprie
- Costanti,Tipi, Variabili e Sottoprogrammi definiti all'interno di un sottoprogramma sono RISORSE **LOCALI**
  - Create all'inizio dell'esecuzione del sottoprogramma
  - "Distrutte" al termine della esecuzione del sottoprogramma
  - In C non è consentito definire funzioni annidate

- Le risorse definite nel **main** sono RISORSE **GLOBALI**

# MECCANISMO DI CHIAMATA CLIENT/SERVER

- Nella invocazione di un sottoprogramma
  - La unità di programma **chiamante** ha il ruolo di **client**
  - La unità **chiamata** ha il ruolo di **server**
- Modello client/server
  - All'atto della chiamata, la esecuzione del chiamante viene sospesa
  - Il controllo passa al sottoprogramma chiamato
  - Al termine, il controllo ritorna al chiamante

# MODELLO DI ESECUZIONE

Come viene eseguito un programma C

- Gestione della memoria
  - Stack o Pila
  - Heap (per le strutture dinamiche – *malloc* e *free*)
- Nello Stack la memoria viene gestita dinamicamente (cioè in esecuzione)
- Nello Stack la memoria è organizzata in segmenti detti **record di attivazione** (r.a.)
  - Allocati ad ogni chiamata di funzione
  - Deallocati al termine (*return* al chiamante)

# ATTIVAZIONI

Le attivazioni sono gestite in C (e non solo) seguendo una disciplina LIFO Last In First Out

- Una funzione *F* viene attivata quando inizia la esecuzione del codice del suo corpo
- L'attivazione termina quando termina la esecuzione del suo codice (con *return* esplicito o meno)
- Una attivazione di *F* può venire sospesa (rimane comunque aperta) se la sua esecuzione è interrotta a causa della esecuzione di un'altra funzione *G* (la cui chiamata compare nel corpo di *F*)
- Una attivazione di *F* riprende al ritorno dalla esecuzione di *G*

# ATTIVAZIONI (continua)

Chiamate ricorsive: quando nel corpo di *F* compare la chiamata di *F* stessa.(v. in seguito)

Una stessa funzione può avere più attivazioni aperte contemporaneamente

- In C ed in tutti i linguaggi che adottano un modello di esecuzione che permette la ricorsione
- Esecuzioni diverse di una stessa funzione *F* usano variabili (locali) diverse, poiché a diverse chiamate di *F*, corrispondono r.a. diversi

Oss: utilizzare una stessa area di memoria per il passaggio parametri ad un sottoprogramma impedisce le chiamate ricorsive (sovrascrittura)

# RECORD DI ATTIVAZIONE

I r.a. sono allocati e deallocati dinamicamente nello Stack.

Un r.a. contiene:

- Nome del sottoprogramma e riferimento al codice
- i binding relativi ai parametri di ingresso e di uscita della funzione
- i binding relativi alle variabili locali della funzione
- l'indirizzo di ritorno (per individuare il punto della unità chiamante a cui dovrà essere restituito il controllo dopo la esecuzione della funzione)
- Altre informazioni (riferimento di catena dinamica e statica)

# ALLOCAZIONE

- Alla chiamata di una funzione *G* presente nel codice di una funzione *F* (al momento in esecuzione) corrisponde in memoria il caricamento del suo r.a.
  - La allocazione delle variabili avviene al tempo della esecuzione (C, Pascal, …), ossia quando serve
- In ogni istante, al top dello Stack si trova il r.a. della funzione al momento in esecuzione

# AREA DATI GLOBALI

- Le variabili globali sono allocate in un segmento di memoria appositamente riservato: l'area dati globali che è statica
- Anche il *main* è una funzione (in C)
- Nella parte iniziale (bottom) dello Stack sono allocati i binding delle variabili del *main*
- Il "r.a." del *main* rimarrà in memoria per tutta la durata del programma e l'indirizzo di ritorno è diretto al SO.
- Caricamento iniziale

# VISIBILITA'

- Le **regole di visibilità** delle risorse del linguaggio di programmazione si giustificano in base al modello di esecuzione
- Ad ogni chiamata di funzione viene caricato nello Stack il r.a. relativo ad essa
  - Le variabili locali "vivono" solo durante la attivazione
- Qualsiasi identificatore ***i*** riferito all'interno del codice di ***F*** deve avere una sua posizione nello Stack.
- Ricerca nei r.a. accatastati in memoria: (in C) se non viene trovato nel r.a. in cima allo Stack, si cercherà nell'area dati relativa al r.a. del *main*

# DEALLOCAZIONE

- Al termine della esecuzione della funzione G, il r.a. relativo alla chiamata di G viene rimosso dallo Stack
  - La memoria viene rilasciata
- Adesso, in cima allo Stack si troverà il r.a. relativo alla funzione F di cui riprendere la esecuzione
- Il r.a. appena rimosso (relativo a G) contiene come indirizzo di ritorno un indirizzo relativo alla funzione F (che ha chiamato G)
- L'ultimo r.a. ad essere rimosso è quello del *main*

# ESEMPIO

```
#include <stdio.h>
void r (int a) {
printf ("valore: %d\n", a)
}
void q (int a) {
r( a);
}
void p ( ) {
int a =10; q(a); return;
}
main ( ){
 p ( );
 }
```

Situazione in RAM

r.a. r

r.a. q

r.a. p

r.a. main

(Bottom) **STACK**

# DIMENSIONE DEI R.A.

Ad ogni sottoprogramma corrisponde un r.a. di dimensione specifica

I r.a. hanno dimensione nota al tempo della compilazione

- Di ciascuna variabile locale è noto il tipo (perciò lo spazio necessario)
- Di ciascun parametro è noto il tipo e la modalità di passaggio

# USO DELLE VARIABILI GLOBALI

Le variabili globali sono accessibili **sempre** e **da tutti** i sottoprogrammi.

E' difficile controllare la sequenza dei valori assunti da una variabile globale

CONSIGLIO

evitare un uso generalizzato delle variabili globali

(v.in seguito effetti collaterali)

# PARAMETRI

- I parametri costituiscono il mezzo di comunicazione tra chiamante e chiamato.
- Parametri formali
  - Entità fittizie specificate nella definizione del sottoprogramma come "segnaposto" per i parametri attuali
  - Nomi dummy che compaiono nella intestazione e nel corpo del sottoprogramma
  - Specifica del tipo del parametro formale
  - Numero e ordine di comparizione dei parametri
- Parametri attuali
  - Entità reali fornite dal chiamante (specificate nella chiamata) sulle quali il sottoprogramma opererà

# LEGAME DEI PARAMETRI

- L'esecuzione della chiamata comporta la associazione dei parametri attuali ai parametri formali.
- Corrispondenza fra parametri attuali e formali
  - Concordanza posizionale
  - Concordanza in numero
  - Identicità/Compatibilità di tipo
- Esistono varie tecniche di legame dei parametri (v. oltre)
  - per valore
  - per indirizzo

# MATCHING

- Corrispondenza fra parametri attuali e formali
  - Concordanza posizionale
  - Concordanza in numero
  - Identicità/Compatibilità di tipo

Intestazione: *long power (int b, int e)* $b^e$

Chiamata: *z = power (10, n)* $10^n$

Att.ne: Possono essere generati errori

- per ordine errato: Se *power (n, 10)* si calcolerà $n^{10}$
- per le conversioni (da tipo + basso ad uno + alto oppure da tipo +alto ad uno +basso).

# TIPI DI SOTTOPROGRAMMI

- Procedure
  - Astrazione della nozione di istruzione.
  - La chiamata può comparire dovunque può esserci una istruzione
  - Producono un EFFETTO
    - Può produrre molteplici valori (ma anche 1 soltanto)
    - Può impattare l'ambiente per la emissione di risultati o per la acquisizione di dati
- Funzioni
  - Astrazione del concetto di operatore
  - Si possono attivare durante la valutazione di una espressione
  - Restituiscono UN VALORE (risultato di tipo semplice)

# Struttura dei SOTTOPROGRAMMI

- Intestazione
  - Costituisce l'aspetto PUBBLICO del sottoprogramma
  - Occorre conoscerla per poter usare il sottoprogramma
- Blocco
  - Costituisce l'aspetto PRIVATO del sottoprogramma
  - Parte dichiarativa
    - Risorse locali
  - Corpo (o parte esecutiva)
    - Ne definisce il comportamento

ESEMPIO: usiamo le funzioni di libreria senza conoscerne la implementazione

# LE FUNZIONI

- Nella intestazione si definisce il dominio ed il codominio.
- Dominio $D_i$: tipo_arg1 x tipo_arg2 x tipo_argn
  - Argomenti : parametri di input
  - Lista dei parametri formali separati da virgole e racchiusa tra parentesi tonde (sintassi non solo del C)
- Codominio $D_o$: tipo_risultato
  - Deve essere di tipo semplice (built-in o definito dall'utente) o (in C) puntatore ad un tipo qualsiasi
- Parte dichiarativa
- Corpo
- Presenza della istruzione return (in C)

# LE FUNZIONI IN C

- **< def. di funzione>** ::= <intestazione> <blocco>
- **<intestazione>** ::=

  <identificatore_di_tipo> <identificatore_di_funzione> (<lista_parametri_formali>)

dove

<identificatore_di_tipo> indica il tipo del risultato della funzione (codominio)

<identificatore_di_funzione> è il nome della funzione

<lista_parametri_formali> è la lista dei parametri formali (dominio) separati da virgola, per ciascun parametro occorre specificare il tipo ed il nome simbolico

- **<blocco>** ::= {<parte dichiarativa> <parte istruzioni>}

# LA ISTRUZIONE *RETURN*

- Per restituire il risultato ogni funzione C utilizza la istruzione ***return***

<istruzione return> ::= ***return*** [<espressione>]

Ha l'effetto di restituire il controllo al chiamante ed, eventualmente, assegnare all'identificatore della funzione il valore della espressione (che è opzionale)  f ⟵ calcolo eseguito da f

- Il *return* potrebbe mancare: il sottoprogramma restituisce il controllo in corrispondenza della parentesi graffa chiusa }

# ESEMPI 1 e 2

```
int leggi_un_intero ( )    /*
    intestazione */
{
    int num;
    scanf("%d", &num);
    return num;
}
```

Non ha parametri di I

Produce il valore di num

```
long potenza (int base, int
    esponente)
{
    int i;
    long p=1;
    for (i=1; i<= esponente; i++)
          p = p * base;
    return p;
}
```

Ha 2 parametri di I, produce il valore di p

# ESEMPIO 3

```
int max (int a, int b)
{
    if (a > b) return a;
        else return b;
}
```

***Att.ne non è strutturata!***

```
int risult
if (a > b) risult = a;
        else risult = b;
return risult;
```

```
#include <stdio.h>
int max (int a, int b) {
int risult;
if (a > b) risult = a;
        else risult = b;
return risult;
}
main ( ){
    int x, y;
    printf ("inserisci 2 numeri");
    scanf ("%d %d", &x, &y);
    printf ("%d\n", max(x,y) );
        }
```

# ESEMPIO 4

```
int stampa_un_intero (int n)
  /* intestazione */
{
   printf (“%d”, n);
}
```

Se non c’è alcuna istruzione ***return,*** il valore restituito sarà ***indefinito***

Oss: E’concettualmente una *procedura* (usare ***void)***

# UNA FUNZIONE C

```
int minimo (int n)   /* funzione minimo */
{
    int i, min, num;  /* var. locali */
        min = 0;                    /* Att.ne: se valori tutti positivi, min varrà 0 */
                                    /* pur non facendo parte dei dati */
        i = 0;
        while (i < n)
      {
        i++;
        scanf ("%d", &num);
        if (num < min)
                min = num;

      }
    return min;
}
```

Una chiamata potrà essere: `risultato = minimo(10);`

# REALIZZAZIONE DELLE PROCEDURE IN C

Il concetto di procedura si implementa come una funzione che non restituisce alcun valore come risultato

- **void**
  - insieme vuoto di valori
- **void** identificatore_di_funzione (…)
  - funzione che non restituisce alcun valore (codominio vuoto)

# ESEMPIO DI PROCEDURA

```
void  stampa_un_intero
    (int n)
{
      printf ("%d", n);
}
```

```
main ( )
{
    int num;
    scanf ("%d", &num);
    stampa_un_intero(num);
}
```

Nota 1: la chiamata è come una istruzione

Nota 2: Nel file *stdio* sono contenute le definizioni di *scanf*, *printf*,…

# LE TECNICHE DI LEGAME

Esistono diverse tecniche di legame dei parametri

- Legame per valore
- Legame per riferimento
- Legame per nome

Ciascun linguaggio di programmazione ammette uno o più meccanismi di legame.

In C è disponibile **solo** il legame **per valore.**

Pascal: valore e riferimento

Algol 60: valore e nome

Simula 67: tutti e tre

# LE TECNICHE DI LEGAME

- Sia **P** una procedura avente un parametro formale $\mathbf{p_f}$
- Supponiamo che **K** sia una unità di programma che chiami P passando una variabile $\mathbf{p_a}$ (visibile nella unità **K)**:

  $\mathbf{P(p_a)}$

```
void P (int p_f)
{…
}
```

```
Unità K
int p_a;
…
P(p_a);
…
```

# LEGAME PER VALORE

- Prima della chiamata
  - (Ip: $\mathbf{p_a}$ è una variabile) Nell'area dati di K, il binding di $\mathbf{p_a}$ punta ad una locazione di memoria contenente per esempio $\alpha$ (in generale, $\mathbf{p_a}$ può essere una espressione)
- Al momento della chiamata
  - Viene valutata $\mathbf{p_a}$
  - Nel r.a. della chiamata a P, una cella è associata a $\mathbf{p_f}$
  - Viene copiato il valore di $\mathbf{p_a}$ in $\mathbf{p_f}$ (avviene un assegnamento $p_f = p_a$ per passare il valore)
- In esecuzione
  - il parametro formale $\mathbf{p_f}$ si comporta come una variabile locale (può anche subire modifiche di valore)
- Al termine della esecuzione
  - la memoria allocata per $\mathbf{p_f}$ viene rilasciata; $\mathbf{p_a}$ rimane inalterato

# LEGAME PER RIFERIMENTO

- Prima della chiamata
  - ($\mathbf{p_a}$ deve essere una variabile) Nell'area dati di K, il binding di $\mathbf{p_a}$ punta ad una locazione di memoria contenente per esempio $\alpha$
- Al momento della chiamata
  - Viene associata all'identificatore $\mathbf{p_f}$ la stessa cella di memoria riferita da $\mathbf{p_a}$ ($\mathbf{p_f}$ è un alias di $\mathbf{p_a}$) – (si passa il riferimento)
- In esecuzione, le modifiche subite dal parametro formale $\mathbf{p_f}$ sono sostanzialmente effettuate su $\mathbf{p_a}$
- Al termine della esecuzione, le modifiche subite da $\mathbf{p_f}$ permangono su $\mathbf{p_a}$.

# Come passare per riferimento in C?

Tramite parametri di tipo puntatore si può simulare il passaggio per riferimento

Per calcolare l'indirizzo si usa l'operatore &.

Per calcolare il valore contenuto nella cella di indirizzo x, è disponibile l'operatore *

```
void scambio  (int *x, int *y){
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
        }
```

CHIAMATA: *scambio (&a, &b);*

# TECNICHE DI LEGAME: ESEMPIO

```
#include <stdio.h>
void f (int ?x);
int n;
main () {
    n=3;
    f(n);
    printf ("%d", n); /* TRE */
 }
void f (int ?x) {
x = x+1;
printf ("%d", n); /* UNO */
printf ("%d", x); /* DUE */ }
```

- Se il legame è per valore
  - UNO stamperà 3
  - DUE stamperà 4
  - TRE stamperà 3
- Se il legame è per riferimento
  - UNO stamperà 4
  - DUE stamperà 4
  - TRE stamperà 4

La semantica delle due tecniche è differente!

# TECNICHE DI LEGAME: ESERCIZIO

```
#include <stdio.h>
void cambia (int ?x);
int n;
main ( ) {
    n=0;
    cambia(n);
    printf ("%d", n); /* TRE */
 }
void cambia (int ?x) {
x = 1;
printf ("%d", n); /* UNO */
printf ("%d", x); /* DUE */ }
```

- Se il legame è per valore
  - UNO stamperà 0
  - DUE stamperà 1
  - TRE stamperà 0
- Se il legame è per riferimento
  - UNO stamperà 1
  - DUE stamperà 1
  - TRE stamperà 1

# TECNICHE A CONFRONTO

- Per valore
  - Permette la trasmissione "sicura" di parametri-dati
  - Permette la separazione tra chiamante e chiamato

  *ma*
  - Aumenta l'occupazione temporanea di memoria
  - Richiede tempo per la copia dei valori
  - Rende problematica la gestione dei parametri di dimensione variabile
- Per riferimento
  - Permette la trasmissione di parametri-risultato (in generale di I/O)
  - Finestra in comune tra chiamante e chiamato
  - La gestione degli indirizzi è a carico del compilatore
  - Non richiede né memoria, né tempo aggiuntivo

  *ma*
  - Può causare effetti collaterali spesso imprevedibili

# LA SCELTA DEI PARAMETRI

- Quanti e quali
  - Dipende dalla funzionalità svolta dal sottoprogramma
  - Dipende dal grado di modularità che vogliamo raggiungere
- Quale modalità di passaggio
  - Dipende dal ruolo di ciascun parametro
  - Valutazione pro-contro  (v. array come parametri)
- Quale ordine di comparizione nella intestazione
  - Non c'è una regola
  - Suggerimento: prima dati di ingresso e poi di uscita

# ESERCIZIO

- Stampa di un quadrato 10 x 10 di asterischi
- Scrivere il frammento di codice C
- Funzione o procedura?
- Variabili locali?
- Parametri? Modalità di legame?
  - 0
  - 1
  - 2
  - 3

# ARRAY COME PARAMETRI

- In C gli array sono sempre passati attraverso il loro indirizzo (puntatore all'array)
- Le eventuali modifiche all'array effettuate all'interno della funzione sopravvivono all'esecuzione della funzione

```
int leggi_vet (int v[ ] , int dim);
```

è equivalente a

```
int leggi_vet (int *v , int dim);
```

# DEFINIZIONE N.E.

# DICHIARAZIONE

- Definizione: è una specifica completa del sottoprogramma
  - descrive le proprietà della funzione (tipo, nome, elenco parametri)
  - descrive la sua implementazione.

# DEFINIZIONE N.E.
# DICHIARAZIONE

- Dichiarazione: è una specifica parziale che descrive solo le proprietà della funzione
  - Si può postporre la definizione di una funzione al suo utilizzo solo se essa viene dichiarata mediante la sua intestazione (header) – prototipo
    Analoga alla direttiva FORWARD del Pascal
- Una funzione può essere dichiarata in punti diversi, ma deve essere definita una sola volta.

# ESEMPIO: due possibilità

```
#include <stdio.h>
int max (int a, int b)
{int risult;
if (a > b) risult = a;
        else risult = b;
return risult;
}
main ( ){
    int x, y;
    printf ("inserisci due
    numeri");
    scanf ("%d %d", &x,
    &y);
    printf ("%d\n", max(x,y)
    ); }
```

```
#include <stdio.h>
int max (int a, int b);
                /*dichiarazione*/
main ( ){
    int x, y;
    printf ("inserisci due numeri");
    scanf ("%d %d", &x, &y);
    printf ("%d\n", max(x,y) );
 }
int max (int a, int b) /*definizione*/
{int risult;
if (a > b) risult = a;
        else risult = b;
return risult;}
```

# I PROTOTIPI IN C

- I prototipi delle funzioni utilizzate possono comparire:
  - Nella parte dichiarazioni globali di un programma
  - Nella parte dichiarazioni del main
  - Nella parte dichiarazioni delle funzioni

Per il compilatore sono equivalenti:

*double sin(double x);* ← è preferibile

*double sin (double);*

Entrambe informano il compilatore che sarà utilizzata una funzione di nome *sin* che restituisce un valore di tipo *double* e che richiede un parametro di tipo *double*.

# SUGGERIMENTO DI STILE

Far comparire la definizione del *main* prima delle definizioni delle altre funzioni (leggibilità)

1. Variabili e tipi globali
2. Dichiarazioni delle funzioni (Elenco dei prototipi)
3. Definizione del *main*
   - Parte dichiarativa (le locali al *main*)
   - Codice del main (in esso compaiono le invocazioni delle funzioni)
4. Definizioni delle funzioni dichiarate
   - Variabili locali rispettive
   - Codice (possono comparire le invocazioni di funzione)

# CLASSIFICAZIONE DELLE RISORSE

- LOCALE: risorsa definita all'interno di un sottoprogramma (i parametri formali vengono trattati come variabili locali)
  - temporanea
- GLOBALE: risorsa definita nel programma principale (tutti gli identificatori pre-definiti sono considerati risorse globali)
  - statica, permane in vita per tutta la esecuzione del programma
- In C, non essendo consentite le funzioni annidate, manca il livello intermedio

# REGOLE DI VISIBILITA'

- Una risorsa LOCALE ad un blocco o sottoprogramma P è visibile in esso e in tutti i blocchi in esso contenuti (durante la esecuzione del blocco o P)
- Una risorsa GLOBALE è visibile sempre dal punto in cui è dichiarata sino alla fine del file sorgente
- A meno dello shadowing

Tempo di vita = tempo di esecuzione del blocco

Ambito = campo di azione in cui si può fare riferimento alla risorsa

# BLOCCHI del C

- Raggruppare dichiarative ed istruzioni senza attribuirvi un nome
- Racchiusi tra gli elementi lessicali {e}
- Il blocco è trattato come se fosse una unità di programma
  - accatastamento del r.a. nello Stack
  - va in esecuzione quando viene incontrato nel codice
  - deallocazione del r.a. al termine del blocco
- Nel blocco possono essere:
  - definite risorse locali (non funzioni)
  - dichiarate funzioni

# ACCESSIBILITA'

- Insieme delle risorse a cui un sottoprogramma può far riferimento (si può "usare" tutto ciò che si "può vedere")
- Dipende da:
  - Fattore statico
    - In base alla struttura del codice (dichiarazioni)
    - Regole di visibilità "statiche" (le proprie locali, le globali e le non-locali-non-globali)
  - Fattore dinamico
    - Sequenza delle attivazioni dei sottoprogrammi
    - Accatastamento dei r.a. nello Stack

  Il C non adotta regole di visibilità dinamiche

# ACCESSIBILITA'

- Un sottoprogramma può riferire
  - Le proprie variabili
  - Le variabili globali
  - (Non in C) Le variabili dichiarate dai sottoprogrammi al momento in esecuzione
    - in esecuzioni diverse potrà accedere a variabili diverse
    - dipende dall'ordine delle chiamate sino a quel momento
    - Esempio: in strutture selettive possono comparire invocazioni differenziate

***if** (k=n) invoca f1 **else** invoca f2*

# SHADOWING

- Shadowing o occlusione
  - ridefinizione di uno stesso identificatore in blocchi differenti
  - Sinominia (per entità senza alcuna attinenza)
- L'accesso è sempre al sinonimo "più vicino"
  - Ricerca a ritroso nello Stack

# ESEMPIO di SHADOWING

```
#include <stdio.h>
int x =0;
void P1( ); /* dichiarazione superflua */
void P2( );
main ( ) {
    x = x +1;
    P2; /* invocazione di P2 */
}
void P1( ) {printf ("\n%d", x); }
void P2( ) {float x; x = 2.14; P1; }
```

In C, viene stampato il valore 1

Nei linguaggi che adottano regole dinamiche, verrebbe stampato 2.14

# EFFETTI COLLATERALI

- Provocati dall'attivazione di una funzione
- Modifica di variabili esterne al proprio ambiente (= proprio r.a.)
- Possibili quando:
  - Si usano parametri di tipo puntatore
  - Si usano variabili globali
- Decade il concetto di funzione matematica!
- Sono da evitare

# LA COMUNICAZIONE

- Un sottoprogramma può comunicare
  - Con l'ambiente esterno
    - Tramite istruzioni di lettura e/o scrittura
  - Con l'ambiente chiamante
    - Implicitamente
      - Attraverso le variabili globali (ed eventualmente, se previsto dal linguaggio, tramite le non-locali-non-globali)
    - Esplicitamente
      - Attraverso i parametri