Procedure e funzioni

Compatibilità tra tipi

- Il concetto di tipo è un importante **strumento di astrazione**: separa la visione esterna di un'informazione dalla rappresentazione interna
- Come "mettere insieme" informazioni di tipo diverso?
- In prima istanza vale il principio del "non sommare le pere con le mele":
  - Ha senso sommare un nome ad un conto corrente?
  - Assegnare il valore 'a' ad un indirizzo?
- Ad un'analisi più attenta, però, nascono interrogativi un po' meno banali:
  - Se ridefinisco età come intero e temperatura come intero non dovrei assegnare un'età a una temperatura…
  - Ma un intero a un reale?
  - Un reale a un intero?
  - …
- E' raccomandabile dare regole precise -non tutti i linguaggi lo fanno- per evitare rischi di interpretazioni
- Vediamo allora le principali regole del C
  - Precise ma anche molto "liberali"

Espressioni con tipi eterogenei

- Un'espressione aritmetica come x + y è caratterizzata dal valore e dal tipo del risultato
- Il tipo degli operandi condiziona l'operazione che deve essere eseguita
  - A operandi di tipo int si applica l'operazione di somma propria di tale tipo, diversa è l'operazione di somma che si applica a operandi di tipo float…
- Se x è di tipo short int e y di tipo int è necessario convertire una delle due variabili per rendere omogenea l'espressione e applicare la corretta operazione
  - x viene temporaneamente convertita in int e la somma tra interi restituisce un risultato intero
- In generale sia x op y
- Regole di **conversione implicita**:
  - Ogni variabile di tipo char o short int (incluse le rispettive versioni signed o unsigned) viene convertita in variabile di tipo int
  - Se dopo l'espressione risulta ancora eterogenea rispetto al tipo degli operandi coinvolti, rispetto alla seguente gerarchia int < long < float < double < long double
    - si converte temporaneamente l'operando di tipo inferiore facendolo divenire di tipo superiore
    - il risultato dell'espressione avrà tipo uguale a quello di più alto livello gerarchico
  - **Nota**: in C è dunque possibile sommare un carattere - o un tipo enumerato- ad un intero: prevale la visione "concreta" che interpreta tutte le sequenze di bit come numeri

Assegnamenti con tipi eterogenei

- Le regole di conversione implicita esposte vengono utilizzate anche per la valutazione di **assegnamenti** tra variabili eterogenee in tipo:
  - double d;
  - int i;
- L'istruzione d = i;
  - provoca una temporanea conversione del valore dell'intero i a double e successivamente l'assegnamento di tale valore double a d
- L'esecuzione dell'istruzione i = d;
  - comporta invece, normalmente, una perdita di informazione
  - il valore di d subisce infatti un troncamento alla parte intera con perdita della parte decimale

Osservazione

- Le regole di uso dei tipi del C sono tutte verificabili dal compilatore, si dice "a **compile-time**"
- Questa caratteristica viene anche indicata come tipizzazione forte; non tutti i linguaggi ne sono dotati
- Generalizzando questa osservazione
  - Molti linguaggi preferiscono, laddove possibile, eseguire operazioni (verifiche di errori, allocazione di memoria) a compile time piuttosto che a run-time
- La verifica dell'assenza di un errore a compile time, ne garantisce effettivamente l'assenza:
  - Errori di sintassi, di tipo (in C) sono errori verificabili a compile-time
  - Non tutti gli errori però hanno questa (piacevole) caratteristica

Alcuni tipici errori a run-time

- La divisione per '0’
  - Infatti l'operazione di divisione, sia intera che reale, è definita (cioè applicabile) solo al caso in cui il divisore sia diverso da '0’
  - Ma ciò, in generale, non è prevedibile prima dell'esecuzione
- L'uso di un indice di un array con valore al di fuori del suo campo di variabilità:
  - int V1[100];
- permette al compilatore di segnalare che l'istruzione: V1[132] = 190 è un errore, ma il caso più generale:
  - scanf (&i); V1[i] = 190;
- non è verificabile durante la compilazione perché dipende dalla lettura del dato i

Sottoprogrammi

- I meccanismi di astrazione sui dati visti finora non corrispondono in pieno al concetto di tipo di dato astratto
- Mancano le operazioni:

```
typedef struct {
        int                     NumFatture;
        DescrizioneFatture      Sequenza[MaxNumFatture];
} ElencoFatture;

typedef struct {
        int    … come sopra
 } ElencoCosti;
```

MaxNumFatture e MaxNumCosti sono costanti

DescrizioneFatture e DescrizioneCosti sono tipi definiti (in precedenza)

Consideriamo le operazioni

- Consideriamo operazioni come “Calcolo del fatturato complessivo”, “Fatturato relativo al periodo x”, “Calcolo della somma dei costi complessivi”, …
- Un codice del tipo

```
RisultatoDiGestione = FatturatoTotale (ArchivioFatture)  – SommaCosti (ArchivioCosti);
```

dove ArchivioFatture e ArchivioCosti sono di tipo ElencoFatture ed ElencoCosti,
è certo meglio (cioè più astratto) di:

```
FatturatoTotale = 0; Cont = 0;
while (Cont < ArchivioFatture.NumFatture) {
        FatturatoTotale = FatturatoTotale + ArchivioFatture.Sequenza[Cont].Importo;
        Cont = Cont + 1;
}
SommaCosti= 0; Cont = 0;
while (Cont < ArchivioCosti.NumCosti) {
        SommaCosti = SommaCosti + ArchivioCosti.Sequenza[Cont].Importo;
        Cont = Cont + 1;
}
RisultatoDiGestione = FatturatoTotale – SommaCosti;
```

Funzioni (definizione e uso)

- Le operazioni astratte considerate finora sono funzioni nel senso matematico del termine: hanno un dominio e un codominio
- La loro definizione (semplificata) consiste in:
  - Una testata (**header**);
  - Due parti, sintatticamente racchiuse fra parentesi graffe:
    - la **parte dichiarativa** (detta parte dichiarativa locale)
    - la **parte esecutiva** (detta corpo della funzione)
- La testata contiene le informazioni più rilevanti per l'uso corretto del sottoprogramma
  - **Tipo del risultato**,
  - **Identificatore** del sottoprogramma,
  - Lista dei parametri cui la funzione viene applicata con il relativo tipo (detti **parametri formali**);
- Ogni parametro formale è un identificatore di tipo seguito da un identificatore
- Una funzione non può restituire array o funzioni ma può restituire un puntatore a qualsiasi tipo

Qualche esempio di testata

- **int** FatturatoTotale (ElencoFatture par)
- boolean Precede (StringaCaratteri par1,
  StringaCaratteri par2)
- boolean Esiste (int par1, SequenzaInteri par2)
- MatriceReali10Per10 *MatriceInversa
  (MatriceReali10Per10 *par)

Funzioni complete

```
int FatturatoTotale (ElencoFatture parametro) {
        int Totale = 0, Cont = 0;

        while (Cont < parametro.NumFatture) {
                Totale = Totale + parametro.Sequenza[Cont].Importo;
                Cont = Cont +1;
        }
        return Totale;
}

int RadiceIntera (int par) {
        int Cont = 0;

        while (cont*cont <= par) cont = cont + 1;
        return (cont–1);
}
```

Chiamata delle funzioni

- Identificatore della funzione seguito dalla lista dei parametri attuali
- Ogni parametro può essere un'espressione qualsiasi, eventualmente contenente un'altra chiamata di funzione
  - Prendono il nome di **parametri attuali**
- La corrispondenza tra parametri segue l'ordine della dichiarazione
- Inoltre il tipo dei parametri attuali deve essere compatibile con il tipo dei corrispondenti parametri formali
- La stessa funzione può essere chiamata in diversi punti dello stesso programma con diversi parametri attuali

Qualche esempio di chiamata

```
x = sin(y) – cos(PiGreco – alfa);
x = cos(atan(y) – beta);
x = sin(alfa);
y = cos(alfa) – sin(beta);
z = sin(PiGreco) + sin(gamma);
RisultatoDiGestione = FatturatoTotale (ArchivioFatture)  - SommaCosti(ArchivioCosti);

Det1 = Determinante (Matrice1);
Det2 = Determinante (MatriceInversa (Matrice2));

TotaleAssoluto = Sommatoria (Lista1) + Sommatoria (Lista2);
ElencoOrdinato = Ordinamento (Elenco);
OrdinatiAlfabeticamente = Precede (nome1, nome2);
```

Funzioni e struttura di programma

- Parti già note di un programma C
  - Direttive (#define…)
  - Parte main()
    - A sua volta costituito da una parte dichiarativa e una esecutiva
- Aggiungiamo ora
  - Una parte **dichiarativa globale** compresa fra la parte direttive e il programma principale
    - Contiene la dichiarazione di tutti gli elementi che sono condivisi, ossia usati in comune, dal programma principale e dai sottoprogrammi
    - Tali elementi possono essere costanti, tipi, variabili e altro ancora che verrà introdotto più avanti
    - Una successione di definizioni di sottoprogrammi (funzioni o procedure), a seguire il programma principale.
  - Il programma principale e ciascun sottoprogramma possono usare tutti e soli gli elementi stati dichiarati nella loro propria parte dichiarativa, nonché quelli che sono stati dichiarati nella parte dichiarativa globale

Prototipo delle funzioni

- All'interno di un programma C una funzione può essere chiamata purché risulti definita oppure **dichiarata**
- La dichiarazione di una funzione (anche detta prototipo) si limita a richiamarne la testata
  - Utile quando la chiamata di una funzione precede, nel codice, la definizione della funzione, oppure le funzioni utilizzate da un programma sono definite in file propri del sistema C (e.g., le **funzioni di libreria**)
  - Aiuta il compilatore ed è buono stile di programmazione
- Riassumendo, la parte dichiarativa del programma principale e quella dichiarativa di una funzione contengono i seguenti elementi:
  - dichiarazioni di costanti
  - dichiarazioni di tipo
  - dichiarazioni di variabili
  - prototipi delle funzioni

Esempio

```
#define MaxNumFatture 1000

typedef char String [30];
typedef struct   {
        String                  Indirizzo;
        int                     ammontare;
        Data                    DataFattura;
} DescrizioneFatture;
typedef struct   {
        int                     NumFatture;
        DescrizioneFatture      Sequenza[MaxNumFatture];
} ElencoFatture;
int FatturatoTotale (ElencoFatture  parametro);
main() {
        ElencoFatture   ArchivioFatture1,  ArchivioFatture2;
        int Fatt1, Fatt2, Fatt;
        ....
        Fatt1 = FatturatoTotale(ArchivioFatture1);
        Fatt2 = FatturatoTotale(ArchivioFatture2);
        Fatt = Fatt1 + Fatt2;
        ....
}
int FatturatoTotale (ElencoFatture parametro) {
        int Totale, Cont;
        ....
        return   Totale;
}
```

Parte dichiarativa globale.

Prototipo di funzione, utilizzata (e specificata) in seguito…

Implementazione della funzione.

Le procedure

- Non tutte le operazioni astratte sono formalizzabili come funzioni nel senso matematico del termine
- Esempio: vogliamo ordinare un archivio di fatture, ad esempio in ordine crescente di data
- L'effetto di una chiamata

  ordina (archivio),

  con archivio del tipo ElencoFatture non dovrebbe essere il valore di tipo ElencoFatture ottenuto ordinando gli elementi in archivio
- Si vuole invece che proprio il contenuto della variabile archivio venga ordinato dall'operazione ordina (archivio)
- Per casi del genere il tipo di sottoprogramma adatto è la **procedura**

Procedure e funzioni

- Da un punto di vista sintattico le procedure si distinguono dalle funzioni per
  - Il "tipo speciale" **void** del risultato
  - La chiamata che non produce un valore
    - Quindi, non si può trovare all'interno di un'espressione

Esempio…

```
typedef struct {
   String              indirizzo;
   int                 ammontare;
   Data                DataFattura;
} DescrizioneFatture;
typedef struct {
   int                      NumFatture;
   DescrizioneFatture       Sequenza[MaxNumFatture];
} ElencoFatture;
ElencoFatture               ArchivioFatture;

main() {
   Data DataOdierna;
   DescrizioneFatture Fattura1, Fattura2;
   void InserisciFattura (DescrizioneFatture Fattura);
   boolean Precede (Data Num1, Data Num2);
   // Sequenza di istruzioni che leggono i dati di una fattura in Fattura1…
   InserisciFattura(Fattura1);
   // Sequenza di istruzioni che leggono i dati di una fattura in Fattura2…
   if (Precede(Fattura2.DataEmissione, DataOdierna))
      InserisciFattura (Fattura2);
   ...
}
```

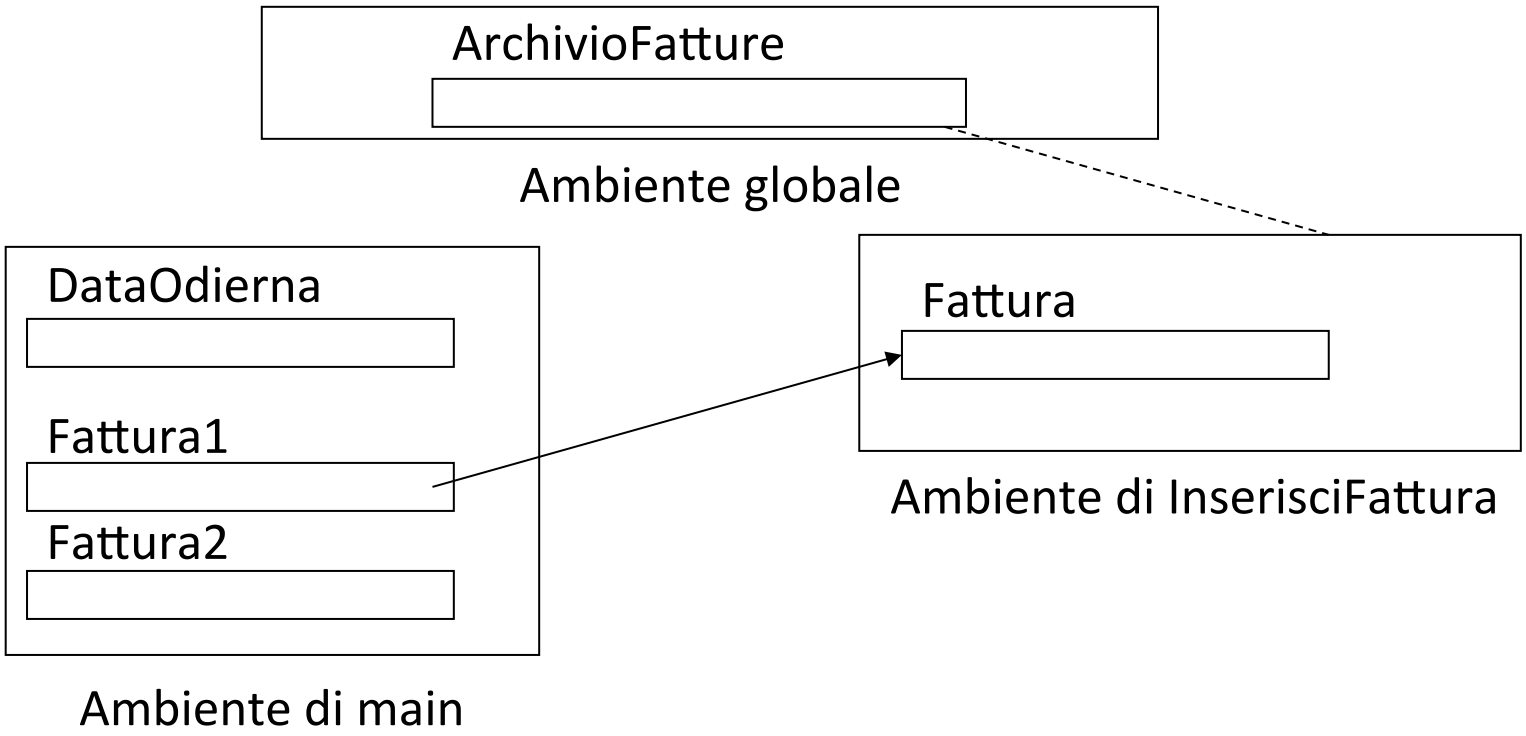
Commento in linguaggio C!

Chiamata a procedura.

…completo


```
void InserisciFattura (DescrizioneFatture  Fattura) {
  if(ArchivioFatture.NumFatture == MaxNumFatture)
    printf("L'archivio è pieno.\n");
  else {
    ArchivioFatture.NumFatture = ArchivioFatture.NumFatture + 1;
    ArchivioFatture.Sequenza[ArchivioFatture.NumFatture–1] = Fattura;
  }
}
```

Ambienti



L'esecuzione di una procedura non produce un risultato ma cambia lo stato del programma agendo sull'ambiente globale!

Le varie procedure e funzioni (main compreso) non possono accedere agli ambienti locali altrui: la "comunicazione" avviene o passandosi parametri e risultati o attraverso l'ambiente globale/comune!

Passaggio parametri

Supponiamo ora di voler eseguire diversi inserimenti di nuove fatture in **diversi archivi**

```
void InserisciFattura (DescrizioneFatture Fattura, ElencoFatture  ArchivioFatture) {
        if (ArchivioFatture.NumFatture == MaxNumFatture)
                printf("L'archivio è pieno.");
        else {
                ArchivioFatture.NumFatture = ArchivioFatture.NumFatture + 1;
                ArchivioFatture.Sequenza[ArchivioFatture.NumFatture–1] = Fattura;
        }
}
```

Esaminiamo l'effetto della chiamata:

```
InserisciFattura (Fattura1, ArchivioFatture5);
```

L'esecuzione della procedura produce l'effetto di inserire il valore di Fattura in ArchivioFatture, ma non in ArchivioFatture5, che è invece rimasto come prima!

Infatti l'operazione astratta della procedura viene eseguita sul **parametro formale**, non sul corrispondente **parametro attuale**

Passaggio per indirizzo

- Utilizzando il costruttore di tipo **puntatore** per la definizione dei parametri formali della funzione
- Usando l'operatore di **dereferenziazione** di puntatore (operatore * o –>) all'interno del corpo della funzione
- Passando al momento della chiamata della funzione come parametro attuale un indirizzo di variabile
  - Usando eventualmente l'operatore &
  - E così scopriamo finalmente il perché di & nella scanf

Esempio: main()

```
main() {
ElencoFatture                ArchivioFatture5;
Data                         DataOdierna;
DescrizioneFatture           Fattura1, Fattura2;

void InserisciFattura (DescrizioneFatture Fattura, ElencoFatture *PointToArchivioFatture);
boolean Precede (Data Num1,  Data Num2);

InserisciFattura (Fattura1, &ArchivioFatture5);
…
if (Precede (Fattura2.DataEmissione, DataOdierna)
        InserisciFattura (Fattura2, &ArchivioFatture5);
}
```

Prototipi delle funzioni e procedure a seguire…

Viene fornito un riferimento ad ArchivioFatture5, ma potrebbe essere un altro…

Esempio: corpo funzione

Uguale al prototipo…

```
void    InserisciFattura (DescrizioneFatture Fattura, ElencoFatture *PointToArchivioFatture) {
        if (PointToArchivioFatture–>NumFatture == MaxNumFatture)
                printf ("L'archivio è pieno.\n");
        else {
                PointToArchivioFatture–>NumFatture = PointToArchivioFatture–>NumFatture + 1;
                PointToArchivioFatture–> Sequenza[PointToArchivioFatture–>NumFatture – 1] = Fattura;
        }
}
```

Dobbiamo dereferenziare il puntatore per accedere al dato!

Qualche dettaglio

- La struttura generale di un programma C
  - una parte **direttiva**
  - una parte **dichiarativa globale** che comprende
    - dichiarazioni di costanti
    - dichiarazioni di tipi
    - dichiarazioni di variabili
    - prototipi di procedure e funzioni
  - un **programma principale**
  - definizioni di **funzioni o procedure**

Il concetto di blocco in C

- Un blocco è composto da due parti sintatticamente 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 C
  - I blocchi possono risultare paralleli o annidati
- La raccomandazione è di non abusarne ...

Esempio di struttura “complessa”

```
#define N 100
int g1, g2;
char g3;
int f1 (int par1, int par2);        ...
main () {
        int a, b;
        int f2 (int par3, int par1);  ...

        {char a, c;
         ...

                { float  a;
                  … }
        }
}
```

Parte direttiva

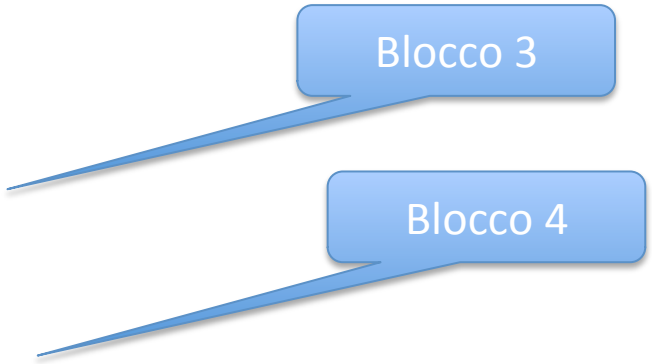
Parte dichiarativa globale

Blocco 1, char a maschera altre variabili con lo stesso nome!

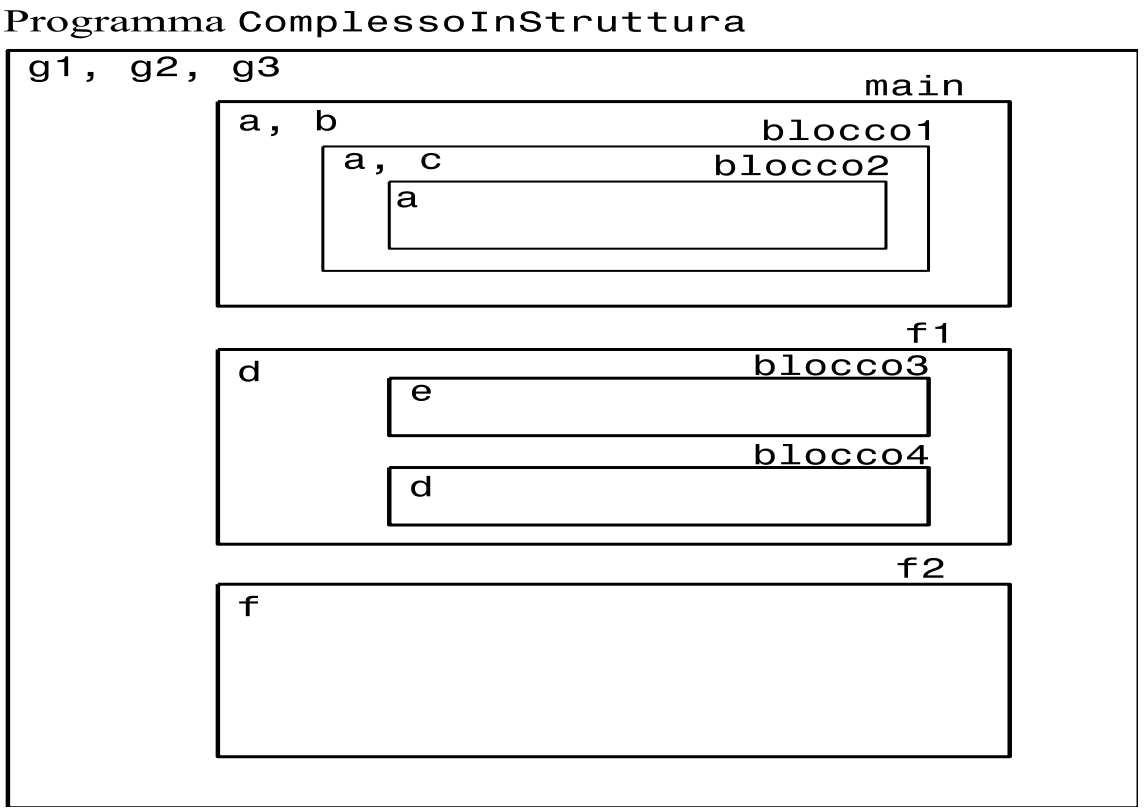
Blocco 2 annidato nel blocco 1: float a maschera altre variabili con lo stesso nome!

Esempio di struttura “complessa”

```
int f1(int par1, int par2) {
        int d;
        ...

        { int e;
          ... }

        { int d;
          ... }
}

int f2 (int par3, int par4) {
        int f;
        ...
}
```



Modello a contorni



Regole di visibilità

- Il main può accedere alle variabili globali g1, g2 e g3 e a quelle locali a e b
  - Il main può inoltre chiamare sia la funzione f1 sia la funzione f2
- Blocco1 può accedere alle variabili globali g1, g2, g3, alla variabile locale al main b e alle variabili a e c del proprio ambiente (a è stata ridefinita da blocco1)
  - Il blocco blocco1 può inoltre richiamare sia f1 sia f2
- Blocco2 può accedere alle variabili globali g1, g2, g3, alla variabile locale al main b, alla variabile del blocco blocco1 c e alla variabile a del proprio ambiente (a è stata ridefinita da blocco2)
  - Il blocco blocco2 può inoltre richiamare sia f1 sia f2

Regole di visibilità

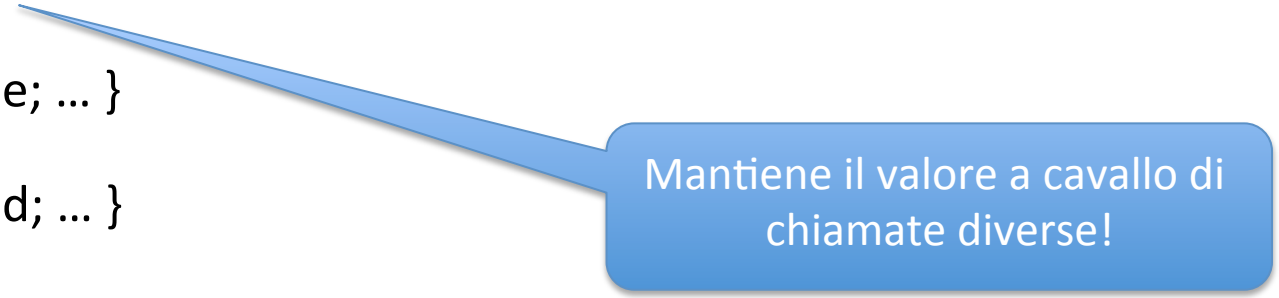
- La funzione f1 può accedere alle variabili globali g1, g2 e g3 e a quella locale d
  - La funzione f1 può inoltre chiamare la funzione f2 @@@CHECK o richiamare se stessa ma non può accedere alle variabili a, b, c, f
- Blocco3 può accedere alle variabili globali g1, g2, g3, alla variabile d, locale a f1, e alla variabile e del proprio ambiente
  - Il blocco blocco3 può inoltre richiamare sia f1 sia f2@@@CHECK
- Blocco4 può accedere alle variabili globali g1, g2, g3, e alla variabile d del proprio ambiente (d è stata ridefinita da blocco4)
  - Blocco4 non può invece accedere alla variabile e del blocco blocco3
  - Blocco4 può inoltre richiamare sia f1 sia f2@@@CHECK
- La funzione f2 può accedere alle variabili globali g1, g2 e g3 e a quella locale f
  - La funzione f2 può chiamare la funzione f1 o richiamare se stessa e non può accedere alle variabili a, b, c, d, e

La durata delle variabili

- Variabili globali o statiche (**static**): i loro valori vengono mantenuti anche all'esterno del loro ambito di visibilità
- Variabili automatiche (**auto**): i loro valori non vengono mantenuti all'esterno del proprio ambito di visibilità
- Quando il controllo passa a una particolare funzione (o blocco) viene allocato in memoria lo spazio per le variabili locali e viene rilasciato quando si ritorna all'ambiente chiamante

Però

```
int f1(int par1, int par2) {
        static int d;

        { int e; … }

        { int d; … }
}
```



- E' possibile usare la variabile d (a durata fissa) ad esempio per contare il numero di chiamate effettuate alla funzione f1 durante l'esecuzione del programma globale
- La variabile d è allocata in memoria quando f1 viene chiamata per la prima volta ma non viene distrutta al termine dell'esecuzione della funzione
- Raccomandazione finale: usare i meccanismi e le regole di visibilità e di vita delle variabili in modo semplice: pochi o niente annidamenti, poche o nessuna variabile statica

Parametri di tipo array

- Quando un array viene passato a una funzione come parametro formale, l'indirizzo di base dell'array viene passato "per valore" alla funzione
- Di fatto si realizza quindi un passaggio di parametro "per indirizzo": il parametro formale dichiarato nella testata della funzione viene trattato come puntatore

- **typedef** **double** TipoArray[MaxNumElem]

- le tre testate di funzione riportate di seguito sono in C di fatto equivalenti:

- **double** sum (TipoArray a, **int** n)
- **double** sum (**double** *a, **int** n)
- **double** sum (**double** a[ ], **int** n)

Il parametro n comunica alla funzione la dimensione dell'array

Esempio

```
double mul (double  a[ ], int n) {
        int    i = 0; double ris;
        ris = 1.0;
        while (i < n; i=i+1) {
                ris = ris * a[i]; i = i+1;
        }
        return ris;
}
```

Supponiamo che nel main sia stata dichiarata una variabile v di tipo array di 50 elementi:

Chiamata	Valore calcolato e restituito
mul (v, 50)	v[0]*v[1]* ... *v[49]
mul (v, 30)	v[0]*v[1]* ... *v[29]
mul (&v[5], 7)	v[5]*v[6]* ... *v[11]
mul (v+5, 7)	v[5]*v[6]* ... *v[11]

- Una funzione C non può mai restituire un array, ma soltanto un puntatore all'array
- Gli array in C sono un "vestito cucito addosso ai puntatori"
- Molta attenzione e rigore: evitare le tentazioni alle scorciatorie offerte dal C

Parametri di tipo struttura invece

- Una struttura può essere passata per valore anche quando contiene un componente di tipo array
- In tal caso, viene copiato nel parametro formale, l'intero valore del parametro attuale, contenuto dell'array, incluso
- Una funzione C può anche restituire per valore o per indirizzo una struttura (anche quando contiene un componente di tipo array)

Effetti collaterali (1/3)

Finora abbiamo distinto in maniera netta l'uso delle procedure da quello delle funzioni, ma il C non è così "puro"…

```
int PrimoEsempio (int par) {
      return (par + x)
}
```

Nel corso del seguente frammento di programma:

```
x = 1;
x = PrimoEsempio (1);
x = PrimoEsempio (1);
```

la sua chiamata produce, la prima volta, il risultato "2",
la seconda volta "3"

Effetti collaterali (2/3)

```
int SecondoEsempio (int *par) {
    *par = *par + 1;
    return *par;
}
```

y = SecondoEsempio (&z) assegna alla variabile y il valore di z+1, ma anche z assume lo stesso valore (**side effect**)!

Effetti collaterali (3/3)

```
int TerzoEsempio (int par) {
    x = par + 2;
    return (par + 1);
}
```

Altro **side effect**: z = TerzoEsempio (4) assegna a z il valore 5, ma anche il valore 6 a x

Uso interscambiabile di procedure e funzioni

```
int f(int par1) {
        ...
        return      risultato;
}
```

Essa può essere trasformata nella procedura seguente:

```
void f(int par1, int *par2) {
        ...
        *par2 = risultato;
}
```

Successivamente, una chiamata come

y = f (x) verrà trasformata in f(x, &y)

Standard library

- Operazioni di ingresso/uscita
  - Operazioni su file
  - Operazioni per la gestione dell'I/O di stringhe e caratteri
  - Operazioni di I/O formattato, operazioni di I/O non formattato, operazioni per la gestione degli errori
- Operazioni matematiche e aritmetiche: operazioni trigonometriche, operazioni esponenziali e logaritmiche e l'operazione per il calcolo del valore assoluto
- Operazioni di gestione della memoria
- Operazioni di gestione di caratteri e di gestione di stringhe
- Operazioni di ricerca e ordinamento applicabili ad array, operazioni di gestione di date e tempo, operazioni di utilità generale quali la generazione di numeri casuali
- Operazioni di comunicazione con l'ambiente del sistema operativo e una operazione per la gestione degli errori che hanno provocato un fallimento nell'esecuzione di una funzione

Concatenare due stringhe

```
#include <stdio.h>
#include <string.h>
#define LunghezzaArray 50
main() {
    char PrimaStringa[LunghezzaArray], SecondaStringa[LunghezzaArray],
        StringaConc[2 * LunghezzaArray];
    unsigned int LunghezzaConc;
    scanf ("%s", PrimaStringa);
    scanf ("%s", SecondaStringa);
    if (strcmp (PrimaStringa, SecondaStringa) <= 0 {
        strcpy (StringaConc, PrimaStringa);
        strcat (StringaConc, SecondaStringa);
    } else {
        strcpy (StringaConc, SecondaStringa);
        strcat (StringaConc, PrimaStringa);
    }
    LunghezzaConc = strlen (StringaConc);
    printf("La stringa ottenuta concatenando le due stringhe lette è %s.
                Essa è lunga %d caratteri\n", StringaConc, LunghezzaConc);
}
```

I file header

- Le funzioni della libreria sono disponibili in C come file di codice compilato, non leggibili direttamente dal programmatore
- È comunque compito del programmatore inserire nel programma i prototipi delle funzioni che verranno usate
- Per facilitare questo compito, la libreria C comprende alcuni file, chiamati **header file**, che contengono (solo) i prototipi di un insieme di funzioni di libreria
- Ciò spiega la #include <stdio.h> e le altre #include <xxx.h>
- Il preprocessore copia il contenuto del file stdio.h nel programma, inserendo i prototipi delle funzioni che appartengono al gruppo di cui xxx.h è il file testata