Istruzioni

Istruzioni

- Si dividono in:
  - blocchi;
  - espressioni;
  - istruzioni di selezione;
  - istruzioni di ciclo;
  - istruzioni di salto.

Blocco

- Le istruzioni sono raggruppate in blocchi, racchiusi tra parentesi graffe.
  - Ogni istruzione o dichiarazione termina con punto e virgola.
    - Si noti che `;` è un terminatore di istruzione, **non** un separatore, come in Pascal.
  - Il `;` isolato è legale e considerato un'istruzione vuota.
  - Le istruzioni sono eseguite nell'ordine in cui sono scritte.
  - Un blocco può rimpiazzare una singola istruzione.
  - Un blocco può essere vuoto.
  - L'intero corpo di una funzione è un blocco.

Dichiarazioni in un blocco

- Un blocco può contenere dichiarazioni e istruzioni, mescolate a piacere.
  - Si raccomanda però di mettere sempre tutte **le dichiarazioni prima delle istruzioni**.
- Le variabili dichiarate in un blocco sono per default automatiche, vale a dire allocate sullo stack all'ingresso del blocco e distrutte all'uscita.
  - Il loro valore iniziale, in assenza di inizializzazioni esplicite, è **indefinito**.
  - Le variabili locali sono visibili **dalla dichiarazione** alla fine del blocco.

Espressioni come istruzioni

- Un’espressione seguita da punto e virgola è un’istruzione.
  - Anche se l’espressione non ha senso: `a + b;` e `1;` sono istruzioni legali!
  - Pertanto il compilatore non segnala errore se si omette l’assegnamento o lo si sostituisce con un altro operatore, per esempio scrivendo:

    ```
    a + b;
    ```

    invece di

    ```
    a += b;
    ```

Esempi di espressioni come istruzioni

```
x = 0;

i++;

printf("ciao\n");

;          // espressione vuota

a + b;   // insensata, ma ammessa
```

Label

- Ogni istruzione può essere identificata da una label: un identificatore seguito dal carattere due punti.
  - Serve come “bersaglio” per un’istruzione `goto`.
  - Una label deve essere unica nel suo blocco.

Esempio di label

```
if (x < 0)
   goto error;
…
error: print_error_message();
```

Istruzioni di selezione

- Permettono di selezionare un'alternativa tra due o più.
  - Sono due: `if` e `switch`.

Istruzione if

- Ha la sintassi:

  `if` (<espressione>) <istruzione-1>

  [`else` <istruzione-2>]

- Viene valutata l'espressione, poi si esegue istruzione-1, se l'espressione è vera, istruzione-2 altrimenti.
  - Le parentesi tonde sono obbligatorie.
  - L'`else` non è obbligatorio.

Esempi di if

```
if (a != 0)     // b = a != 0? a: 1;
     b = a;
else
     b = 1;
```

Equivalente, ma meno chiaro:

```
if (a)          // b = a? a: 1;
     b = a;
else
     b = 1;
```

Uso di else

- Ogni `else` è associato all'`if` libero più vicino, nello stesso blocco.
- Volendo un'associazione diversa, bisogna usare le parentesi graffe per creare nuovi blocchi.
- L'indentazione va usata per chiarire le intenzioni e facilitare la lettura.

Esempi di uso di else

```
if (a > 0)   // Primo if
   if (b > 0)     // Secondo if
   …
   else       // Associato al secondo if

if (a > 0)   // Primo if
   {
   if (b > 0)     // Secondo if
   …
   }
else          // Associato al primo if
```

Istruzione switch

- Ha la sintassi:

  `switch` (<espressione>)

  <istruzione>

- L'istruzione è normalmente un blocco formato da gruppi di istruzioni, preceduti da una delle due label:

  `case` <espressione costante> `:`

  `default:`

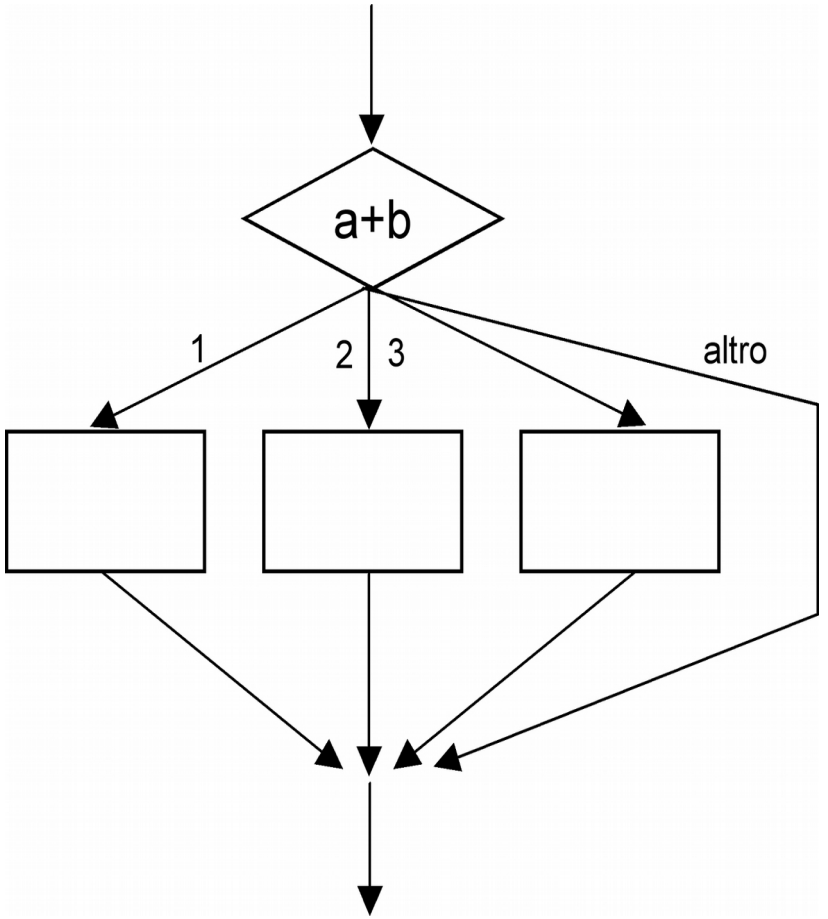
Esecuzione dello switch

- Viene valutata l'espressione, quindi:
  - se il valore è uguale a quello di una delle espressioni costanti, il controllo passa alla prima istruzione dopo il `case` corrispondente;
  - altrimenti il controllo passa alla prima istruzione dopo il `default`;
  - in assenza di `default`, il controllo passa alla prima istruzione dopo il blocco.
- In assenza di trasferimento esplicito di controllo (p. es., `break`), vengono eseguite **tutte le istruzioni successive**.

Switch C e case Pascal (1)

- Pascal:

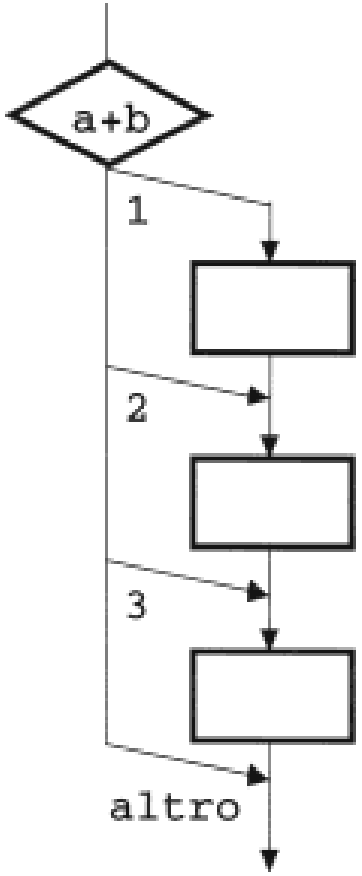
```
case a + b of
  1: …
  2: …
  3: …
end
```



Switch C e case Pascal (2)

- C:

```
switch (a + b)
    {
    case 1: …
    case 2: …
    case 3: …
    }
```

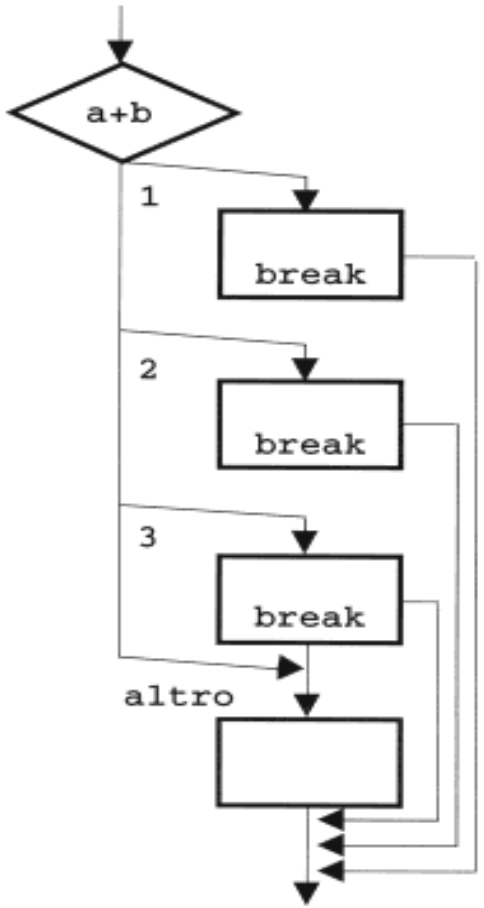


PIÙ FLESSIBILE

Switch C e case Pascal (2)

- Per ottenere la semantica Pascal in C:

```
switch (a + b)
      {
      case 1: …
            break;
      case 2: …
            break;
      case 3: …
            break;
      default: …
      }
```



Vincoli dello switch

- L’espressione e le costanti devono essere di tipo intero.
  - L’espressione viene sempre promossa a intero.
- I valori delle costanti devono essere tutti diversi.
  - Le costanti sono convertite al tipo dell’espressione prima di effettuare i confronti.
  - Pertanto si possono usare costanti di tipi enumerati.
- E’ ammesso al massimo un `default`.

Esempio di switch (1)

Programma che conta cifre, spazi e altri caratteri in input.

```
# include <stdio.h>
# include <stdlib.h>
int main(void)
  {
  int               c;
  unsigned long     nwhite, nother, ndigit;

  nwhite = nother = ndigit = 0;
  while ((c = getchar()) != EOF)
```

Esempio di switch (2)

```
switch (c)
     {
     case '0': case '1': case '2':
     case '3': case '4': case '5':
     case '6': case '7': case '8': case '9':
          ++ndigit;
          break;
     case ' ': case '\n': case '\t':
          ++nwhite;
          break;
     default:
          ++nother;
          break;
     }
```

Esempio di switch (3)

```
printf("Cifre: %lu, spazi: %lu, altri: %lu\n",
     ndigit, nwhite, nother);
return EXIT_SUCCESS;
}
```

Istruzioni di ciclo

- Permettono eseguire piu volte un'istruzione.
  - Sono tre: `while`, `do while` e `for`.

Istruzione while

- Ha la sintassi:

  `while` (<espressione>) <istruzione>
- Viene valutata l'espressione: se è vera, si esegue istruzione, ripetendo il ciclo; se è falsa si esce.
  - Le parentesi tonde sono obbligatorie.

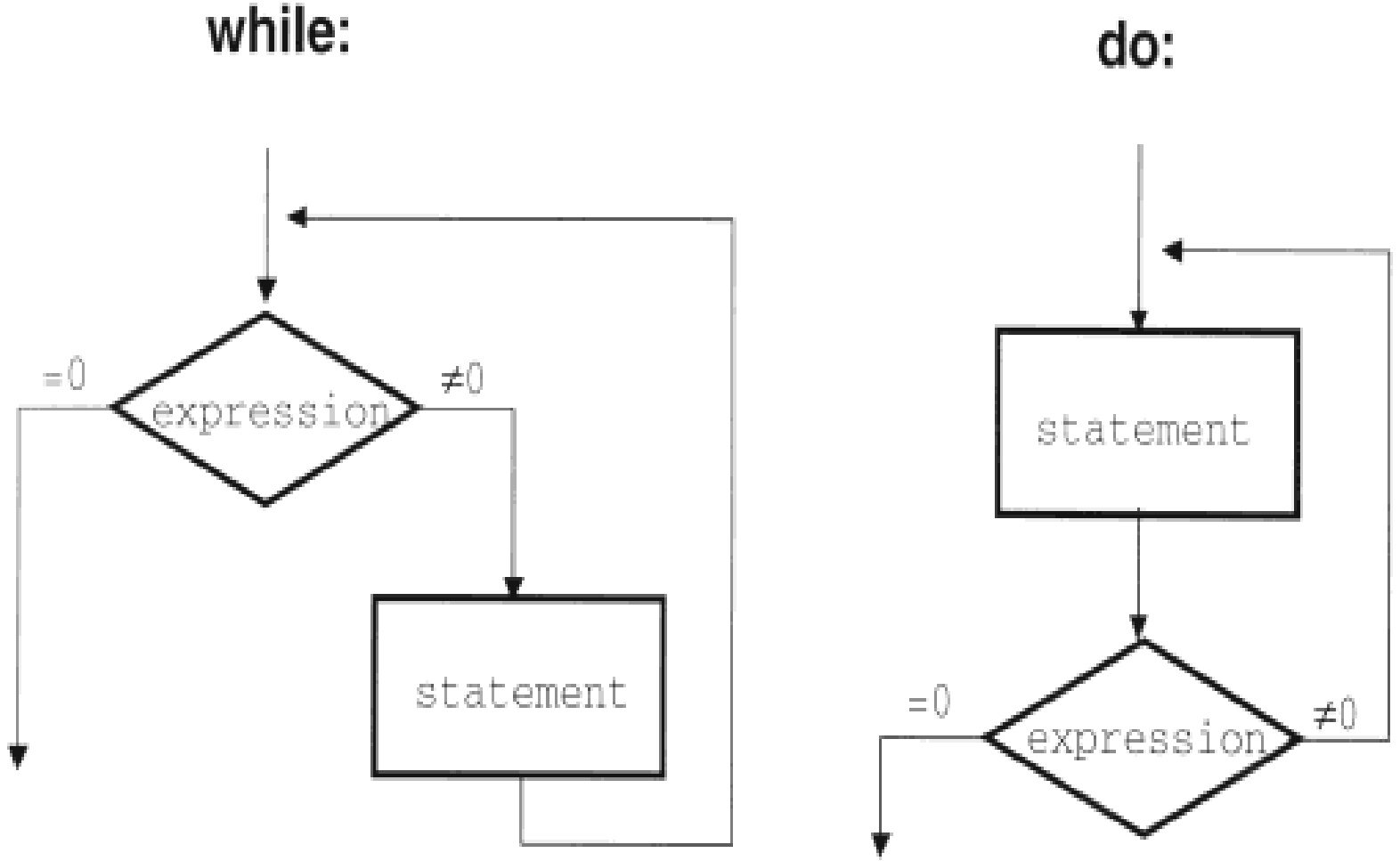
Istruzione do … while

- Ha la sintassi:

  `do` <istruzione> `while ( ` <espressione> `) ;`

- Viene eseguita istruzione, poi viene valutata l'espressione: se è vera, si ripete il ciclo; se è falsa si esce.
  - Le parentesi tonde sono obbligatorie.

Differenza tra le forme di ciclo



Esempio di ciclo (1)

Leggi e conta caratteri in input fino a `EOF`.

```
ctr = 0;
ch = getchar();
while (ch != EOF)
    {
    ++ctr;
    ch = getchar();
    }
```

Esempio di ciclo (2)

Versione alternativa.

```
ctr = -1;
do
      {
      ch = getchar();
      ++ctr;
      } while (ch != EOF);
```

Meglio:

```
while ((ch = getchar()) != EOF)
      ++ctr;
```

Istruzione for

- Ha la sintassi:

  `for` `(` [<for-init>] `;` [<espressione>] `;` [<for-update>] `)` <istruzione>
- For-init è **un'espressione** o **una dichiarazione** di variabili con inizializzazioni.
  - Le variabili sono visibili fino alla fine dell'istruzione `for`.
- For-update è **un'espressione**.
- Le parentesi tonde sono obbligatorie.
- Le tre espressioni possono mancare, in qualsiasi combinazione.

Esecuzione dell'istruzione for (1)

- Viene valutato for-init, quindi viene valutata l'espressione: se è vera, si esegue l'istruzione, si valuta for-update e di ripete il ciclo; se è falsa si esce.
- Se la seconda espressione è omessa, è come se fosse sempre vera.

  `for (;;)` indica un ciclo infinito.

Esecuzione dell'istruzione for (2)

```
for (expr1; expr2; expr3)
      statement
```

Equivale a:

```
expr1;
while (expr2)
      {
      statement
      expr3;
      }
```

Esempi di for

```
for (i = 0; i < 50; ++i)

for (int i = 0; i < 50; ++i)

for (i = 0, j = n; i < j; ++i, --j)

for (int i = 0, j = n; … ; …)
```

for errato (due dichiarazioni)

```
for (int i = 0, int j = 5; … ;…)
```

For in C e in Pascal

Pascal:

```
for i := 1 to n do
```

…

Il limite del ciclo è calcolato **una sola volta**; l'indice non può essere modificato all'interno del ciclo, è **indefinito** all'uscita.

C:

```
for (i = 1; i <= n; ++i)
```

…

Il limite del ciclo è **riesaminato** a ogni iterazione; l'indice è **liberamente gestito** dal programmatore nel ciclo, è **definito** all'uscita.

Esempio di for

Inverti l'ordine dei caratteri di una stringa.

```
void reverse(char s [])
      {
      register    char              c;
      register    unsigned int      i;
      register    unsigned int      j;

      for (i = 0, j = strlen(s) - 1; i < j; ++i, --j)
            {
            c = s [i];
            s [i] = s [j];
            s [j] = c;
            }
      }
```

Istruzioni di salto

- Permettono saltare a un punto nel codice.
  - Sono quattro: `goto`, `break`, `continue` e `return`.

Istruzione goto

- Ha la sintassi

  `goto` <identificatore> `;`
- Permette di saltare a una label nella stessa funzione.
- **Se ne sconsiglia assolutamente l'uso!**

Istruzione break

- Ha la sintassi:

  `break;`
- Provoca l'uscita dal ciclo o `switch` più interno.
  - E' ammessa solo entro cicli o `switch`.

Istruzione continue

- Ha la sintassi:

  `continue;`
- Provoca il passaggio alla successiva iterazione, valutando l'espressione condizionale.
  - Se entro un `for`, la parte for-update viene eseguita.
- Si riferisce sempre al ciclo più interno.
  - E' ammessa solo entro cicli.

Esempio di continue

```
for (i = 0; i < n; ++i)
        {
            // Tratta solo elementi non
            // negativi
        if (a [i] < 0)
            continue;
        …
        }
```

Istruzione return

- Ha la sintassi:

  `return` [<espressione>]`;`
- Provoca l'arresto dell'esecuzione della funzione corrente e il ritorno a quella chiamante.
- L'espressione, se presente, viene valutata e il valore diventa il valore di ritorno della funzione.
  - L'espressione è **vietata** nelle funzioni `void`, **obbligatoria** nelle altre.

Funzioni

Dichiarazione di funzione

- Ha la sintassi

  [<classe di memoria>] <tipo>

  <nome> `(` <lista argomenti> `);`
- Il tipo può essere un qualsiasi tipo semplice o strutturato (anche `void`), tranne i vettori.
  - Il tipo `void` indica che la funzione non produce valore.
- La dichiarazione deve sempre precedere l'utilizzo.

Argomenti formali delle funzioni

- Sono liste di dichiarazioni della forma

  [<classe di memoria>] <tipo> <nome>

  separate da virgole.
  - La classe di memoria è facoltativa e può essere solo `auto` o `register`.
  - Il tipo può essere qualsiasi tipo, anche strutturato, esclusi i vettori.
  - Il nome è facoltativo nelle dichiarazioni.
- Se non ci sono argomenti, si scrive `void` al posto della lista.

Definizione di funzione

- Ha la sintassi della dichiarazione, con `;` sostituito da un blocco di istruzioni e un eventuale `inline` davanti.

  [`inline`] [<classe di memoria>] <tipo>

  <nome> (<lista argomenti>) <blocco>
- La definizione e dichiarazione devono coincidere per classe di memoria, tipo, nome della funzione e lista di argomenti.
  - Gli argomenti della dichiarazione devono coincidere ordinatamente per tipo e numero con quelli della definizione.
    - Il nome è irrilevante e può essere omesso in dichiarazione, ma è bene che ci sia e coincida, per chiarezza.

Esempio di funzioni

- Dichiarazioni:

```
void    f(void);
double  sqrt(double x);
double  sqrt(double);  // Equivalente
```

- Definizioni:

```
void f(void)
   {}   // Funzione vuota

double sqrt(double x)
   {
   // Corpo da completare,
   // lasciato come esercizio.
   }
```

Chiamata di funzioni

- Dopo il nome si scrivono gli argomenti attuali, tra parentesi e separati da virgole.
  - La chiamata assegna **ordinatamente** gli argomenti attuali a quelli formali.
    - Il compilatore verifica la corrispondenza dei tipi.
    - Se sono di tipo diverso, vengono convertiti, con le stesse regole dell'assegnamento.
  - Il valore prodotto dalla funzione con l'istruzione `return` è il valore finale dell'espressione.

Esempio

- Calcolo delle potenze di un intero.

```
int power(int base, unsigned int exp)
   {
   int  value;

   value = 1;
   for (; exp > 0; --exp)
        value *= base;
   return value;
   }
```

- Chiamata:

```
n = power(k, 3);
```

Ordine di valutazione degli argomenti (1)

- L'ordine di valutazione degli argomenti di una funzione è **indefinito**.
  - È solo garantito che tutti vengano valutati prima della chiamata della funzione.
- Ricordate che il **separatore** virgola non è l'**operatore** virgola, quindi non forza un ordine di valutazione.

Ordine di valutazione degli argomenti (2)

- Pertanto codice tipo:

```
for (i = 0; i < NUM_ELEMENTS;
    printf("%d, %d\n", i, i++));
```

può scrivere:

```
0,  0
1,  1
```

ecc.

oppure:

```
1,  0
2,  1
```

ecc.

Prototipi delle funzioni

- Lo Standard impone ora di dichiarare e definire le funzioni con **prototipi**, che permettono un più stretto controllo sugli argomenti.
  - I prototipi vanno utilizzati **sempre**.
- Il "vecchio" stile di dichiarazione e definizione, ufficialmente dichiarato obsolescente, resta però valido e può creare inconvenienti, se inavvertitamente utilizzato.

Esempio di funzioni “vecchio stile”

Dichiarazione:

```
int f();
```

Definizione:

```
int f(i, x)
    int     i;
    double  x;
    {
    …
    }
```

Uso dei prototipi

- **Attenzione:** lo Standard non garantisce un comportamento corretto se una funzione viene **definita** con il prototipo in un file e **dichiarata** senza in un altro o viceversa.
  - In alcune implementazioni può capitare che il programma abbia comportamenti imprevedibili.
  - Il compilatore può generare **sequenze di chiamata differenti** per funzioni con e senza prototipo.
- Le regole di conversione degli argomenti per funzioni con e senza prototipo sono **differenti**.

Prototipo di funzione senza argomenti

- Attenzione:

```
int f();
```

  non è la dichiarazione di una funzione **senza argomenti**, ma la dichiarazione **vecchio stile** di una funzione (senza **controllo sugli argomenti**).

- Una funzione senza argomenti va dichiarata così:

```
int f(void);
```

Funzioni con numero variabile di argomenti

- Nel prototipo e nella definizione va specificata l'intenzione di passare un numero variabile di argomenti con il simbolo `...` dopo l'ultimo argomento.
  - Deve esistere **almeno un argomento fisso.**
  - L'ultimo argomento fisso non deve essere dichiarato `register`.
  - Per accedere agli argomenti variabili vanno utilizzate le macro definite nel file stdarg.h.

File stdarg.h

- Contiene le seguenti definizioni:
  - `va_list`: tipo della variabile contenente le informazioni necessarie per accedere agli argomenti;
  - `va_start`: macro da utilizzare per inizializzare le variabili di tipo `va_list`;
  - `va_arg`: macro da utilizzare per accedere al successivo argomento;
  - `va_end`: macro da utilizzare obbligatoriamente per concludere l'accesso agli argomenti variabili.

Accesso agli argomenti variabili

- L'accesso agli argomenti variabili deve essere fatto nel seguente modo:
  - si definisce una variabile di tipo `va_list`;
  - la si inizializza con la macro `va_start`;
  - si accede agli argomenti, uno alla volta e in ordine, con la macro `va_arg`;
  - si esegue la macro `va_end`.
- Per un eventuale secondo accesso, si deve ricominciare con una nuova inizializzazione.

Esempio di funzione a numero variabile di argomenti (1)

```
void    f(const char* argument, ...)
    {
    va_list  argument_ptr;
    int      n;
    double   x;

    va_start(argument_ptr, argument);
```

Il secondo parametro di `va_start` deve essere l'ultimo argomento fisso.

Esempio di funzione a numero variabile di argomenti (2)

```
n = va_arg(argument_ptr, int);
```

Il secondo parametro di `va_arg` deve essere il tipo dell'argomento.

```
x = va_arg(argument_ptr, double);
…
va_end(argument_ptr);
}
```

Esempio di funzione a numero variabile di argomenti (3)

La funzione potrà essere chiamata così:

```
f("abc", 9, 12.0);
```

ma anche così:

```
f("abc", (char)'x', (float)1.0);
```

perché comunque `(char)'x'` viene convertito a `int` e la costante `float` a `double`.

Pericoli delle funzioni con numero variabile di argomenti

- Se durante l'esecuzione si tenta di accedere a più argomenti di quelli effettivamente passati o se si tenta di accedere specificando un tipo diverso da quello effettivo, il **comportamento è indefinito** e le conseguenze sono imprevedibili.
  - Non esiste modo durante l'esecuzione di sapere quanti siano gli argomenti **effettivamente passati** né il loro tipo: il programmatore deve stabilire qualche convenzione e **rispettarla**.
  - Nell'accedere agli argomenti variabili bisogna tenere conto del fatto che sono soggetti alle regole di conversione delle funzioni dichiarate senza prototipo.

Conversioni degli argomenti delle funzioni

- Valgono due insiemi distinti di regole.
  - Funzioni dichiarate **con prototipo**: gli argomenti vengono **convertiti**, se possibile, al tipo richiesto. Se la conversione è impossibile o se il numero degli argomenti è sbagliato, il compilatore segnala l'errore.
  - Funzioni dichiarate **senza prototipo**: gli argomenti vengono **passati così come sono**, tranne che argomenti `_Bool`, `short int` e `char` sono promossi a interi e gli argomenti `float` sono convertiti a `double`. Se il numero degli argomenti è errato, o il tipo dell'argomento formale non è uguale a quello dell'argomento attuale, il comportamento è indefinito.

Argomenti variabili

- Gli argomenti in numero variabile (corrispondenti al simbolo ...) delle funzioni a numero variabile di argomenti vengono passati come quelli delle funzioni senza prototipo.
  - Quindi così come sono, tranne per la promozione a intero e la conversione di `float` a `double`.

Esempi di conversione di argomenti

```
double    sqrt(double);
double    sin();
double    x;
int       i;
float     f;
```

`x = sqrt(i);` `i` viene convertita a `double`.

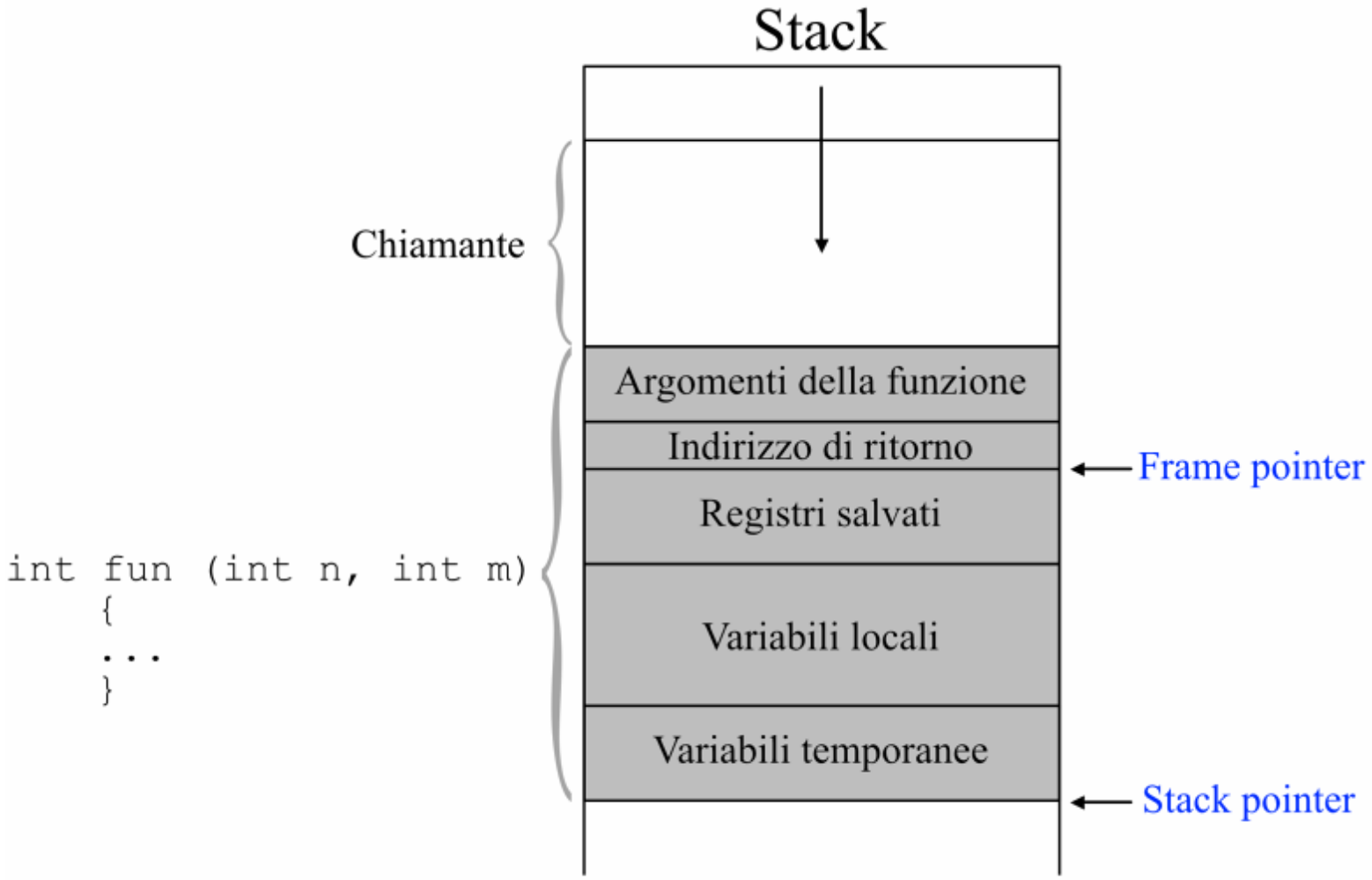
`x = sin(i);` `i` non viene convertita a `double` e si ha un errore.

`x = sin(f);` `f` viene convertita a `double`.

Stack e passaggio argomenti (1)

- Alla chiamata di una funzione viene allocata sullo stack un'area (stack frame) contenente:
  - gli argomenti;
  - l'indirizzo di ritorno;
  - la copia dei registri utilizzati;
  - le variabili locali;
  - eventuali variabili temporanee.
- L'area viene poi liberata all'uscita.

Stack e passaggio argomenti (2)



Stack e passaggio argomenti (3)

- Dato che la funzione opera su **copie** dei valori, eventuali assegnamenti agli argomenti entro la funzione non hanno effetto sul chiamante.
- Se una funzione deve modificare variabili del chiamante, bisogna utilizzare i puntatori.

Ricorsione

- Una funzione può chiamare se stessa, sia direttamente, sia indirettamente.
- A ogni chiamata viene allocato un nuovo stack frame, con una nuova copia degli argomenti e delle variabili locali.
- I frame vengono poi distrutti in ordine inverso, uscendo dalle funzioni.

Esempio di ricorsione

Calcolo del fattoriale:

```
unsigned long long
        factorial(unsigned int n)
        {
        if (n == 0)
            return 1;
        return n * factorial(n - l);
        }
```

Istruzione return

- Restituisce il controllo al chiamante, producendo il valore dell'espressione come valore della funzione.
  - Il valore viene convertito al tipo della funzione, se di tipo diverso, con le stesse regole dell'assegnamento.
  - Il valore viene **copiato** nel frame del chiamante.
    - Non deve essere un puntatore a una variabile locale della funzione.
  - L'espressione non è permessa nelle funzioni `void`.

Uscita dalle funzioni

- Nelle funzioni `void` si può uscire arrivando semplicemente alla fine del blocco.
- Se si esce arrivando alla fine del blocco di una funzione non `void`, il **comportamento** è indefinito.

Valore delle funzioni

- Se non serve, può essere ignorato senza alcuna conseguenza.
  - Meglio scrivere un cast esplicito a `void` per chiarire l'intenzione.
  - Per esempio:

```
printf("hello\n");
```

ignoro il valore prodotto dalla `printf`, che è il numero di caratteri scritti, se positivo, o un numero negativo in caso d'errore.

Meglio:

```
(void)printf("hello\n");
```

Funzioni inline (1)

- Specificando `inline` una funzione, si **suggerisce** al compilatore di espanderne il codice in linea al posto della chiamata.
  - Il codice diventa in genere più lungo, ma più veloce.
  - Ha senso per **piccole** funzioni molto semplici.
- Il compilatore decide caso per caso se espandere o meno la chiamata.
  - Può farlo anche solo per alcune chiamate di una funzione.
  - Non rende nota la scelta in alcun modo.

Funzioni inline (2)

- Una funzione `inline` si comporta **a tutti gli effetti** come una funzione qualsiasi, salvo che se non è dichiarata `extern`, **non è riferibile** da altre unità di compilazione.
  - Se tutte le unità di compilazione la dichiarano `inline` e non `extern`, può risultare non definita per il linker.

Vincoli sulle funzioni inline

- Una funzione `inline` non `static` non deve:
  - contenere variabili `static` e non `const`;
  - riferire funzioni o variabili `static`.
- Se definita in più file e non `static`, le definizioni devono essere **identiche**.
  - In pratica, la definizione si mette generalmente in file inclusi.
- Il `main` non deve essere dichiarato `inline`.