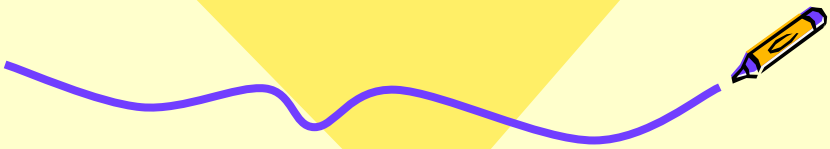
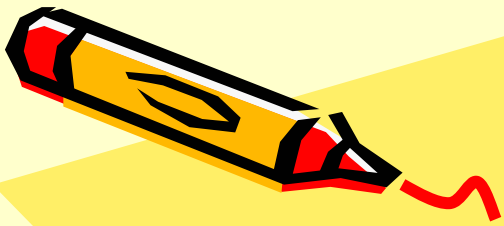
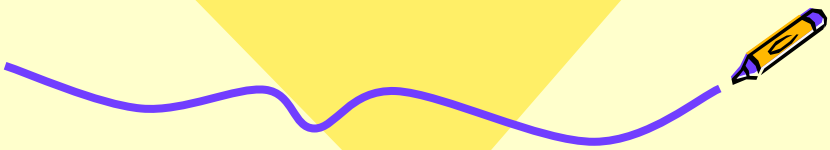
Laboratorio di Algoritmi e Strutture Dati

Aniello Murano
http://people.na.infn.it/~murano





Il linguaggio C



Indice (seconda parte)

- Funzioni
- Array
- Puntatori
- Preprocessore
- Storage Class
- Ricorsione

Funzioni

- Nel *C* è possibile scomporre problemi complessi in moduli più semplici sfruttabili singolarmente.
- Le funzioni sono blocchi di programmi indipendenti da altri moduli, ciascuno destinato ad una precisa operazione.
- Un programma nel *C* non è altro che una grossa funzione *main()* che ingloba nel suo interno altre funzioni.
- La comunicazione tra i diversi moduli avviene mediante gli argomenti, i valori di ritorno e le variabili esterne.
- L'uso delle funzioni consente di nascondere l'implementazione di una certa operazione e concentrarsi solo sul "***cosa fa***" e non sul "***come lo fa***".

Funzioni (2)

- Una funzione può:
  - compiere un azione
  - effettuare un calcolo
- Nel primo caso la sua invocazione provoca il verificarsi di una certa azione, nel secondo il risultato del calcolo viene restituito dalla funzione stessa.
- Una funzione viene definita nel seguente modo:
  - *tipo-ritornato nome_f (dichiarazione argomenti)*
  - *{dichiarazioni ed istruzioni }*

Funzioni (3)

- Ogni funzione presenta un valore di ritorno che può essere di qualsiasi tipo predefinito o definito dall'utente.
- Se nella dichiarazione viene omesso il tipo ritornato, esso viene considerato automaticamente un intero.
- Nel caso di funzione che compie un azione, ovvero non deve ritornare nessun valore, si usa il tipo predefinito ***void*** come valore di ritorno.
- Una funzione può avere o meno una lista di argomenti

Controllo dell'esecuzione

- All'atto della chiamata di una funzione il controllo nell'esecuzione viene passato alla prima istruzione del corpo della funzione stessa.
- Esistono due modi per restituire il controllo al programma chiamante:
  - attraverso l'istruzione: **return *espressione;***
  - termine dell'esecuzione della funzione "}"
- Nel primo caso l'espressione viene, se necessario, convertita al tipo ritornato dalla funzione.
- E' opportuno controllare sempre che la chiamata di una funzione ed il suo valore di ritorno siano consistenti.

Lista degli argomenti

- La lista argomenti è usata per passare dati ad una funzione chiamata. La lista argomenti può essere anche vuota.
- Le variabili da passare devono essere specificate tra parentesi dopo il nome della funzione.
- Nel corpo della funzione le variabili devono essere dichiarate con il loro tipo corrispondente.
- Esempio:

```
float Calcola ( int x, int y , int y)
{
      [ corpo della funzione ]
}
```

Esempio

```
int lower(int c)
{
   int k;
   k = (c >= 'A' && c <= 'Z') ? (c + 'a' - 'A') : (c + 'A' - 'a');
   return k;
}
main()
{
   printf("Dammi un carattere in input\n");
   c = getchar();
   printf("\nIl carattere %c convertito è %c\n",c,lower(c));
}
```


Argomenti di una funzione

- Il metodo di passaggio delle variabili è per valore, ossia, nella funzione chiamata si farà una copia locale delle variabili passate e non si potrà modificarne il loro valore globale, a meno di non passare alla funzione il "riferimento" della variabile.
- E' responsabilità del programmatore controllare che il numero ed il tipo degli argomenti passati ad una funzione corrisponda con quanto specificato nella dichiarazione della funzione stessa.
- Con l'introduzione del **prototipi** ad ogni funzione è assegnato un prototipo di chiamata, quindi il compilatore è in grado di controllare tipi e numero di argomenti passati.

Prototipi

- Il prototipo di una funzione non rappresenta altro che una dichiarazione di funzione antecedente alla sua definizione.

  int function(int x, char c); int function(int,char);
- Nell'ANSI C l'uso dei protoitpi è obbligatorio quando la definzione di una funzione avviene successivamente al suo utilizzo.
- Lo scopo dei **prototipi** delle funzioni è quello di permettere al compilatore di compiere il controllo sui tipi degli argomenti che vengono passati ad una funzione.

Prototipi

- Se i prototipi vengono inseriti prima della definizione di una funzione, quando il compilatore incontra la dichiarazione conosce il numero ed i tipi degli argomenti e può controllarli.
- La dichiarazione e la definizione di una funzione devono essere consistenti sia nel numero che nel tipo dei parametri.
- Attenzione se dichiarazione e definizione di funzione si trovano nello stesso file sorgente eventuali non corrispondenze nei tipi degli argomenti saranno rilevati dal compilatore, in caso contrario al più ci sarà un warning.

Prototipi

- **NOTA STORICA**: Nel vecchio C di Kernighan e Ritchie non esisteva l'uso dei prototipi in quanto la definzione di una funzione era differente.
- **Definizione di funzione ANSI C**

```
float volume( int lung, int larg, int alt) { ..... }
```

- **Definizione di funzione Kernighan & Ritchie**

```
float volume(lun,larg,alt)
int lung,arg,alt;
{ .... }
```

- Per questo motivo non era dato modo al compilatore di controllare sia il numero che il tipo degli argomenti, ciò generava non poche situazioni di errore.

Array

- Un array è una collezione di variabili dello stesso tipo che condividono un nome comune.
- Un array viene dichiarato specificando il tipo, il nome dell'array e uno o più coppie di parentesi quadre contenenti le dimensioni.
- Esempio:

```
int nomi[4]; /* 1 dimensione di 4 interi */
float valori[3][4] /* 2 dimensioni di 12 float */
char caratteri[4][3][5][7] /* 4 dimensioni 420 elementi */
```

Rappresentazione di Array

- Il C memorizza i valori degli elementi di un array in locazioni consecutive di memoria.

  - *locazione base* + 0 stanze[0] "3"
  - *locazione base* + 1 stanze[1] "2"
  - *locazione base* + 2 stanze[2] "0"
  - *locazione base* + 3 stanze[3] "2"
  - *locazione base* + 4 stanze[4] "1"
  - *locazione base* + 5 stanze[5] "4"
- E' molto importante conoscere come sono stati memorizzati gli elementi poiché questo consente di capire come sia possibile puntare ad un preciso elemento.

Dichiarazione

- In C è obbligatorio specificare in modo esplicito la dimensione di un array in fase di dichiarazione.
- In una definizione di funzione, come vedremo, non occorre specificare la dimensione del vettore passato come parametro; sarà il preprocessore a risolvere l'ambiguità all'atto della chiamata.
- Esempio:

```
int somma(int numeri[],dimensioni)
.....
main()
int num[3];
{
.....
totale = somma(num,3);
}
```

Array multidimensionali

- Gli array multidimensionali vengono dichiarati specificando il numero di elementi per ciascuna dimensione.
- Un array bidimensionale con 6 elementi per ciascuna dimensione viene dichiarato come:

```
int alfa[2][6];
```

- Per referenziare un singolo elemento è necessario utilizzare due coppie di parentesi quadre:

```
alfa[1][2] = 1;
```

- In pratica un array multidimensionale è una collezione di oggetti, ciascuno dei quali è un vettore.

Esempio

```
*/ Tabelline dall'1 al 10*/
main()
{
   int tabelline[10][10];
   int i, j;
   for (i = 0 ; i < 10 ; i++)
   for(j = 0 ; j < 10 ; j++)
   tabelline[i][j] = (i + 1) * (j + 1);
}
```

Indirizzamento

- Per accedere ad un array si usano gli **indici**.
- E' compito del programmatore fare in modo di non andare oltre i limiti di dimensione dell'array in questione.
- Gli array partono dall'indice 0 fino alla lunghezza dichiarata meno 1;
- Referenziare un array al di fuori dei suoi limiti può portare a errori di indirizzamento (memory fault) oppure può "sporcare" altre variabili in memoria diventando così molto difficile da localizzare.
- Per copiare elementi da un array all'altro bisogna copiare singolarmente ogni elemento.

Array di stringhe

- Una stringa è un array monodimensionale di caratteri ASCII terminati da un caratter *null* '\0'
- Ad esempio "**Questa è una stringa**" è un array di 21 caratteri.
- L'array è quindi il seguente:

  elemento zero 'Q'

  primo elemento 'u'

  secondo elemento 'e'

  .....

  ventesimo elemento 'a'

  ventunesimo elemento '\0'

Esempio

```
/* Stampa i caratteri e la codifica ASCII */
char stringa[] = "Questa è una stringa";
main()
{
   int i = 0;
   while ( stringa[i] != '\0' )
   {
     printf ( "%c = %d\n",stringa[i], stringa[i] );
     ++i;
   }
}
```


Array come argomenti di funzioni

- Il metodo di default di passaggio delle variabili è **per valore,** quindi si potrà modificare solo una copia locale della variabile.
- Eccezione a questa regola globale sono gli array.
- Gli array, infatti, vengono sempre passati **per reference** ossia è il loro indirizzo che viene passato invece del loro contenuto.
- Questo consente, solo nel caso dei vettori, di poter agire direttamente sulla variabile e non sulla copia. Ovvero le azioni che si effettuano sull'argomento della funzione avranno effetto anche al termine dell'esecuzione della funzione.

Puntatori

- Un **puntatore** è una variabile che contiene l'indirizzo di un'altra variabile.
- I puntatori sono "**type bound**" cioè ad ogni puntatore è associato il tipo a cui il puntatore si riferisce.
- Nella dichiarazione di un puntatore bisogna specificare un asterisco (*) di fronte al nome della variabile pointer.
- Esempio:

```
int *pointer;      puntatore a intero
char *pun_car;     puntatore a carattere
float *flt_pnt;    puntatore a float
```


Inizializzazione di puntatori

- Prima di poter usare un pointer questo deve essere inizializzato, ovvero deve contenere l'indirizzo di un oggetto.
- Per ottenere l'indirizzo di un oggetto si usa un operatore unario **&**.

```
int volume, *vol_ptr;
vol_ptr = &volume;
```

- Il puntatore vol_ptr contiene ora l'indirizzo della variabile volume.
- Per assegnare un valore all'oggetto puntato da vol_ptr occorre utilizzare l'operatore di indirezione " *".

```
*vol_ptr= 15;
```

Esempi

```
/* Dichiarazioni */
int v1, v2, *v_ptr;
/* moltiplica v2 per il valore puntato da v_ptr */
v1 = v2* (*v_ptr);
/* somma v1,v2 e il contenuto di v_ptr */
v1 = v1+v2 +*v_ptr;
/* assegna a v2 il valore che si trova tre interi dopo v_ptr */
v2 = *(v_ptr + 3);
/* incrementa di uno l'oggetto puntato da v_ptr */
*v_ptr += 1;
```


Vantaggi nell'uso dei puntatori

- Utilizzando i puntatori si riduce la quantità dimemoria statica utilizzata dal sistema.
- Nelle chiamate a funzione, ricordate che i parametri vengono passati per valore. Dunque, ogni modifica locale sui parametri passati si perde all'uscita della funzione. Utilizzando i puntatori, è possibile passare l'indirizzo dei valori e dunque è possibile rendere globale ogni modifica sui parametri passati.

Aritmetica dei puntatori

- Sui puntatori sono lecite le seguenti operazioni:
- Assegnamento tra puntatori dello stesso tipo,

```
int *ptr1, *ptr2;
*ptr1 = 1;
ptr2 = ptr1;
```

- Addizione e sottrazione tra puntatori ed interi,

```
int *ptr, arr[10];
ptr = &arr[0];
ptr = ptr + 4 /* punta al quinto elemento dell'array arr[4] */
```


Puntatori e stringhe di caratteri

- Molto spesso vengono usati i puntatori a caratteri in luogo degli array di caratteri (stringhe), questo perché il C non fornisce il tipo predefinito stringa.
- Esiste una differenza sostanziale tra array di caratteri e puntatori a carattere. Ad esempio in:

```
char *ptr_chr = "Salve mondo";
```

- il compilatore non crea una copia della stringa costante "Salve mondo", ma semplicemente crea un puntatore che punta ad una locazione di memoria in cui risiede il primo carattere della stringa costante.
- Ciò significa che posso utilizzare il puntatore in modo che punti a qualcosa di diverso senza modificare il contenuto della stringa costante.

Inizializzazione di array e puntatori a caratteri

- Ecco alcuni esempi di inizializzazione di array di caratteri:

```
char caratteri[4] = { 'a' , 'A' , 'H' , 'k' };
char stringa_2 [] = "MMMM";
```

- Ed ecco alcuni esempi di inizializzazione di puntatori a caratteri:

```
char c
char *carattere;
c = 'a';
*carattere = c;
```


Puntatori come argomenti di funzioni

- Se l'argomento di una funzione è una variabile puntatore il passaggio della variabile avviene per **reference** (per indirizzo) ossia la funzione chiamata sarà in grado di modificare il valore globale della variabile che riceve.
- Passaggio argomenti per valore:

```
int numero;
square(numero);
```

- Passaggio per indirizzo:

```
square(&numero);
```

Esempio

```
void swap(int *x_ptr, int *y_ptr)
{
     int temp;
     temp = *x_ptr;
     *x_ptr = *y_ptr;
     *y_ptr = temp;
}
main()
{
     int a = 3;
     int b = 5;
     swap(&a,&b);
     printf("%d %d\n",a,b);
}
```


Array di puntatori

- Un array di puntatori è un array i cui elementi sono dei puntatori a variabili:

  int *arr_int[10]

- arr_int[0] contiene l'indirizzo della locazione di memoria contenente un valore intero.
- Nel C i puntatori a caratteri vengono usati per rappresentare il tipo stringa che non risulta definito nel linguaggio, e gli array di puntatori per rappresentare stringhe di lunghezza variabile.
- Un insieme di stringhe potrebbe essere rappresentato come un array bidimensionale di caratteri, ma ciò comporta uno spreco di memoria.

Array di puntatori a carattere

- Ad esempio:

  char *term[100];

  Indica che gli elementi di *term* sono dei puntatori a carattere, cioe term[i] è l'indirizzo di un carattere.

  term[7] = "Ciao";

  Indica che il contenuto di term[7] è il puntatore alla stringa "Ciao";

Esempio

```
#include <stdio.h>
main()
{
   char *giorni[7] = { "Lunedi", "Martedi", "Mercoledi", "Giovedi",
   "Venerdi", "Sabato", "Domenica" };
   int i;
   for( i=0; i< 7; i++) {
   printf("\n %d ",*giorni[i]);
   printf("%s",giorni[i]); }
}
```

76 Lunedi || 77 Martedi || .....

Esempio grafico

Preprocessore C

- Il **preprocessore C** è una estensione al linguaggio che fornisce le seguenti possibilità:
  - **definizione delle costanti**
  - **definizione di macro sostituzioni**
  - **inclusione di file**
  - **compilazione condizionale**
- I comandi del preprocessore iniziano con **#** nella prima colonna del file sorgente e non richiedono il "**;**" alla fine della linea.
- Un compilatore C esegue la compilazione di un programma in due passi successivi. Nel primo passo ogni occorrenza testuale definita attraverso la direttiva # viene sostituita con il corrispondente testo da inserire (file, costanti, macro)

Preprocessore C : Costanti

- Attraverso la direttiva #define del preprocessore è possibile definire delle costanti:
- Sintassi: **#define** *nome* *testo da sostituire* :

```
#define   MAXLEN   100
#define   YES      1
#define   NO       0
#define   ERROR    "File non trovato\n"
```

- E' uso comune usare lettere maiuscole per le costanti di #define
- Perché usare queste costanti?
  - favoriscono la leggibilità del programma
  - consentono un facile riuso del codice

Preprocessore C : Macro

- L'suo della direttiva **#define** consente anche di definire delle macro.
- Una macro è una porzione di codice molto breve che è possibile rappresentare attraverso un nome; il preprocessore provvederà ad espandere il corrispondente codice in linea.
- Una macro può accettare degli argomenti, nel senso che il testo da sostituitre dipenderà dai parametri utilizzati all'atto della chiamata. Il preprocessore espanderà il corrispondente codice in linea avendo cura di rimpiazzare ogni occorrenza del parametro formale con il corrispondente argomento reale.

Macro : Esempi

```
#define square(x) ((x)*(x))
#define MIN(a,b) ( (a<b)? (a) : (b) )
#define ASSERT(expr) if(!(expr)) printf("error")
```

Nel file sorgente le linee :

```
square(2);
MIN(2,3);
ASSERT (a > b);
```

saranno sostituite in compilazione con

```
((2) * (2));
( (2 < 3) ? (2) : (3) );
if (!(a<b)) printf("error");
```


Cosa non è una Macro

- Anche se ciò può trarre in inganno, le macro **NON** sono funzioni.
- Per esempio sugli argomenti delle macro non esiste controllo sui tipi.
- Inoltre, una chiamata del tipo : MAX (i++, j++) verrà sostituita con:

  ```
  ( (i++ > j++) ? (i++) : (j++);
  ```

- E' importante stare attenti all'uso delle parentesi, ad esempio in:

  ```
  #define square(x) x*x
  ```

- Una chiamata del tipo: `x = square(3+1);` , genera:

  ```
  x = 3 + 1 * 3 + 1; --> x = 7 ??????
  ```

Preprocessore C : Compilazione condizionale

- Le direttive : **#if #ifdef #ifndef #elif #else #endif** consentono di associare la compilazione di alcune parti di codice alla valutazione di alcune costanti in fase di compilazione.

```
#if < espressione_costante>
< statement_1>
#else
<statement_2>
#endif
```

- Se l'espressione costante specificata, valutata in compilazione ritorna TRUE allora verranno compilati gli statement_1 altrimenti verranno compilati gli statement_2

Classi di memorizzazione

- Nel linguaggio C esistono 4 diversi specificatori di **storage class** rappresentati dalle label:
  - **auto**
  - **extern**
  - **static**
  - **register**
- A ciascuno di essi corrisponde un diverso comportamento da parte del compilatore ed un diverso uso.

Variabili automatiche

- Una variabile definita all'interno di un blocco di funzione (variabile locale), od anche preceduta dal suffisso auto viene considerata automatica.
- Per default ogni variabile, è considerata automatica per cui il prefisso **auto** è opzionale.
- Una variabile automatica è referenziabile solo all'interno della funzione nella quale è stata dichiarata.
- Una variabile dichiarata **auto** senza tipo viene considerata di tipo **int**.
- Il valore di una variabile **auto** o locale è perso quando si esce dalla funzione che l'ha dichiarata.

Variabili e funzioni esterne

- Una variabile viene definita "esterna" quando la sua definizione è posta al di fuori del modulo che la utilizza.
- Una variabile è esterna quando:
  - è definita **globalmente**, ovvero quando la sua definizione è visibile a più moduli differenti, ma sempre anteposta ai moduli che la utilizzano.
  - è dichiarata esplicitamente attraverso la label ***extern***, in tal caso la sua definizione può anche essere successiva al modulo che la utilizza.
- C'è differenza tra dichiarazione e definizione di una variabile: con la dichiarazione si rende noto al compilatore il solo tipo della variabile, mentre con la definizione si crea spazio in memoria riservata ad essa.

A cosa servono le variabili esterne

- Le variabili esterne sono spesso utilizzate come modo alternativo di scambio di dati tra moduli differenti.
- Al contrario delle variabili automatiche le variabili esterne conservano il loro valore anche tra due chiamate successive di funzione.
- Se due funzioni devono condividere dei dati, e non si invocano a vicenda, è possibile utilizzare delle variabili esterne per consentire loro di condividere dei dati anziché usare la lista degli argomenti.

"Scope" di una variabile esterna

- Il range di visibilità di una variabile, ovvero l'insieme dei moduli in cui risulta essere visibile la variabile stessa, viene definito come scope della variabile.
- Lo scope di una variabile esterna globale va dal punto della sua dichiarazione fino alla fine del file sorgente.
- Lo scope di una variabile esterna "*extern*" comprende sia il file sorgente in cui è presenta la sua definizione che tutti i file diversi da quello sorgente, in cui è presente una sua dichiarazione attraverso la label ***extern***.

Variabili e funzioni " static "

- Questo tipo di variabili hanno due tipi di utilizzo interno ed esterno.
- Nel caso interno, una variabile static mantiene il suo valore al rientro di un blocco. Nel seguente codice, con la variabile static cnt, si tiene traccia di quante volte viene chiamata una funzione.

```
f()
{static int cnt=0
++cnt … }
```

- Lo **scope** di una variabile **static interna** (dichiarata all'interno di una funzione) è uguale allo scope di una variabile automatica.
- Lo scope di una variabile **static esterna** è tutto il resto del file dal momento in cui viene dichiarata (visibilità ridotta rispetto).

Variabili "register"

- Lo specificatore **register** comunica al compilatore che, se può, deve rappresentare la variabile in oggetto con un registro di CPU.
- Sintassi: register int count;
- Per quelle variabili che vengono referenziate spesso è utile la dichiarazione **register** in quanto l'accesso al dato è molto più veloce.

La Ricorsione

- Il concetto di ricorsione nasce dalla possibilità di eseguire un compito applicando lo stesso algoritmo ad un dominio ridotto rispetto a quello originale fondendo i risultati.
- Le funzioni C possono essere usate ricorsivamente, cioè una funzione può chiamare se stessa sia direttamente che indirettamente.
- Nella ricorsione è importante la condizione di uscita.
- Il problema deve poter essere suddiviso in sottoproblemi piu' piccoli fino ad arrivare ad un sottoproblema banale di cui si conosce immediatamente la soluzione.

Esempio di ricorsione

- Calcolo del fattoriale di un numero:
- n! = n*(n-1)*(n-2)* .. .* (n - (n-1))

```
int fact(int num)
{
int product=1;
for(; num>1; --num)
product*=num;
return product;
}
```

```
int fact(int num)
{
if ( num <= 1)
return 1;
else
return (num* fact(num -1));
}
```

Esempio di ricorsione

- Calcolo del numero di Fibonacci

  f (n) = f (n-1) + f (n-2); f(0) = 0 ; f(1) =1;

```
#include <stdio.h>
int num;
main() {
printf("\n numero? ");
scanf("%d",&num);
printf("\n risultato %d ",fib(num));
}
```

```
int fib(int num) {
int tmp=0, ris=1, prec=0;
switch (num) {
case 0: return 0;break;
case 1: return 1;break;
default: {
for (; num>1;--num) {
tmp=ris;
ris+=prec;
prec=tmp; }
return ris; } }}
```

```
int fib(int num)
{
switch (num) {
case 0: return 0;break;
case 1: return 1;break;
default:
return (fib(num-1) + fib(num-2));
        }
}
```


Esercizio: Algoritmo Quicksort

- Il quickort è un algoritmo di ordinamento in cui un vettore può essere ordinato suddividendolo in due sottoinsiemi da ordinare indipendentemente.
- In particolare, dato un vettore si considera un elemento discriminate tale da dividere il vettore in due sottoinsiemi. Successivamente si scandisce il vettore da sinistra fino ad incontrare un elemento maggiore dell'elemento discriminante e da destra fino ad incontrare un elemento maggiore di quello discriminante. I due elementi vengono invertiti e si continua nella scansione fino ad avere sicuramente a sinistra dell'elemento discriminante elementi minori e a destra quelli maggiori. L'istruzione conclusiva consiste nello spostare l'elemento discriminante con l'elemento più a sinistra della porzione destra del vettore suddiviso. Il passo conclusivo consiste nel applicare questa procedura ricorsivamente al sottoinsieme sinistro ed a quello destro del vettore.