Lettura controllata: intero

- Si vuole costruire un modulo che consenta la lettura **controllata** di un valore intero
- Tale modulo deve:
  - Stampare a video un messaggio che preceda la lettura
  - Effettuare la lettura
  - Controllare che la lettura sia andata a buon fine (è stato effettivamente inserito un intero)
    - Se è tutto ok, comunicare all'esterno il valore letto e il successo dell'operazione
    - Se qualcosa è andato storto, stampare un messaggio d'errore e chiedere se l'utente vuole annullare l'operazione
      - Se l'utente desidera annullare l'operazione, terminare l'esecuzione del modulo restituendo insuccesso, altrimenti rileggere il valore (ricominciare da capo)

Lettura controllata: intero

- Parametri di ingresso:
  - messaggio da stampare prima della lettura
  - messaggio da stampare in caso d'errore
- Valori in uscita:
  - valore letto
  - indicazione di successo/insuccesso
- Non potendo avere due valori in uscita in una funzione, ***valore di ritorno sarà l'indicazione di successo*** mentre il ***valore letto sarà inserito in un parametro passato per indirizzo***

Lettura controllata

- Interfaccia

Perché senza dimensione dell'array funziona tutto ugualmente?

```
BOOLEAN readInt(char text[], char errorText[], int *n);
```

Notare che degli array di caratteri (stringhe) non si indicano le dimensioni; quindi in C la dichiarazione sopra è del tutto equivalente a:

```
BOOLEAN readInt(char *text, char *errorText, int *n);
```

Ciò non significa che text e errorText siano parametri in/out, ma solo che gli array sono ***sempre passati per indirizzo...***

Lettura controllata: intero

```
BOOLEAN readInt(char text[], char errorText[], int *n)
{
    BOOLEAN success = FALSE;
    do
    {
        printf(text);
        success = scanf("%d", n);
        while (getchar() != '\n');
        if (!success)
        {
            char op;
            printf(errorText);
            printf("\n");
            printf("Annullare l'operazione? (s/n)");
            scanf("%c", &op); while (getchar() != '\n');
            if (op == 's' || op == 'S')
                return FALSE;
        }
    }
    while (!success);
    return TRUE;
}
```

Svuota il buffer di lettura

Lettura controllata: array

- Si supponga di voler leggere un ***array di interi*** di cui poi calcolare la ***media*** e lo ***scarto quadratico medio***

- Scomposizione del problema
  - Lettura di un intero (vedi esempio precedente, ma...)
  - Lettura di un insieme di interi (come terminare?)
  - Calcolo della media
  - Calcolo dello scarto quadratico medio

Funzioni di utilità ulteriori: sscanf, sprintf

Esistono funzioni di libreria ulteriori che possono essere utili per gestione/processamento di stringhe, e in particolare per ***conversione da/verso altri tipi di dato***

- **`int sscanf(char *s, char *format, …)`**

  permette di processare il contenuto di s e trasformarne parti nella rappresentazione interna dei tipi di dato specificati

```
char s[100] = “eta = 12”, s1[100]; int code, x;
code = sscanf(s, "%s = %d", s1, &x);
```

- **`int sprintf(char *s, char *format, …)`**

  permette di trasformare i dati specificati in rappresentazione esterna e di scriverli nella stringa s

```
code = sprintf(s, “i1 vale %d; i2 vale %f”, i1, i2)
```

Lettura controllata: array

- Richiedere quanti interi si desiderano inserire (min 1, max dimensione array)
- Eseguire un ciclo finché:
  - Il ciclo non termina naturalmente → lettura completata
  - Terminazione per volontà dell'utente (annullamento operazione)

- È necessaria una funzione di lettura di interi – **`readConstrainedInt()`** – in cui sia possibile specificare anche gli estremi entro cui l'intero letto risulta valido
- In realtà è sufficiente modificare leggermente la **`readInt()`**...

Lettura controllata: intero

```
BOOLEAN readConstrainedInt(char text[], char errorText[], int
   min, int max, int *n)
{
    BOOLEAN success = FALSE;
    do
    {
        printf(text);
        success = scanf("%d", n);
        while (getchar() != '\n');
        if (!success || *n<min || *n>max)
        {
            char op;
            printf(errorText);
            printf("\n");
            printf("Annullare l'operazione? (s/n)");
            scanf("%c\c\n", &op); while (getchar() != '\n');
            if (op=='s' || op=='S')
                return FALSE; //Fail
        }
    }
    while (!success);
    return TRUE; //Succeed
}
```

Errore se è fallita la scanf() oppure se il valore non rientra nell'intervallo

Riutilizzo e fattorizzazione

- **`readInt()`** e **`readConstrainedInt()`** sono molto simili → *come (e perché può essere utile) evitare replicazione di codice?*

- Come favorire il riutilizzo?
  - **`readInt()`** è un caso particolare di **`readConstrainedInt()`** dove gli estremi dell'intervallo sono il minimo e il massimo valore assumibile da una variabile di tipo **`int`**
  - Valori minimo e massimo (insieme ad altri valori limite) sono contenuti nell'*header file* **`limits.h`**
    - Minimo: **`INT_MIN`**
    - Massimo: **`INT_MAX`**

Lettura controllata: intero

La nuova versione della readInt ☺

```
...
#include <limits.h>
...

BOOLEAN readInt(char text[], char errorText[], int *n)
{
    return readConstrainedInt(text,
            errorText, INT_MIN, INT_MAX, n);
}
```

Lettura controllata: array

- Parametri di input:
  - Array da leggere
  - Dimensione effettiva dell'array
- Parametri di output:
  - Numero di valori effettivamente letti ed inseriti nell'array (0 se non è stato letto nulla...)
- Interfaccia:

```
int readIntArray(int intArray[], int dim);
```


Lettura controllata: array

```
int readIntArray(int intArray[], int dim)
{
    int i, count, value;
    char text[50], errorText[80];
    sprintf(text, "Elementi da inserire (max %d): ", dim);
    sprintf(errorText, "Inserire un valore compreso fra 1 e %d", dim);
    if (!readConstrainedInt(text, errorText, 1, dim, &count))
        return 0;

    for (i = 0; i < count; i++)
    {
        sprintf(text, "Elemento %d: ", i);
        if (!readInt(text, "Inserire un intero", &value))
            return i;

        intArray[i] = value;
    }
    return count;
}
```

Media e Deviazione Standard

- Si vuole produrre un modulo che, dato un array di interi, sia in grado di effettuare il calcolo della media e della deviazione standard
- Come dovrebbe essere noto, date N misure della stessa grandezza x
  - media è definita come: $$\overline{x} = \frac{\sum_{i=1}^{N} x_i}{N}$$
  - deviazione standard: $$\sigma = \sqrt{\frac{\sum_{i=1}^{N} (x_i - \overline{x})^2}{N}}$$

Media e Deviazione Standard

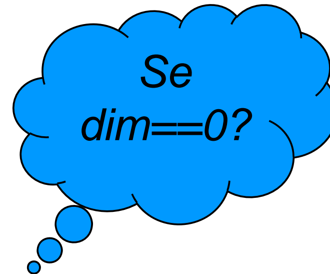
- In entrambi i casi:
  - Parametro di ingresso 1 = array di interi
  - Parametro di ingresso 2 = dimensione array
  - Valore di ritorno = risultato
- Interfaccia

```
double media(int values[], int dim);
double devStandard(int values[], int dim);
```

Per parametrizzare le funzioni in modo opportuno, è necessario passare anche la dimensione dell’array che può variare di chiamata in chiamata

Media

```
double media(int values[], int dim)
{
    int i, sum = 0;
    for (i = 0; i < dim; i++)
        sum += values[i];
    return (double)sum / dim;
}
```



Deviazione Standard

```
double devStandard(int values[], int dim)
{
    int i;
    double meanValue, temp, sum=0;
    meanValue = media(values, dim);
    for (i=0; i<dim; i++)
    {
        temp = values[i]-meanValue;
        sum = sum + temp*temp;
    }
    return sqrt(sum/dim);
}
```

Lettura controllata: array

- Interessante estensione: lettura di un array di interi con vincoli di *upper* e *lower bound* sui valori inseriti
- Un utilizzo potrebbe essere: inserire i propri voti (18 <= voto <= 30) per poi calcolare media e deviazione standard

Happy Coding!

Ricerca in array

- Se l'array non è ordinato → ricerca lineare
- Se l'array è ordinato → ricerca binaria

- Nota: conviene ordinare un array per usare la ricerca binaria?
  - ***Dipende*** → *si vedrà poi in quali condizioni e perché...*

Ricerca binaria

- Definizione
  - Sia **`dim`** la dimensione dell'array
  - Se l'elemento mediano (posizione **`med`**) dell'array è l'elemento da cercare → ***elemento trovato***
  - Se l'elemento mediano dell'array è maggiore dell'elemento da cercare → ***cercare nella prima metà dell'array*** (dalla posizione "0" alla posizione **`med-1`**)
  - Se l'elemento mediano dell'array è minore dell'elemento da cercare → ***cercare nella seconda metà dell'array*** (dalla posizione **`med+1`** alla posizione "finale")
- La definizione è evidentemente ricorsiva...

Ricerca binaria

- Parametri in ingresso:
  - Array in cui cercare
  - Dimensione dell'array
  - Elemento da cercare
- Valori in uscita:
  - Posizione dell'elemento nell'array
  - Successo della ricerca

  →I due valori sono sintetizzabili in uno unico?
  - →La posizione in un array è sempre maggiore o uguale a zero
  - →***Un numero negativo può essere considerato un insuccesso nella ricerca...***

Ricerca binaria

```
#include <limits.h>
int binarySearch(int intArray[], int dim, int toSearch)
{
    int midPos = dim/2;
    if (intArray[midPos] == toSearch)
        return midPos;
    if (midPos == 0)
        return INT_MIN;
    if (intArray[midPos] > toSearch)
    {
        return binarySearch(intArray, midPos, toSearch);
    }
    else
    {
        int startPos = midPos+1;
        return startPos + binarySearch(&intArray[startPos],
                dim-startPos, toSearch);
    }
}
```


Ricerca binaria: note

- `&intArray[startPos]`

  → Indirizzo dell'elemento di posizione **`startPos`**

  → Sotto-array parzialmente sovrapposto all'array di partenza (**`intArray`**) i cui elementi sono quelli compresi fra **`startPos`** (compreso) e la fine dell'array

- `startPos + binarySearch(&intArray[startPos],`
  `dim-startPos, toSearch);`

  → La ricerca riparte dal sotto-array che inizia da **`startPos`**

  → occorre sommare la posizione di partenza al risultato della sottoricerca

  → la dimensione del sotto-array è **`dim-startPos`**

Ricerca binaria – estensione

- E se cambia il tipo di dato? Come permettere il riutilizzo di codice *(solo se necessario…)*?

Il tipo di dato DEVE essere dotato di una opportuna operazione di confronto:

- `int compare(TYPE d1, TYPE d2);`
- Il risultato è:
  - Positivo per `d1` maggiore `d2`
  - Nullo per `d1` uguale `d2`
  - Negativo per `d1` minore `d2`

Domande a cui sapere assolutamente rispondere in sede d'esame ☺

- Perché nella *signature* di una funzione che prevede il passaggio di array è possibile omettere la dimensione dell'array stesso?
- ***Quali sono le (piccole) differenze fra array e puntatori in C?***
- È possibile cambiare *upper* e *lower bound* di un array?

Perché nella *signature* di un metodo…

Nella ***definizione di un array*** la ***dimensione*** serve per ***allocazione della memoria***. A runtime, all'atto della chiamata di funzione, non viene effettuato alcun *bound checking* (attenzione, non c'è quindi alcun controllo!) → alla macchina runtime servono solo:

- indirizzo del primo elemento dell'array
- dimensione del tipo di dato contenuto dall'array

Definizione:

```
int myArray[53];
```

Passaggio:

```
void myProcedure(int anArray[])
{
        anArray[3] = 10;
}
```


Array e puntatori?

- La variabile che denota un array contiene ***l'indirizzo del primo elemento dell'array***…
- …tale indirizzo può essere ugualmente contenuto in un puntatore!

- Però la variabile che denota l'array è assimilabile a un ***puntatore costante*** (mantenuto in modo simile a quanto fatto in altri linguaggi per variabili con valori non modificabili), mentre un normale puntatore può ovviamente cambiare di valore:

```
int *p, a[5];
p = a; //Ok!
a = p; //Errore!
```

Array e puntatori?

Per il resto, che piaccia o meno, le notazioni di array e puntatori in C sono del tutto simili e possono essere usate in modo mescolato

```
int *p, a[5];
p = a;
p[1] = 4;
*(a+2) = 3;
p = &a[2];
```

Che cosa cambia a livello di allocazione?

…*upper* e *lower bound*?

Da dimenticare...

In C

- Il *lower bound* di un array è sempre 0, l'*upper bound* è la dimensione dell'array meno 1
- *Upper* e *lower bound* degli array non vengono verificati:

```
int i, a[4];
i = a[-2];
```

Non genera errore di compilazione ma "solo" eventuale errore a runtime

- Usando le proprietà di array e puntatori è possibile ottenere un "array" dove *upper* e *lower bound* sono diversi dal solito

```
int i, *p, a[5];
p = &a[2];
for (i = -2; i <= 2; i++)
    p[i] = i;
```

Lower bound = -2; Upper bound = 2

Si usa **p** *come se fosse un normale array… un po' speciale!*

…*upper* e *lower bound*?

Da dimenticare...

- Si supponga di voler fare in modo che il *lower bound* di un array sia 1 → potrebbe aver senso in quanto il primo elemento sarebbe l'elemento di indice 1…

```
int a[5], *p;
p = &a[-1]; // p = a - 1;
p[1]…
```

è il primo elemento; p[0] è l'elemento -1esimo: occhio!

…*upper* e *lower bound*?

Da RICORDARE!!!

- Attenzione: cambiare le convenzioni è sempre pericoloso
- La cosa deve essere altamente giustificata, per esempio per far aderire meglio il programma al sistema che si sta modellando… ma anche in quel caso…
- Fra le altre cose, cambiare upper e lower bound rende il programma meno leggibile
- Un altro discorso è voler estrarre un sotto-array:

```
int a[5], *sa;
sa = a + 2;
```

Sa = sotto-array che comincia due elementi più avanti → vedi ricerca binaria ricorsiva

C'era una volta un hacker…

Calcolo della lunghezza di una stringa

- Versione 0

```
int lunghezza(char s[])
{
   int lung;
   for (lung=0; s[lung]!='\0'; lung++);
   return lung;
}
```

- Versione 1

```
int lunghezza(char  *s)
{
   int lung=0;
   for (lung=0; s[lung]!='\0'; lung++);
   return lung;
}
```


C'era una volta un hacker…

- Versione 2

```
int lunghezza(char *s)
{
   char *s0 = s;
   while (*s) s++;
   return s-s0;
}
```

- Versione 3

```
int lunghezza(char *s)
{
   char *s0 = s;
   while (*s++);
   return s-s0-1;
}
```

1. Viene dereferenziato il puntatore e usato nel test

2. Viene incrementato il puntatore (e non il valore puntato)

*→ Gli operatori unari * e ++ sono equiprioritari e associativi da destra a sinistra!*