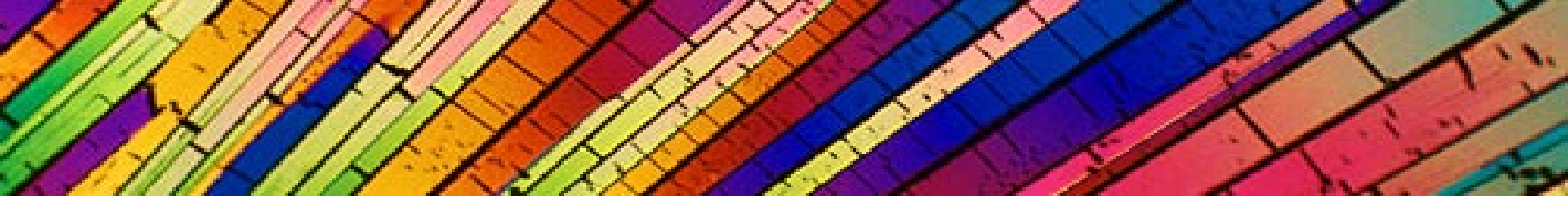


Complementi - 1

Ver 3

Inizializzazione di max e min

- Quando si deve cercare il massimo o il minimo di una sequenza di valori, è necessario inizializzare il valore delle variabili che conterranno questi valori, ad esempio per il max l'idea di base è questa:

```
max = ???;
for (i=0; i<N; i++)
{
    scanf("%d", &v);
    if (v>max)
        max = v;
}
```

Inizializzazione di max e min

- Soluzione 1 - Si imposta il `max` al primo valore
  - Questo metodo funziona sempre ed è indipendente dal tipo della variabile in quanto l'inizializzatore è uno dei valori effettivamente introdotti
  - È solitamente la soluzione preferibile nel caso di vettori e matrici già in memoria:
    ```
    max = mx[0][0];
    ```
  - Il primo valore può essere letto prima di entrare nel ciclo o dentro il ciclo stesso

Inizializzazione di max e min

- Se `max` viene inizializzato prima di entrare nel ciclo, il ciclo non viene ritardato

```
scanf("%d", &v);
max = v;
for (i=1; i<N; i++)
{
    scanf("%d", &v);
    if (v>max)
        max = v;
}
```

Notare che nel ciclo si legge un valore in meno in quanto il primo è già stato letto

Inizializzazione di max e min

- Se `max` viene inizializzato dentro il ciclo è necessario inserirvi un controllo che viene eseguito inutilmente ogni volta (tranne la prima) e quindi introduce una qualche inefficienza

```
for (i=0; i<N; i++)
{
    scanf("%d", &v);
    if (i==0 || v>max)
        max=v;
}
```

Per piccoli programmi va comunque bene

Inizializzazione di max e min

- Soluzione 2 - Si imposta il `max` a un valore costante deciso dal programmatore in base al tipo di dato e a che cosa rappresentano i valori
  - Se ad esempio i valori sono aree o distanze, dato che queste non possono essere negative si può impostare `max` a 0 o a un valore negativo: qualsiasi valore introdotto sarà maggiore (o uguale)
  - Ma se i valori in input possono invece essere qualsiasi (es. temperature), bisogna considerare che potrebbero essere introdotti anche solo valori negativi e quindi in questo caso se `max`=0, `max` non verrebbe mai aggiornato (sono tutti inferiori) ossia resterebbe pari a 0, valore che non è neppure uno di quelli introdotti dall'utente → errore

Inizializzazione di max e min

- Soluzione 3 - Si imposta il `max` al valore più piccolo possibile per quel tipo di dato
  - Si usano i valori costanti forniti dal compilatore attraverso i file di `#include` (vedere le slide sui tipi di dati)
  - Nel caso di valori interi si deve includere `<limits.h>`, ad esempio per gli `int` il valore più piccolo è disponibile la costante `INT_MIN` (per gli altri interi è simile) e questo è un valore negativo

Inizializzazione di max e min

- Nel caso di valori floating point si deve includere `<float.h>`, ma l'intero più piccolo è un valore positivo, ad es. per i `float` la costante `FLT_MIN` potrebbe essere 1E-37 (molto piccola ma positiva), quindi il valore più negativo con cui inizializzare `max` è in realtà `-FLT_MAX` (–1E+37)
- Questo metodo è sicuramente efficace, ma se si deve adattare il programma perché usi un altro tipo di dato, anche la constante va cambiata, cosa che non capita con i metodi precedenti.

Inizializzazione di max e min

- Nel caso si debba inizializzare il min, il discorso è analogo a quanto visto per il max
- Quando si devono cercare sia il max sia il min:
  - Con la Soluzione 1, impostando il max e il min al primo valore, si può usare il costrutto:

```
if (v>max)
    max = v;
else if (v<min)
    min = v;
```

  in quanto qualsiasi valore letto non può essere contemporaneamente maggiore del max e minore del min. Quell'`else` permette di minimizzare il numero di esecuzioni del secondo `if` → efficiente

Inizializzazione di max e min

- Con le Soluzioni 2 e 3, impostando `max` e `min` a valori costanti differenti, è necessario il costrutto:

```
if (v>max)
    max = v;
if (v<min)
    min = v;
```

L'esecuzione è qui meno efficiente della soluzione precedente perché qui l'`else` non può essere utilizzato. Infatti, qualora fosse inserito un solo valore o i valori immessi fossero tutti crescenti, ogni iterazione aggiornerebbe sempre solo `max` e l'`else` non sarebbe mai eseguito, quindi il valore di `min` resterebbe il valore dell'inizializzazione, quindi errato

L’indentazione del codice

- Per indentazione si intende il precedere le righe di codice con un certo numero di spazi
- Ha lo scopo di evidenziare i blocchi di codice, ossia dove iniziano e finiscono
- Ignorata dal compilatore, serve al programmatore per cogliere visivamente la struttura del programma, quindi è utile indentare il codice *mentre lo si sviluppa*
- Il numero di spazi è sempre multiplo di un valore scelto come base (4, 8)
- Gli editor/IDE permettono di impostare il tasto Tab perché introduca spazi di indentazione

L'indentazione del codice

- L'indentazione rende evidente una *dipendenza*, un *controllo:* l'istruzione non indentata che precede il blocco **controlla** il blocco indentato
- Esempio:

```
for (i=0; i<10; i++)
{
    scanf("%d", &a);
    printf("%d\n", a*2);
}
```

Istruzione non indentata

Blocco indentato

Il blocco con le istruzioni `scanf` e `printf` è controllato dall'istruzione `for` precedente, la `printf` non è controllata dalla `scanf` e per questo è allo stesso livello (non è indentata)

L'indentazione del codice

- Esempio:

```
1  for (i=1; i<100; i++)
2  {
3      printf("%d", i);
4      if (i % 7 == 0)
5      {
6          printf(" divisibile per 7");
7          cont++;
8      }
9      printf("\n");
10 }
```

Le istruzioni alle righe 3, 4 e 9 appartengono al blocco del `for` e sono allo stesso livello, quelle alle righe 6 e 7 appartengono al blocco dell'`if` che quindi è ulteriormente indentato

L'indentazione del codice

- L'indentazione è ancora più utile in assenza delle parentesi graffe (quando sono opzionali)
- Esempio:

```
for (i=0; i<10; i++)
    for (j=0; j<20; j++)
        printf("%d,%d", i, j);
```

L'istruzione `printf` è controllata dall'istruzione `for` precedente che a sua volta è controllata dall'istruzione `for` più esterna

L'indentazione del codice

- Per il posizionamento delle parentesi graffe che racchiudono il blocco si suggerisce di utilizzare la modalità Allman:
  - la graffa di inizio blocco è collocata sotto il primo carattere della parola chiave che controlla il blocco, in una riga a sé stante
  - tutte le istruzioni del blocco sono indentate
  - la graffa di fine blocco è allineata sotto quella di inizio blocco, in una riga a sé
- La soluzione degli esercizi rispetta sempre questa formattazione, gli esempi di queste slide quasi sempre salvo problemi di spazio

Istruzione nulla

- In alcuni casi un'istruzione di controllo non ha bisogno di istruzioni nel blocco controllato perché l'elaborazione avviene altrove
- Esempio
  Per calcolare la lunghezza di una stringa, nel ciclo FOR seguente ciò che serve è solo incrementare `i` fino a raggiungere il carattere terminatore `'\0'` e questo viene fatto nell'istruzione `for` stessa

```
for (i=0; v[i]!='\0'; i++)
   ;
lunghezza = i;
```

Istruzione nulla

- Quando è necessario fornire un corpo vuoto a un'istruzione si può usare l'***istruzione nulla***, ossia il solo carattere ';'
- Non produce alcuna azione e per chiarezza è bene sia *collocata in una riga a sé stante indentata opportunamente*
- In alternativa (meno preferibile) si possono usare le parentesi graffe `{}` di definizione del corpo del FOR lasciandolo vuoto oppure utilizzare l'istruzione `continue`

Expression statements

- Un *expression statement* è un'espressione che produce un risultato che però non viene né memorizzato in una variabile né utilizzato
- In alcuni casi quello che interessa è l'effetto collaterale prodotto
- Esempi (alcuni utili, altri inutili)

`sin(x);` ← *non serve a nulla*

`x+y;` ← *non serve a nulla*

`i++;` ← *incrementa* `i` *come effetto collaterale*

`printf("Ciao\n");` ← *stampa Ciao ma il risultato (numero di caratteri stampati) non viene utilizzato*

Expression statements

- Se si vuole esplicitamente indicare che il valore restituito da una funzione deve essere scartato si fa un cast a `void`:

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

- Raramente è necessario specificarlo

Magic numbers

- Le costanti numeriche presenti in un programma hanno spesso un significato preciso e spesso sono utilizzate in più punti
- Per chiarezza si deve evitare di utilizzare queste costanti e dare ad esse all'inizio del programma un nome significativo del contenuto, ad es. con `#define` o mediante variabili con il qualificatore `const` (così da non poterle più modificare)
- Questo permette anche di parametrizzare il programma: se serve modificare quel valore è sufficiente farlo nella definizione

Magic numbers

- Esempio errato

```
int temperatura[180];
for (i=0; i<180; i++)
   scanf("%d", &temperatura[i]);
```

Qui è facile capire che quel "365" è la dimensione di `temperatura`*, ma in un punto del programma distante dalla definizione potrebbe essere poco evidente, inoltre se si vuole modificare a 1 anno?*

- Esempio corretto

```
#define NUMGIORNI 180
for (i=0; i<NUMGIORNI; i++)
   somma += temperatura[i];
```

Magic numbers

- Esempio errato

```
int v[80];
for (i=0; i<80; i++)
   scanf("%d", &v[i]);
for (i=79; i>0; i++)
   printf("%d ", &v[i]);
```

- Esempio corretto (inoltre se serve cambiare la dim. del vettore basta farlo nella `#define`):

```
#define MAXVETT 80
int v[MAXVETT];
for (i=0; i<MAXVETT; i++)
   scanf("%d", &v[i]);
for (i=MAXVETT-1; i>0; i++)
   printf("%d ", &v[i]);
```

Uscita dal main

- Il programma termina quando nel `main` viene eseguita l'istruzione `return`:
  `return` *status*`;`
- In un programma possono esserci più istruzioni `return` (anche se non è più completamente strutturato è accettabile)
- Nel `main` l'istruzione `return` termina il programma e passa il valore *status* al Sistema Oper. per informarlo sull'esito dell'esecuzione:
  - il valore 0 indica che il programma non ha avuto problemi: terminazione con successo
  - valori diversi da 0 indicano che il programma ha avuto problemi (es. non ha trovato un file): terminazione con errore

Uscita dal main

- Includendo `<stdlib.h>`, per *status* si possono usare (ed è consigliabile) le costanti:
  - `EXIT_SUCCESS` (al posto di 0) per indicare una terminazione con successo
  - `EXIT_FAILURE` (valore ≠ 0) per indicare una terminazione con errore, in genere 1
- Sempre includendo `<stdlib.h>`, per terminare un programma si può utilizzare la *funzione* (e quindi richiede le parentesi):
  `exit(`*status*`);`
  `return` è invece un'istruzione del linguaggio
- Nel `main` la `exit` è equivalente alla `return` (non è così nelle funzioni)

Effetti collaterali

- Un'espressione può produrre un risultato e/o dare ***effetti collaterali*** (*side-effect*)
- Esempio

  `x = 3 * i++;`

  `3*i` è un *calcolo* che produce un valore, mentre `i++` ha l'*effetto collaterale* di incrementare `i` di 1
- Quando tutti i side-effect di un'espressione sono portati a termine, si dice che si è raggiunto un ***sequence point***
- Prima del raggiungimento del sequence point, il valore delle variabili soggette a side-effect è *indefinito* e quindi non devono essere usate

Effetti collaterali

- In una stessa espressione (si ricordi che *espr1=espr2* è anch'essa un'espressione) NON deve comparire più di una volta una variabile soggetta a side-effect (salvo il raggiungimento di un sequence point intermedio), il compilatore (al più) segnala un Warning
- Un sequence point è raggiunto in particolare:
  - prima di passare all'istruzione successiva (ma quando di preciso non è specificato)
  - dopo l'esecuzione di ciascuna delle espressioni separate dagli operatori `&&`, `||`, `?:` e `,`
  - Al termine della valutazione delle espressioni di `while`, `for`, `do`, `if`, `switch` e `return`
  - DOPO che *tutti* gli argomenti di una funzione sono stati valutati, ossia appena prima della chiamata

Ordine di valutaz. e side-effect

- Le regole di precedenza tra gli operatori e le parentesi impongono un ordinamento nella valutazione delle espressioni, ma bisogna considerare anche i side-effect
- Ad esempio in un'espressione come

  ```
  x = f() + g() * h();
  ```

  viene calcolata prima la moltiplicazione e poi la somma, ma lo Standard indica che:
  - non si può supporre che `f()` sia calcolata dopo le altre solo perché il suo risultato viene usato dopo la moltiplicazione
  - non si può supporre che `g()` sia calcolata prima di `h()` perché la moltiplicazione è associativa a destra

Ordine di valutaz. e side-effect

- In questo caso se si vuole essere sicuri che f sia calcolata dopo aver valutato g e h si deve utilizzare una variabile per mantenere il risultato intermedio:

```
y = g() * h();
x = f() + y;
```
← *istruzione successiva*

- Esempi (errati)
  - `y = ++x * (x-1) * z;`
    L'operatore prefisso `++` incrementa `x` prima di fare la moltiplicazione, ma non è detto che `x-1` non sia già stato calcolato: l'ordine delle *operazioni* è da sinistra a destra, ma quando i singoli *operandi* sono valutati non è noto; l'unica certezza è che prima di eseguire l'istruzione successiva la `x` sarà stata incrementata

Ordine di valutaz. e side-effect

- Esempi (errati)
  - `y = sin(1/++x) + cos(1/x);`
    L'operatore prefisso `++` incrementa `x` prima che la funzione `sin` sia eseguita, ma non è dato sapere se venga calcolata prima `sin` o prima `cos`, né quindi se la `x` della `cos` è stata incrementata o no; poi la somma dei risultati avviene normalmente da sinistra a destra (associatività degli operatori matematici)
  - `v[i] = i++;`
    non si sa se il valore dell'indice `i` in `v[i]` sia quello primo o dopo l'incremento, l'assegnazione non inserisce un sequence point

Ordine di valutaz. e side-effect

- Esempi (errati)
  - `printf("%d %f\n", ++x, tan(1.0/x));`
    La virgola che separa `++x` da `tan(1.0/x)` non è l'*operatore virgola* (descritto più avanti), ma un semplice *separatore* e quindi non implica alcun ordine di valutazione.
    Il sequence point viene inserito *dopo* che *tutti* gli argomenti di una funzione (qui la `printf`) sono stati valutati e subito prima di eseguirla
    Non è dato sapere se viene valutata prima l'espressione `++x` e poi calcolata la funzione `tan()` o viceversa, ossia se `tan()` usa il valore di `x` prima o dopo l'incremento

Operatore virgola

- Due espressioni separate da una virgola diventano *sintatticamente* un'unica espressione (detta *comma expression*):
  ```
  i=0, j=9;
  ```
- Mentre due espressioni separate da un punto e virgola sono due normali espressioni indipendenti, anche se sono sulla stessa riga:
  ```
  i=0; j=9;
  ```
  equivale in tutto a:
  ```
  i=0;
  j=9;
  ```

Operatore virgola

- L'espressione composta risultante può essere inserita dove sintatticamente il linguaggio ne prevede una sola:

```
for (i=0, j=9; i<j ; i++, j--)
    if (v[i] != v[j])
        palindroma = 0;
```

- L'operatore virgola inserisce un *sequence point:* l'espressione di destra viene valutata solo dopo che l'espressione di sinistra è stata valutata completamente (ossia i suoi side-effects sono portati a termine)

Operatore virgola

- Se le due espressioni sono di tipo diverso:
  - il tipo e il valore dell'espressione composta sono sempre quelli dell'espressione a destra della virgola
  - tipo e valore dell'espressione di sinistra sono *scartati*
- L'operatore virgola ha la *priorità più bassa in assoluto* tra **tutti** gli operatori, quindi anche dell'operatore di assegnazione, allora l'espressione:

  `i=0, j=9`

  viene valutata come:

  `(i=0), (j=9)`

  e dunque `i` assume il valore 0 e `j` il valore 9

Operatore virgola

- Sempre per il fatto che l'operatore = ha priorità maggiore dell'operatore virgola, l'espressione:
  `x = 8, sqrt( 2*x );`
  viene valutata come:
  `(x = 8), (sqrt( 2*x ));`
  e dunque `x` assume il valore 8 e poi viene calcolata la `sqrt` del valore 16; l'espressione composta vale complessivamente 4.0 (il risultato dell'espressione a destra della virgola, notare che è un double), ma il valore non viene assegnato ed è quindi scartato (è un *expression statement*)

Operatore virgola

- Sono necessarie parentesi esplicite intorno a tutta la *comma expression* se si vuole assegnare il risultato di questa a una variabile:
  `double y = (x=8, sqrt( 2*x ));`
  Qui `x` assume il valore 8 e `y` il valore 4.0
- L'operatore virgola ha associatività da sinistra a destra, quindi più espressioni possono essere composte, ad esempio:
  `a = (x=8, y=x++, z=y+5);`
  viene valutata come:
  `a = ((x=8, y=x++), z=y+5);`
  e i risultati sono: `x`=8, `y`=9, `z`=13, `a`=13

Operatore virgola

- Altri esempi

```
int x, y;
double z;
x=2*4, 5*6;          x=8, 30 è scartato
x=(2*4, 5*6);        x=30, 8 è scartato
x=2*4, y=5*6;        x=8, y=30
x=(2*4, y=5*6);      8 scartato, y=30, x=30
x=(y=2*4, 5*6);      y vale 8, x = 30
x=(y=2*4, 5*6);      y vale 8, x = 30
x=2*4, z=5*6;        x=8, z=30.0 (double)
x=(2*4, z=5*6);      8 scartato, z=30.0, x=30 ma
```

dà Warning perché 30.0 è un `double` *e viene fatta un'assegnazione a* `x` *che è un* `int`

Operatore virgola

- In un contesto dove la virgola ha il significato di *separatore*, per inserire una comma expression si includono l'*operatore* virgola e i suoi operandi tra parentesi tonde:

  ```
  funz(a, (b=1,b+2), c);
  ```

  qui `funz` ha 3 parametri il secondo vale 3

- Analogamente, come argomento l'espressione di una precedente slide può essere scritta:

  ```
  y = sqrt((x=8,2*x));
  ```

  la coppia di parentesi più interna serve per far considerare la virgola un operatore e non il separatore dell'inesistente secondo argomento

Espressione condizionale

- È un'unica espressione (detta *ternaria*) che può assumere due valori in base a una condizione
- `x = (`*condizione*`) ? `*espr1*` : `*espr2*`;` equivale approssimativamente a:

```
if (condizione)
    x = espr1;
else
    x = espr2;
```

- *condizione* viene valutata completamente prima delle *expr* (inserisce un *sequence point*)
- Le parentesi sono opzionali ma consigliabili per chiarezza
- Viene calcolata *una sola* delle due *expr*

Espressione condizionale

- Esempi
  - Il maggiore tra `a` e `b`:

```
x = (a>b) ? a : b;
```

  - Il valore assoluto di `a`:

```
x = (a>0) ? a : -a;
```

  - La radice quadrata del valore assoluto di `a`:

```
x = (a>0) ? sqrt(a) : sqrt(-a);
```

  - Il plurale di una parola

```
printf("Ci sono %d oggett%c\n",
                    k, (k==1) ? 'o' : 'i');
```

Espressioni condizionali

- Il tipo del risultato è *sempre* il più "capiente" tra quelli prodotti dalle due *expr*

  `(x == 1) ? 11 : 12.0;`

  11 è un `int` e 12.0 è un `double`, quindi restituirà sempre un valore `double` (qui o 11.0 o 12.0)
- Un espressione condizionale non produce un L-value quindi non si può scrivere:

  `(a>b)? a : b = 12;`

  ma con i puntatori si potrà scrivere:

  `*(a>b ? &a : &b) = 12`

Espressioni condizionali

- Hanno associatività da destra a sinistra, questo significa che vengono raggruppate da destra a sinistra, ma quando sono annidate l'esecuzione prevede che vengano comunque eseguite da quelle più esterne a quelle più interne
- Esempio

```
x = (a>=b && a>=c) ? a : (b>=c)? b:c;
```

equivale a:

```
x = (a>=b && a>=c) ? a : ((b>=c)? b:c);
```

e non a:

```
x = ((a>=b && a>=c) ? a : (b>=c))? b:c;
```

Ossia viene valutata la condizione a sinistra che è la più esterna e, se è falsa, quella più interna

Espressioni condizionali

- Altro esempio

  `x = a ? b : c ? d : e ? f : g ? h : i;`

  equivale a:

  `x = a ? b : (c ? d : (e ? f : (g ? h : i)));`

  Ossia viene valutata la condizione `a`, se è vera viene restituito `b`, altrimenti viene valutata la condizione `c`, se è vera viene restituito `d`, ecc. Prima dell'assegnazione a `x`, il tipo della variabile temporanea è il più capiente tra quelli di `b`, `d`, `f`, `h` e `i`, mentre `a`, `c`, `e` e `g` sono espressioni logiche (vero/falso)
- Salvo rari casi, si *sconsiglia* di utilizzare espressioni condizionali annidate essendo spesso poco leggibili

Priorità e associatività

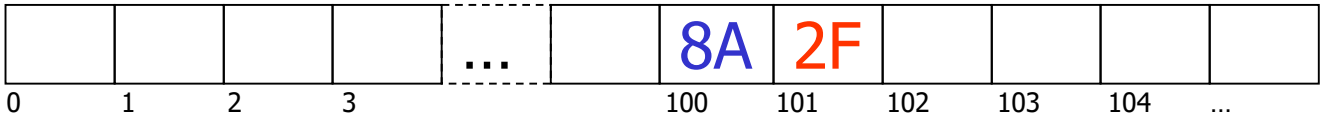
`() [] -> .`	S→D
`! ~ ++ -- + - * & (`*cast*`) sizeof`	S←D
`* / %`	S→D
`+ -` *(somma e sottrazione)*	S→D
`<< >>`	S→D
`< <= > >=`	S→D
`== !=`	S→D
`&`	S→D
`^`	S→D
`|`	S→D
`&&`	S→D
`||`	S→D
`?:`	S←D
`= += -= *= /= %= &= ^= |= <<= >>=`	S←D
`,`	S→D

- S→D: da Sinistra a Destra, S←D: da Dst. a Sin.

Big-endian e Little-endian

- Un valore scalare composto da più byte può essere collocato in memoria in diversi modi
- Si consideri uno `short` di 2 byte di valore $2F8A_{16}$, si ha una rappresentazione:
  - **big-endian** quando il byte *più significativo* dei due (il "big end", 2F) è memorizzato nel byte di indirizzo *più basso*

  - **little-endian** quando il byte *meno significativo* dei due (il "little end" , 8A) è memorizzato nel byte di indirizzo *più basso*



Big-endian e little-endian

- I processori Intel sono little-endian, gli ARM possono essere configurati in entrambi i modi (in genere little-endian)
- Il TCP/IP memorizza i dati come big-endian (detto per questo anche *network order*)
- I processori Motorola, i mainframe IBM e i supercomputer Cray sono big-endian
- Con valori di più di 2 byte ci possono essere rappresentazioni ulteriori

La funzione system

- Esegue un comando dell'*interprete dei comandi* della console
- È definita in `<stdlib.h>`
- Sintassi

  `system(`*stringa_con_comando*`);`
- Esempio

  Pulisce lo schermo:

  `system("CLS");` → *DOS/Windows*

  `system("clear");` → *Unix/Linux/OSX*
- Esempio

  Sospende l'esecuzione di un programma:

  `system("pause");` → *DOS/Windows*

Operatori bitwise

- Operano su valori *interi* con o senza segno a livello dei singoli bit di ogni valore
- Gli operandi sono **sempre convertiti** secondo le regole delle promozioni integrali

`<<`	SHIFT a sinistra	`x << 2`
`>>`	SHIFT a destra	`x >> 2`
`~`	complemento a 1	`~x`
`&`	AND bit a bit	`x & y`
`^`	EXOR bit a bit	`x ^ y`
`|`	OR bit a bit	`x | y`

- Per tutti gli operatori con due operandi esiste la forma di assegnamento abbreviata:

`&=` `^=` `|=` `<<=` `>>=`

Operatori bitwise << >>

- Gli operatori << e >> applicano all'operando di sinistra (in binario) uno *shift* (scorrimento di tutti i bit) rispettivamente a sinistra (<<) e a destra (>>) del numero di posizioni indicato dall'operando di destra
- Esempio

  `x = 3;` → `0000000000000011`

  `y = x << 2;` → `0000000000001100`

  fa uno shift di 11_2 (ossia $+2_{10}$) a sinistra di due posizioni: 1100_2 → $+8_{10}$
- Il numero di posizioni deve essere maggiore o uguale a zero e *strettamente minore* del numero di bit dell'operando di sinistra

Operatori bitwise << >>

- L'operatore `<<` aggiunge sempre bit 0 a destra del numero (in binario), i bit che fuoriescono a sinistra sono scartati
- L'operatore `>>` :
  - se il numero è positivo o `unsigned` aggiunge a sinistra dei bit pari a 0
  - se è negativo (ovviamente **`signed`**) il comportamento non è definito, *in genere* aggiunge degli 1 come estensione del segno (se sono valori in Complemento a 2), ma per una migliore portabilità del codice è bene convertirlo in numero positivo, fare lo shift e riconvertirle il risultato)

  e i bit che fuoriescono a destra sono scartati

Operatori bitwise ~, &, ^, |

- Gli operatori bitwise `~`, `&`, `|` e `^` corrispondono agli operatori logici `!`, `&&` e `||`, l'operatore di ExOR `^` non ha equivalente logico), ma operano *sui singoli bit* delle quantità
- Operatori logici e bitwise non sono equivalenti
- La priorità degli operatori bitwise è (in ordine decrescente):

  `~  &  ^  |`

  Si noti (vedere la tabella precedente) che:
  - `~` ha priorità tra le più altre in assoluto
  - `&`, `^` e `|` hanno priorità inferiore agli operatori relazionali e agli aritmetici, ma maggiore di quelli logici `&&` e `||`

Operatori bitwise ~

- È l'operatore di Complemento a 1, ossia inverte ciascuno dei bit del valore a cui è applicato, a seguito delle eventuali promozioni integrali

```
z = ~x;
```

- Esempi
  - `int x = 53;`
    53 in binario è: `00…00110101`
    `~x` dà: **11…11**`001`**0**`1`**0** (–54 se il bin è in Complemento a 2)
  - `~0` dà un valore pari a `11..11` (–1)

Operatore bitwise &

- `z = x & y;`
- Ciascuno dei bit di `z` viene determinato calcolando l'AND dei corrispondenti bit di `x` e `y` (dopo averli eventualmente promossi)
- L'operatore `&` viene spesso usato per azzerare ("unset" o "clear") alcuni dei bit di un valore dato (`x`) lasciando invariati gli altri
- Per specificare quali bit debbano essere azzerati si predispone una *maschera* di bit da mettere in AND bit a bit con `x`
- Trattando bit, è comodo che la maschera sia espressa come valore ottale o esadecimale

Operatore bitwise &

- La maschera è un numero intero, i bit del numero dato `x` in corrispondenza dei bit a 0 della maschera vengono azzerati (mascherati, non passano attraverso la maschera), mentre quelli in corrispondenza dei bit a 1 restano invariati (passano attraverso la maschera)
- Esempio

```
y = x & 015;
```

valore: `x` $=26_{10}$ → `0..011010`$_2$
maschera: 15_8 → `0..001101`$_2$
risultato: `y` → `0..010100`$_2$

Operatore bitwise |

- `z = x | y;`
- Ciascuno dei bit di `z` viene determinato calcolando l'OR dei corrispondenti bit di `x` e `y`
- L'operatore `|` viene spesso usato per impostare a 1 ("set") alcuni dei bit di un valore dato (`x`) lasciando invariati gli altri
- Per specificare quali bit debbano essere impostati a 1 si predispone una *maschera* di bit da mettere in OR bit a bit con `x`

Operatore bitwise |

- La maschera è un numero intero, i bit del numero dato `x` in corrispondenza dei bit a 1 della maschera vengono impostati a 1, mentre quelli in corrispondenza dei bit a 0 restano invariati
- Esempio

```
y = x | 012;
```

valore: `x` $=22_{10}$ → $0..010110_2$
maschera: 12_8 → $0..001100_2$
risultato: `y` → $0..011110_2$

Operatore bitwise ^

- `z = x ^ y;`
- Ciascuno dei bit di `z` viene determinato calcolando l'EXOR dei corrispondenti bit di `x` e `y` (dà 1 se i bit sono diversi)
- L'operatore `^` viene spesso usato per invertire alcuni dei bit di un valore dato (`x`) lasciando invariati gli altri
- Per specificare quali bit debbano essere invertiti si predispone una *maschera* di bit da mettere in EXOR bit a bit con `x`

Operatore bitwise ^

- La maschera è un numero intero, i bit del numero dato `x` in corrispondenza dei bit a 1 della maschera vengono invertiti, mentre quelli in corrispondenza dei bit a 0 restano invariati
- Esempio

```
y = x ^ 016;
```

valore: `x` $=22_{10}$ → `0..01`**`011`**0_2
maschera: 16_8 → `0..001110`$_2$
risultato: `y` → `0..01`**`100`**0_2

Operatori bitwise e logici

- Quando le espressioni collegate dagli operatori hanno solo valori 0 e 1, si potrebbero utilizzare gli operatori bitwise al posto degli operatori logici, ma:
  - la valutazione delle espressioni logiche non si ferma non appena il valore del risultato è individuabile
  - non c'è la garanzia di esecuzione da sinistra a destra degli operatori
- Potrebbe invece essere utile utilizzare l'operatore bitwise EXOR in quanto non esiste il corrispondente operatore logico, ma:
  `a` EXOR `b` → `(a && !b) || (!a && b)`

Operatori bitwise e logici

- Operatori bitwise e logici danno generalmente risultati diversi:
  - 1 & 2 → $00\ldots01_2$ & $00\ldots10_2$ → $00\ldots00_2$ → 0
  - 1 && 2 → 1
  - 2 & 3 → $00\ldots10_2$ & $00\ldots11_2$ → $00\ldots10_2$ → 2
  - 2 && 3 → 1
  - 1 & 0 → $00\ldots01_2$ & $00\ldots00_2$ → $00\ldots00_2$ → 0
  - 1 && 0 → 0

Esercizi

1. Determinare di quanti bit è composto un tipo `char`, `short`, `int` e `long` contando i bit.
2. Visualizzare in binario il valore intero (decimale con segno) dato in input.
3. Scrivere un programma che chieda un valore intero decimale, lo visualizzi in binario e calcoli quanti sono i bit pari a 1 che contiene.
4. Scrivere un programma che chieda un valore decimale senza segno (da collocare in una variabile `unsigned int`) e "ruoti" i bit di *n* posizioni a sinistra o a destra a richiesta ("ruotare" i bit significa che quelli che escono da una parte entrano dall'altra).

Homework 3-4

Metodo di ordinamento Radix Sort

- Per ordinare numeri interi *positivi*
- Lavora facendo riordinamenti parziali prima sulle unità, poi sulle decine, poi sulle centinaia, ecc.
- Esempio
  - Dati i valori: 7, 12, 9, 25, 36, 11, 20, 4
  - Scriverli utilizzando per tutti lo stesso numero di cifre: 07, 12, 09, 25, 36, 11, 20, 04
  - Definire una matrice avente per righe i valori delle cifre e per colonne i numeri dati
  - Collocare i numeri sulla riga corrispondente alla cifra delle unità, ma nella stessa colonna

Homework 3-4

Collocamento dei valori in base alla cifra unità

	07	**12**	**09**	**25**	**36**	**11**	**20**	**04**
0							20	
1						11		
2		12						
3								
4								04
5				25				
6					36			
7	07							
8								
9			09					

Homework 3-4

- Raccogliere i valori riga per riga e ricollocarli in base alla cifra decine

	20	**11**	**12**	**04**	**25**	**36**	**07**	**09**
0				04			07	09
1		11	12					
2	20				25			
3						36		
4	. . . . .							

- Raccoglierli nuovamente riga per riga (se ci fossero più cifre l'operazione andrebbe ripetuta sulle centinaia, sulle migliaia, etc.)

04 07 09 11 12 20 25 36

Homework 3

Si scriva un programma che realizzi il metodo di ordinamento Radix Sort di valori decimali (max 100 valori), considerando le singole cifre *decimali* come visto nell'esempio, i valori vengano assegnati in decimale mediante `scanf`.

Homework 4

Si scriva un programma che realizzi il metodo di ordinamento Radix Sort di valori decimali (max 100 valori), considerando le singole cifre in *binario* (il valore binario equivalente al numero decimale) e utilizzando gli operatori bitwise sulle variabili), i valori vengano assegnati in decimale mediante `scanf`.