Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2022/23

A. Apicella

TIME

`#include <time.h>`

Header per sfruttare funzioni di libreria atte al fornire informazioni riguardo al "tempo".

Funzioni principali:

- `time_t time( time_t* arg );`

restituisce un object di tipo `time_t` contenente la data e l'ora attuali. Generalmente, `time_t` corrisponde ad un semplice intero positivo contenente il numero di secondi trascorsi dal 1 gennaio 1970 fino ad oggi. L'argomento `arg` è un puntatore che, al termine della funzione, conterrà lo stesso valore restituito da quest'ultima (può quindi essere ignorato passandogli `NULL`);

In [3]:
```
#include <stdio.h>
#include <time.h>
int main()
{
    time_t sec_dal_1970 =  time(NULL);
    printf("numero di secondi trascorsi dal 1 gennaio 1970: %ld\n",se
    printf("numero di anni dal 1 gennaio 1970: %f\n", sec_dal_1970/60
    return 0;
}
```

```
numero di secondi trascorsi dal 1 gennaio 1970: 1684510298
numero di anni dal 1 gennaio 1970: 53.415471
```

In [ ]:
```

```

- `struct tm* gmtime( const time_t *timer );`

Dato in input un numero di secondi trascorsi dal 1 gennaio 1970, restituisce un puntatore ad una `struct` contenente la data (UTC) distribuiti su vari campi.

`struct tm {`

int tm_sec	seconds after the minute;
int tm_min	minutes after the hour;
int tm_hour	hours since midnight; [0, 23]
int tm_mday	day of the month; [1, 31]
int tm_mon	months since January; [0, 11]
int tm_year	years since 1900;
int tm_wday	days since Sunday; [0, 6]
int tm_yday	days since January 1; [0, 365]

`};`

In [2]:
```
#include <stdio.h>
#include <time.h>
int main()
{
    time_t sec_dal_1970 =  time(NULL);
    printf("numero di secondi trascorsi dal 1 gennaio 1970: %ld\n", s
    struct tm* data    = gmtime(&sec_dal_1970);
    printf("data (UTC): giorno:%d mese:%d anno:%d ore:%d minuti:%d se
           data->tm_mday, data->tm_mon+1,1900+data->tm_year,
           data->tm_hour, data->tm_min, data->tm_sec);

    return 0;
}
```

```
numero di secondi trascorsi dal 1 gennaio 1970: 1684567907
data (UTC): giorno:20 mese:5 anno:2023 ore:7 minuti:31 secondi:47
```

- `struct tm *localtime ( const time_t *timer );`

come `gmtime`, ma la data viene espressa secondo il fuso orario locale

- `char* asctime(const struct tm *timeptr);`

Dato l'indirizzo di una struct di tipo `tm`, restituisce una stringa contenente la data già formattata

In [3]:
```
#include <stdio.h>
#include <time.h>
int main()
{
    time_t sec_dal_1970 =  time(NULL);
    printf("numero di secondi trascorsi dal 1 gennaio 1970: %d\n", se
```

```
    struct tm* data    = localtime(&sec_dal_1970);
    char* string_data  = asctime(data);
    printf("data (locale): %s\n", string_data );

    return 0;
}
```

```
numero di secondi trascorsi dal 1 gennaio 1970: 1649245935
data (locale): Wed Apr  6 13:52:15 2022
```

Random numbers

La funzione `rand()` restituisce un numero *pseudo*-casuale nell'intervallo tra 0 e `RAND_MAX`.

`RAND_MAX` è una costante il cui valore può essere diverso in base alle diverse implementazioni, ma sicuramente è ≥ 32767

In [15]:
```
#include <stdio.h>
#include <time.h>
int main()
{

    printf("RAND_MAX: %lld", RAND_MAX);
    return 0;
}
```

```
RAND_MAX: 2147483647
```

In [16]:
```
#include <stdio.h>
#include <time.h>
int main()
{
    long int r = rand();
    printf("%lld\n", r);
    r = rand();
    printf("%lld\n", r);
    r = rand();
    printf("%lld\n", r);
    r = rand();
    printf("%lld\n", r);
    return 0;
}
```

```
1804289383
846930886
1681692777
1714636915
```

In altri termini, la funzione `rand()` genera un numero intero $\in \{0, 1, 2, \dots, RAND\_MAX\}$ ad ogni invocazione.

Attenzione però che diverse esecuzioni dello *stesso programma* daranno sempre lo stesso numero! Nell'esempio precedente, mi aspetto che ad ogni esecuzione del programma sia restituita sempre la stessa sequenza di numeri (ossia 1804289383, 846930886, 1681692777, 1714636915). Questo perchè, essendo *pseudo*-casuali (e non casuali), tali numeri sono generati sempre in funzione del numero casuale precedente ottenuto *durante l'esecuzione*. In altri termini, durante una singola esecuzione del programma, il t-esimo numero pseudocasuale N_t generato da `rand()` è il risultato di una funzione che lavora sul numero casuale precedente generato da `rand()`. Formalmente, possiamo dire che N_t cambia in funzione del valore precedente, i.e. $N_t = f(N_{t-1})$. Il primo di questi valori, $N_1 = f(N_0)$, dipende da un valore iniziale detto $seme$ N_0, tipicamente pari a 1.

Un modo per avere valori diversi ad ogni esecuzione è quindi quello di utilizzare semi diversi ad ogni esecuzione del programma (in altri termini, inizializzare N_0 con valori diversi ad ogni esecuzione del programma). la funzione

```
void srand( unsigned seed );
```

imposta il seme al valore `seed`.

L'ideale sarebbe avere un seme diverso ad ogni esecuzione $\rightarrow$ si può usare il numero di secondi restituiti da `time(...)`.

In [17]:
```
#include <stdio.h>
#include <time.h>
int main()
{
    srand(time(NULL));
    long int r = rand();
    printf("%lld\n", r);
    r = rand();
    printf("%lld\n", r);
    r = rand();
    printf("%lld\n", r);
    r = rand();
    printf("%lld\n", r);
}
```

```
1572501703
2021161100
1915516915
624023057
```

Supponendo che trascorra almeno un secondo tra più esecuzioni di questo programma, dato che il valore del seme sarà diverso ad ogni esecuzione, la sequenza di valori random generati sarà diversa per ogni esecuzione.

Generare un numero **intero** random $\in \{0, 1, 2, \dots, b\}$

Esempio: voglio un numero casuale tra 0 e 5 (estremi compresi).

Possibile ragionamento: sfruttare l'operatore modulo. Si ricorda che, dati due numeri interi d e u, $mod(d,u)$ sarà un numero intero tale che $0 \leq mod(d,u) \leq u-1$.

$mod(1,5)=1,\ mod(2,5)=2,\ mod(3,5)=3,\ mod(4,5)=4,\ mod(5,5)=0$

Quindi, per avere un numero tra 0 e 5, nell'operazione $mod(d,u)$ basta impostare $u=5+1=6$ mentre d può essere qualsiasi numero (pseudo)casuale, i.e.

$$0 \leq mod(rand(), b+1) \leq b$$

In [18]:
```
#include <stdio.h>
#include <time.h>

long int random_int_in_max(int b)
{
    return rand() % (b + 1);
}


int main()
{
    int max_number = 5;
    srand( time(NULL) );
    for (int i = 0; i < 50; i++)
    {
        long int r = random_int_in_max(max_number);
        printf("%lld ", r);
    }

}
```

```
5 0 1 4 3 2 2 0 4 5 2 3 4 2 1 3 2 5 5 5 0 3 0 0 4 0 0 5 5 5 3 4 3 2
0 1 4 2 5 0 1 0 2 4 2 3 1 4 2 0
```

Generare un umero **intero** random $\in \{a, a+1, a+2, \ldots, b\}$

Esempio: voglio un numero casuale tra 2 e 5 (estremi compresi).

Possibile ragionamento: innanzitutto devo sapere *quanti* sono i numeri effettivi nell'intervallo desiderato. In questo caso, il numero desiderato può essere uno nell'insieme $\{2,3,4,5\}$, per un totale di $w=4$ numeri. Generalizzando, $w=b+1-a$.

$c = mod(rand(), w)$ mi darà un numero casuale intero il cui valore sarà uno tra 0 e $w-1$.

Sommando al valore c il valore a, si avrà quindi un numero casuale compreso tra $0+a=a$ e $w-1+a=(b+1-a)-1+a=b$.

In [5]:
```
#include <stdio.h>
#include <time.h>

long int random_int_in_range(int a, int b)
{
    return rand() % (b + 1 - a) + a;
}



int main()
{
    srand( time(NULL) );
    for(int i = 0; i<30; i++)
    {
        long int r = random_int_in_range(2,5);
        printf("%lld ", r);
    }
    return 0;
}
```

```
5 4 3 5 5 5 5 5 4 4 2 5 3 3 2 5 5 3 4 2 2 3 4 4 2 2 5 2 3 2
```

Generare un numero **reale** random $\in [0,1]$

$0 \leq rand() \leq RAND\_MAX \Rightarrow 0.0 \leq \frac{rand()}{RAND\_MAX} \leq 1.0$

In [20]:
```
#include <stdio.h>
#include <time.h>

double random_real_in_0_1()
{

    double x =  (float)rand() / RAND_MAX;
    return x;
```

```
}


int main()
{
    srand( time(NULL) );
    for(int i = 0; i<20; i++)
    {
        double r = random_real_in_0_1();
        printf("%lf ", r);
    }
    return 0;
}
```

```
0.310926 0.685900 0.302752 0.254879 0.047773 0.926345 0.087862 0.852
664 0.104581 0.785533 0.016734 0.426548 0.722991 0.011384 0.020128
0.640147 0.550135 0.839033 0.135204 0.944068
```

Generare un numero **reale** random $\in [0, b]$

In [1]:
```
#include <stdio.h>
#include <time.h>

double random_real_in_max(int b)
{

    double x =  ((float)rand() / RAND_MAX) * b;
    return x;
}


int main()
{
    srand( time(NULL) );
    for(int i = 0; i<20; i++)
    {
        double r = random_real_in_max(5);
        printf("%lf ", r);
    }
    return 0;
}
```

```
1.945090 0.593442 4.638261 0.794654 2.835281 2.727794 4.567822 2.789
679 0.416285 4.897671 1.439158 1.669975 0.301105 0.135909 3.934027
2.564836 2.161356 4.207200 4.511484 4.630343
```

Generare un numero **reale** random $\in [a, b]$

Calcolo la distanza tra b ed a, ossia $w = b - a$.

Dato che $0 \leq \frac{rand()}{RAND\_MAX} \leq 1$, allora $w \cdot 0 \leq w \cdot \frac{rand()}{RAND\_MAX} \leq w \cdot 1$.

Sommando a:

$$a + 0 \leq a + w \cdot \frac{rand()}{RAND\_MAX} \leq a + w = a + b - a = b.$$

Quindi: $a \leq a + (b - a) \cdot \frac{rand()}{RAND\_MAX} \leq b$

In [6]:
```
#include <stdio.h>
#include <time.h>

double random_real_in_range(int a, int b)
{

    double x = ((b - a) * ((double)rand() / RAND_MAX)) + a;
    return x;
}


int main()
{
    srand( time(NULL) );
    for(int i = 0; i<100; i++)
    {
        double r = random_real_in_range(1,2);
        printf("%lf ", r);
    }
    return 0;
}
```

```
1.571983 1.036639 1.856773 1.002382 1.501812 1.947896 1.944476 1.561
045 1.482949 1.864293 1.837613 1.474696 1.919647 1.270162 1.042264
1.890886 1.126113 1.479243 1.303184 1.307271 1.036848 1.014611 1.182
290 1.875430 1.805373 1.309830 1.572245 1.653529 1.573379 1.587789
1.830877 1.145362 1.624427 1.687650 1.147744 1.126239 1.635547 1.092
220 1.687285 1.118496 1.956513 1.524897 1.593192 1.876160 1.795059
1.635456 1.767046 1.921173 1.114699 1.070229 1.228443 1.151547 1.084
840 1.410733 1.026977 1.890213 1.720563 1.599222 1.543742 1.293942
1.187010 1.374620 1.439304 1.811438 1.062270 1.587048 1.937677 1.697
817 1.679268 1.624962 1.816312 1.635782 1.149859 1.409504 1.511942
1.944918 1.044960 1.278987 1.866091 1.159660 1.349216 1.094534 1.311
207 1.434056 1.505267 1.338184 1.324269 1.225830 1.937406 1.868012
1.519772 1.124416 1.242631 1.959076 1.935854 1.304901 1.546124 1.873
532 1.002718 1.225392
```