

Università di Pisa
Corso di Laurea Magistrale in Fisica



Appunti di Algoritmi di Spettroscopia

Giovanni Moruzzi

Anno accademico 2012/2013

versione del 6 aprile 2013

Indice

1	Impariamo a leggere e a scrivere	1
1.1	Interazione con tastiera e monitor	1
1.2	Una prima funzione	2
1.3	Lettura un po' più avanzata da tastiera	4
1.4	Lettura e scrittura di files	6
2	Grafica sotto il protocollo X Window	11
2.1	Introduzione alla grafica sotto X Window	11
2.2	Definiamo la struttura XVideoData	16
2.3	Semplice animazione	18
3	Libreria	25
3.1	Il file delle dichiarazioni delle funzioni	25
3.2	Le funzioni StartXWindow e CloseXWindow	28
3.3	Funzioni grafiche	33

Capitolo 1

Impariamo a leggere e a scrivere

1.1 Interazione con tastiera e monitor

Scriviamo un programma, che possiamo chiamare `leggi.cc`, che legga quello che noi scriviamo sulla tastiera e scriva qualcosa sul monitor.

```
1 #include <stdio.h>
2
3 main()
4 {
5     char risposta[80];
6
7     for (;;)
8     {
9         printf("scrivi qualcosa:\n");
10        fgets(risposta,80,stdin);
11        printf("hai scritto:%s\n\n",risposta);
12        if (risposta[0]=='x') break;
13    }
14 }
```

Listato 1.1 Lettura da tastiera e scrittura su monitor

La riga 1 include lo *header* `stdio.h`, che contiene le definizioni delle funzioni `printf()` e `fgets()` usate nel programma. La riga 3 dichiara l'inizio del codice del programma principale alla successiva riga 4, con la parentesi a grappa aperta. Il programma principale termina alla riga 14 con la parentesi a grappa chiusa. Alla riga 5 viene definito un *array* di 80 caratteri chiamato `risposta`. In fase di compilazione, al codice `risposta` verrà associato un indirizzo in memoria corrispondente al primo degli 80 caratteri disponibili, numerati da `risposta[0]` a `risposta[79]`.

Alla riga 7 viene dichiarato un *loop infinito* del tipo `for(;;)` con codice racchiuso dalle parentesi a graffe dalla riga 8 alla riga 13. L'istruzione `printf()` della riga 9 scrive sul monitor del computer `scrivi qualcosa: .` Notare che l'andata a capo `\n` nella stringa di formattazione è necessaria: senza di essa i caratteri precedenti della stringa di formattazione non verrebbero scritti, ma resterebbero in memoria in attesa di una ulteriore sequenza di caratteri terminata da una `\n`.

L'istruzione `fgets` della riga 10 ha tre argomenti. Un indirizzo di memoria, in questo caso l'indirizzo di inizio dell'*array* `risposta` dove scrivere, un numero massimo di caratteri da leggere, in questo caso 80, e un indirizzo da cui leggere, in questo caso il codice `stdin` sta per *standard input*, ovvero la tastiera del computer. Giunta a questa istruzione, l'esecuzione del programma si ferma ed aspetta che noi scriviamo qualcosa sulla tastiera. Il codice corrispondente al primo tasto che battiamo viene memorizzato all'indirizzo `risposta[0]`, il secondo all'indirizzo `risposta[1]` e così via, fino a che non battiamo una *andata a capo* (tasto *Enter*, *Invio*, o simile), che viene memorizzato con il codice 0, e a questo punto si passa all'esecuzione dell'istruzione successiva del programma.

La funzione `printf()` alla riga 11 ha due argomenti: la stringa di formattazione e l'indirizzo di `risposta`. La funzione copia sul monitor tutti i caratteri del *format* fino alla prima (e, in questo caso, unica) sequenza di caratteri che inizia con `%`. A questo punto entra in gioco il secondo argomento. Il fatto che la sequenza sia `%s` dice che il secondo argomento va interpretato come un *array* di caratteri. Viene quindi stampato il carattere all'indirizzo `risposta[0]`, poi quello all'indirizzo `risposta[1]`, e così via fino a quando non si trova un carattere col codice 0, che non viene stampato, ma serve a dire che la “stringa” è terminata.

L'istruzione alla riga 13 controlla se il primo carattere che avevamo battuto era una *x*. In questo caso il “loop infinito” viene interrotto dall'istruzione `break`, altrimenti si procede all'iterazione successiva del loop, ricominciando dalla riga 9.

1.2 Una prima funzione

Abbiamo visto che la funzione `fgets()` alla riga 10 del listato 1.1, con gli argomenti impostati, memorizza quanto noi battiamo da tastiera fino a quando non premiamo il tasto *Invio* (o fino a quando non abbiamo battuto il limite di 80 caratteri, per non correre il rischio di sovrapposizione e corrispondente “corruzione” di memoria). Questo ci permette di passare da tastiera al programma una certa quantità di informazione, in particolare possiamo passare anche informazioni separate, in una singola mandata. Possiamo anche passare un numero diverso di “informazioni separate” ad ogni interazione, ma in questo caso dobbiamo dare al programma la capacità di prenderne atto. Un modo possibile è quello di considerare la nostra stringa di risposta come separata in tanti campi, ognuno relativo ad una singola informazione, mediante uno o più spazi bianchi tra un campo e l'altro. Alla fine il nostro *array* di caratteri, o *stringa*, cui punta `risposta`, potrebbe avere l'aspetto

```
gatto    15.572   347 ornitorinco    3.14159                                     (1.1)
```

che corrisponde a 5 campi diversi. Naturalmente potremmo separare i campi in altro modo, per esempio mediante virgole, come nel formato *csv* (*comma separated variables*), ma qui useremo gli spazi. In quanto segue, quando abbiamo una stringa divisa in N campi, considereremo i campi numerati da 0 a $N - 1$, partendo dal campo più a sinistra.

La prima funzione che scriveremo per la nostra libreria, la `fieldcpy.cc` del listato 1.2, riceverà in ingresso l'indirizzo di una stringa separata in campi mediante spazi o sequenze di spazi, un numero intero corrispondente al numero d'ordine n del campo che vogliamo estrarre, e l'indirizzo della stringa su cui vogliamo copiare il campo numero n .

Le righe 4-8 del codice servono ad includere gli *header files*. Dei files inclusi qui, il file `math.h` contiene le dichiarazioni delle funzioni matematiche, `stdlib.h` (libreria standard) è lo header file che dichiara funzioni e costanti di utilità generale, come l'allocazione della memoria, mentre `string.h` contiene definizioni di macro, costanti e dichiarazioni di funzioni e tipi usati nella manipolazione delle stringhe e della memoria. Infine il file `corso.h`, di cui trovate il codice

```

1 // ..... fieldcpy.cc
2 // ..... versione 08.APR.2003
3
4 #include <math.h>
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8 #include <corso.h>
9
10 void fieldcpy(char *dest, char *source, int n)
11 {
12     int i, nField;
13     unsigned char *ptr;
14
15     // ..... void string
16     dest[0]=0;
17     // ..... negative field number or void source string
18     if (n<0 || source[0]==0) return;
19     // ..... search for first non-blank byte
20     ptr=(unsigned char *)source;
21     while(ptr[0]<=' ')
22     {
23         ptr++;
24         if (ptr[0]==0) return;
25     }
26     // ..... count fields
27     nField=0;
28     while (nField<n)
29     {
30         while(ptr[0]>' ') {ptr++;if (ptr[0]==0) return;}
31         while(ptr[0]<=' ') {ptr++;if (ptr[0]==0) return;}
32         nField++;
33     }
34     // ..... copy nth field
35     i=0;
36     for (;;)
37     {
38         if (ptr[i]<=' ') {dest[i]=0;break;}
39         dest[i]=ptr[i];
40         i++;
41     }
42 }

```

Listato 1.2 Estrazione di un campo da una sequenza di caratteri

nel listato 3.1, contiene le dichiarazioni delle funzioni che svilupperemo in questo corso. La riga 10 dice che la funzione `fieldcpy()` non restituisce direttamente valori (`void`), e che i suoi tre

argomenti sono

1. l'indirizzo `dest` dell'array di caratteri in cui il campo prescelto deve essere copiato;
2. l'indirizzo `source` dell'array di caratteri da cui estrarre il campo;
3. l'intero n che identifica il campo da copiare.

La riga 12 dichiara le variabili intere `i` e `nField`, mentre la riga 13 dichiara che la variabile `ptr` è un puntatore ad un array di caratteri *non segnati*.

La riga 16 copia il codice 0 nel primo carattere di `dest`. In questo modo, nel caso che il campo da copiare non esista, `dest` corrisponderà a una stringa vuota. La riga 18 controlla che il numero del campo richiesto, n , non sia negativo, e che la stringa da copiare non sia vuota: in ognuno di questi casi la funzione ritorna.

La riga 20 copia nel puntatore `ptr` l'indirizzo dell'inizio della stringa `source`. Il loop delle righe 21-25 a ogni iterazione controlla se il carattere puntato da `ptr` ha un codice ≤ 32 (decimale), in questo caso il puntatore viene avanzato di una posizione (riga 23) per leggere il carattere successivo. Il codice 32, infatti, corrisponde allo spazio, che noi utilizziamo per separare un campo dall'altro. I codici inferiori a 32 sono invece codici non stampabili, che comprendono, tra gli altri, il TAB e l'andata a capo. Se, dopo l'avanzamento, `ptr` punta ad un codice 0, questo indica la fine della stringa `source`, quindi non c'è più niente da leggere e si ritorna al programma che ha chiamato la funzione (riga 24).

Quando usciamo dal loop `ptr[0] > 32`, il codice è stampabile, e quindi comincia il primo campo della nostra stringa, il campo numero 0. La variabile `nField`, che numera il campo attivo al momento viene così posto uguale a 0 alla riga 27.

Se il campo richiesto, n , è maggiore di zero, il puntatore viene avanzato dal loop delle righe 28-33. Il loop interno alla riga 30 avanza il puntatore finché il carattere puntato è > 32 , cioè finché siamo all'interno dello stesso campo. Ad ogni avanzamento di `ptr` viene controllato se il codice puntato è 0: in questo caso la stringa `source` è finita, e si esce dalla funzione.

All'inizio della riga 31, quindi, abbiamo $0 < ptr[0] \leq 32$, e siamo nell'intervallo tra due campi. Il loop della riga 31 è uguale al loop delle righe 21-25, e serve per avanzare il puntatore fino al campo successivo. Alla riga 32 viene aggiornato il "contatore dei campi" `nField`. Se `nField` è uguale a n si esce dal loop passando alla riga 35, altrimenti si prosegue con le iterazioni fino a che `nField` non raggiunge n .

Alla riga 35 il contatore `i` viene posto uguale a 0. Poi il loop delle righe 36-41 copia il campo numero n in `dest`. Alla riga 38 se `ptr[i] ≤ 32` (fine del campo) viene posto `dest[i]=0` (fine della stringa `dest`) e si esce dal loop. alla riga 39 il carattere `ptr[i]` viene copiato in `dest[i]`, alla riga 40 la variabile `i` viene aumentata di 1, poi si passa all'iterazione successiva.

Alla fine, nella stringa `dest` è stato copiato il campo numero n della stringa `source`, che è rimasta inalterata. Per esempio, se `source` è la stringa (1.1) e n vale 0, `dest` conterrà i caratteri `gatto`. Se invece n vale 2 `dest` conterrà i caratteri `347`. Attenzione, in questo ultimo caso la stringa contiene i tre *caratteri* 3, 4 e 3, non il numero intero 347 né una sua codifica.

1.3 Lettura un po' più avanzata da tastiera

Vediamo adesso come può essere usata la funzione `fieldcpy()` del listato 1.2 in una variante del programma del listato 1.1, mostrata qui di seguito. Alla riga 2 è importante includere il file

`corso.h`, che contiene la dichiarazione di `fieldcpy()`. La riga 6 dichiara la variabile intera `i` e la riga 7 dichiara i puntatori alle stringhe `buff` e `risposta`, rispettivamente di 80 e di 120 caratteri. Il loop principale del programma è codificato alle righe 9-24. Le righe 11-12 sono

```

1 #include <stdio.h>
2 #include <corso.h>
3
4 main()
5 {
6     int i;
7     char buff[80], risposta[120];
8
9     for (;;)
10    {
11        printf("scrivi qualcosa:\n");
12        fgets(risposta, 80, stdin);
13        i=0;
14        for (;;)
15        {
16            fieldcpy(buff, risposta, i);
17            if (buff[0]==0) break;
18            if (i==0) printf("hai scritto %s\n", buff);
19            else printf("e anche %s\n", buff);
20            i++;
21        }
22        if (risposta[0]=='x') break;
23        printf("\n");
24    }
25 }

```

Listato 1.3 Lettura leggermente più avanzata da tastiera e scrittura su monitor

equivalenti alle righe 9-10 del listato 1.1. Il loop alle righe 14-21 analizza la stringa `risposta` letta alla riga 12. Alla prima iterazione (con `i` posto uguale a 0 alla riga 13) il campo numero 0 di `risposta` viene copiato sulla stringa `buff`. Se `buff` è vuoto la riga 17 fa uscire dal loop e passare alla riga 22. Altrimenti, sempre con `i=0`, la riga 18 scrive sul monitor solo il campo n. 0 di `risposta`, e la riga 19 viene saltata. Alla riga 20 `i` viene incrementato di 1, poi si passa alla iterazione successiva. Questa volta la riga 16 copia in `buff` il campo n. 1, e, se non è vuoto, la riga 18 viene saltata e la riga 19 scrive `buff`. E così via finché tutti i campi di `risposta` sono stati letti. A questo punto la riga 17 fa uscire dal loop. La riga 22 controlla se l'ultima `risposta` letta cominciava per `x`, e in questo caso si esce dal loop principale e poi dal programma. Altrimenti viene letta una nuova `risposta`.

L'esecuzione del programma listato in 1.3 su di un terminale aperto sul monitor

```

giovanni@moruzzi3: ~/xcpp/algorithmi/lavoro
File Edit View Search Terminal Help
moruzzi3:giovanni:~/xcpp/algorithmi/lavoro>leggi2
scrivi qualcosa:
15 aragosta 132.777 gatto
hai scritto 15
e anche aragosta
e anche 132.777
e anche gatto

scrivi qualcosa:
ieri domani
hai scritto ieri
e anche domani

scrivi qualcosa:
xenon argon elio
hai scritto xenon
e anche argon
e anche elio
moruzzi3:giovanni:~/xcpp/algorithmi/lavoro>

```

Figura 1.1 Esecuzione del programma 1.3 su un terminale.

del nostro computer è mostrata in fig. 1.1. A ogni iterazione il programma chiede di scrivere qualcosa, poi aspetta. A questo punto noi battiamo la nostra risposta sulla tastiera, terminata dalla premuta del tasto *Invio*. A questo punto riprende l'esecuzione del programma, e la nostra risposta viene analizzata dal loop alle righe 14-21. Il programma termina quando noi battiamo una risposta che ha *x* come primo carattere.

1.4 Lettura e scrittura di files

```
1 15.5
2 26.37
3 6.3
4 36
5 55.327
6 100
7 15
```

Listato 1.4 Esempio di file di dati

Naturalmente leggere da tastiera e scrivere sul monitor è importante, ma non sufficiente. Normalmente dovremo elaborare grandi quantità di dati contenuti in files. Il nostro programma dovrà essere in grado di leggerli. Una volta elaborati, i dati ci forniranno dei risultati che, perché non vadano persi, il programma dovrà scrivere in altri files. Per vedere come leggere da un file e scrivere su un altro file, cominciamo scrivendo, con un editor qualunque (ma che permetta di scrivere in modo *non formattato*, cioè puramente ASCII), un file di dati tipo quello del listato 1.4. Al file diamo, per esempio, il nome `dati.txt`. Per leggerlo scriviamo il file del listato 1.5, che possiamo chiamare `scrivifile.cc`. Il listato comincia con la solita inclusione degli header files. All'interno del programma principale, nelle dichiarazioni delle variabili, la riga 14 dichiara la variabile `hnd` come puntatore ad un file.

Listato 1.5 Programma che legge da un file e scrive su un altro file

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <math.h>
5 #include <corso.h>
6
7 main()
8 {
9     int i, nDati;
10    double x;
11    double *xx;
12    char *ptr;
13    char buff[80], filename[80], risposta [120];
14    FILE *hnd;
15
16    printf("Scrivi il nome del file da leggere:\n");
17    fgets(filename, 80, stdin);
18    if (ptr=strchr(filename, '\n')) ptr[0]=0;
19    if ((hnd=fopen(filename, "r"))==NULL)
20    {
21        printf("Non riesco ad aprire il file %s!!!\n\n", filename);
22        exit(0);
23    }
24    // ..... conta i dati
25    nDati=0;
26    for (;;)

```

```

27 {
28     fgets (risposta ,80 ,hnd);
29     if (feof(hnd)) break;
30     fieldcpy (buff ,risposta ,0);
31     if (buff[0]==0) continue;
32     nDati++;
33 }
34 printf ("Nel file ci sono %d dati\n" ,nDati);
35 rewind (hnd);
36 // ..... riserva la memoria per i dati
37 xx=new double [nDati];
38 // ..... leggi i dati
39 i=0;
40 for (;;)
41 {
42     fgets (risposta ,80 ,hnd);
43     if (feof(hnd)) break;
44     fieldcpy (buff ,risposta ,0);
45     if (buff[0]==0) continue;
46     xx[i]=atof (buff);
47     i++;
48 }
49 fclose (hnd);
50 // ..... scrivi i dati
51 printf ("Scrivi il nome del file da scrivere:\n");
52 fgets (filename ,80 ,stdin);
53 if (ptr=strchr (filename ,'\n')) ptr[0]=0;
54 if ((hnd=fopen (filename ,"w"))==NULL)
55 {
56     printf ("Non riesco ad aprire il file %s!!!\n\n" ,filename);
57     exit (0);
58 }
59 for (i=0;i<nDati;i++)
60 {
61     fprintf (hnd ,"%15.5lf %15.5lf\n" ,xx[i] ,sqrt (xx[i]));
62 }
63 fclose (hnd);
64 }

```

La riga 16 chiede il nome del file da leggere, che deve essere battuto sulla tastiera per venire copiato sulla stringa `filename` alla riga 17. La riga 18 serve per cancellare un eventuale *andata a capo* presente nella stringa, rappresentata dal carattere `\n` (certi programmi, nel leggere la nostra risposta da tastiera, copiano anche l'andata a capo finale, che inseriscono prima del carattere 0 che segna la fine della stringa). A questo punto il nome del file che vogliamo leggere, per esempio `dati.txt`, è copiato nella stringa `filename`. Nella riga 19 abbiamo un `if` che condiziona l'esecuzione del codice tra le parentesi a grappa delle righe 20 e 23. All'interno delle parentesi tonde dell'`if` prima la funzione `fopen()` cerca sul disco un file con il nome copiato in `filename` e, se lo trova, lo apre (tutte le funzioni che maneggiano files anno il nome che comincia per `f`), e copia l'indirizzo del suo inizio sul puntatore `hnd`. Se il file non viene trovato la funzione copia in `hnd` l'indirizzo 0 (NULL). Il secondo argomento di `fopen()`, "`r`" dice di aprire il file in modo *lettura* (`r` per *read*). Se il file non viene trovato, per esempio perché abbiamo sbagliato a battere il nome, viene eseguito il codice delle righe 21 e 22: il programma

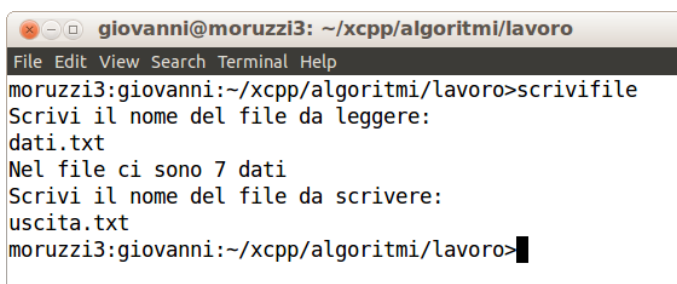
scrive sul monitor che non è riuscito ad aprire il file, ed esce.

Altrimenti alla riga 25 la variabile intera `nDati` viene posta uguale a 0, e, con il loop della riga 26 si comincia a leggere il file per contare quanti dati contiene. Alla prima iterazione la funzione `fgets()` alla riga 28 legge la prima riga del file puntato da `hnd` e la copia sulla stringa `risposta`. Il secondo argomento di `fgets()` dice di troncare la riga se contiene più di 80 caratteri. Uscendo dalla funzione, il puntatore `hnd` viene avanzato all'inizio della seconda riga. La riga 30 interrompe il loop se il file da leggere è finito: nel nome della funzione `feof()` la prima `f` ci dice che si tratta di una funzione che maneggia files, mentre le lettere `eof` stanno per *end of file*. Alla riga 30 `fieldcpy` copia il primo campo di `risposta` in `buff`. Alla riga 31 `buff` è vuoto si passa all'iterazione successiva, altrimenti alla riga 32 si incrementa `nDati` di 1, poi si passa all'iterazione successiva. Alla seconda iterazione `hnd` punta alla seconda riga del file, alla terza iterazione punterà alla terza, e così via. Uscendo dal loop, `nDati` contiene il numero di righe non vuote del file, che viene scritto sul monitor alla riga 34.

Alla riga 35 il puntatore `hnd` viene riposizionato all'inizio del file da leggere per la lettura definitiva, effettuata dal loop delle righe 40-48. Alla riga 37 viene assegnata la memoria necessaria per la lettura del file: il puntatore `xx` punterà ad una zona di memoria contenente `nDati` numeri in doppia precisione. Alla riga 39 la variabile `i` viene inizializzata a 0.

Il loop infinito dichiarato alla riga 40 effettua la lettura dei dati del file. La riga 42 legge una riga (fino a 80 caratteri) e la copia in `risposta`. La riga 43 fa uscire dal loop se tutte le righe del file sono state lette. Alla riga 46 la funzione `atof()`, (ASCII to float) dichiarata in `stdlib.h`, converte la sequenza di caratteri ASCII che rappresentano il dato nella stringa `buff` nella codifica binaria del corrispondente numero in doppia precisione. Ad ogni iterazione il numero appena letto viene copiato nell'elemento del vettore in doppia precisione `xx[i]`, poi `i` viene incrementato di 1. Alla riga 49 il file di dati viene chiuso, da questo momento non è più in interazione con il nostro programma.

Alla riga 51 il programma chiede il nome del file su cui scrivere quanto verrà elaborato. Battiamo il nome sulla tastiera, che viene letta dall'`fgets()` della riga 52. Alla riga 53 viene cancellata l'eventuale andata a capo alla fine del nome del file da scrivere. Il file stesso viene aperto, in modo scrittura ("`w`" per *write*) alla riga 54. A questo punto `ptr` punta all'inizio del nuovo file. Visto che il nuovo file viene creato (o, se esisteva già un file con quel nome, viene cancellato e sostituito da uno nuovo vuoto), è praticamente impossibile che la funzione `fopen()` dia errore. Nell'improbabile caso che accada, questo viene gestito dal codice tra le parentesi a graffa delle righe 55 e 58.



```

giovanni@moruzzi3: ~/xcpp/algoritmi/lavoro
File Edit View Search Terminal Help
moruzzi3:giovanni:~/xcpp/algoritmi/lavoro>scrivifile
Scrivi il nome del file da leggere:
dati.txt
Nel file ci sono 7 dati
Scrivi il nome del file da scrivere:
uscita.txt
moruzzi3:giovanni:~/xcpp/algoritmi/lavoro>

```

Figura 1.2 Esecuzione del programma 1.5 su un terminale.

incontra un carattere `%`. Nel nostro caso il primo `%15.51f` dice di riservare 15 caratteri per scrivere un numero in doppia precisione (`1f` sta per *long float*), che comprendono sia l'even-

Una volta aperto il file, parte il loop di scrittura delle righe 59-62. Su ogni del file, l'istruzione alla riga 61 scrive il numero letto nella riga corrispondente del file di dati, più la sua radice quadrata. Gli argomenti della funzione `fprintf()` sono il puntatore `hnd` al file su cui scrivere, una stringa di formattazione, e ulteriori argomenti. La stringa di formattazione viene copiata tale e quale in uscita finché non si

tuale segno meno che il punto decimale. Il 5 dopo il punto dice di stampare il numero con 5 cifre frazionarie (cifre dopo il punto nella notazione anglosassone, dopo la virgola nella notazione europea continentale). Il numero da scrivere è la prima variabile dopo la stringa di formattazione, cioè `xx[i]`. Poi viene stampato lo spazio bianco, poi al secondo `%15.5lf`, che viene applicato alla seconda variabile dopo la stringa di formattazione. Questa è `sqrt(xx[i])`, ovvero la radice quadrata di `xx[i]`. La funzione radice quadrata, `sqrt()` è definita in `math.h`. Finalmente il codice `\n` dice di andare a capo, terminando la riga. La chiamata del programma del listato 1.5, con l'interazione utente/monitor, è mostrata in Fig. 1.2, mentre il file prodotto in uscita dal nostro programma è mostrato del listato 1.6.

1	15.50000	3.93700
2	26.37000	5.13517
3	6.30000	2.50998
4	36.00000	6.00000
5	55.32700	7.43821
6	100.00000	10.00000
7	15.00000	3.87298

Listato 1.6 File in uscita prodotto dal listato 1.5 dopo lettura del listato 1.4

Nota: In realtà, in questo contesto, l'istruzione `fieldcpy` alle righe 30 e 44 del listato 1.5 è inutile, le righe 31 e 45 avrebbero potuto poi essere sostituite dall'istruzione

```
if (risposta[0]==0) continue;
```

e il programma avrebbe funzionato nello stesso modo. La `fieldcpy()` è stata inserita perché sarebbe stata utile in una piccola modifica al programma per leggere un file di dati le cui righe contengano due o più dati l'una.

Capitolo 2

Grafica sotto il protocollo X Window

2.1 Introduzione alla grafica sotto X Window

Come primo esempio di grafica sotto il protocollo *X Window* (senza *s* finale!), detto anche protocollo *X11* o semplicemente *X*, scriviamo un programma che disegni una simulazione dei colori dell'arcobaleno sul monitor, come in Fig. 2.1. Il file `xcolori.cc` è mostrato nel listato 2.1, che discuteremo qui sotto.

Listato 2.1 Un primo programma di grafica

```
1 // ..... #include
2 #include <math.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <X11/Xlib.h>
7
8 #include <corso.h>
9
10 main()
11 {
12     int i, j;
13     int screen_num;
14     char *display_name;
15     XColor color;
16     XEvent ev;
17     Display *display;
18     Window root_window;
19     Window win;
20     Colormap colormap;
21     GC gc;
22
23     display_name=getenv("DISPLAY");
24     display=XOpenDisplay(display_name);
25     screen_num=DefaultScreen(display);
26     root_window=RootWindow(display, screen_num);
27     win=XCreateSimpleWindow(display, root_window, 10, 10, 750, 200, 2,
28                               BlackPixel(display, screen_num),
29                               WhitePixel(display, screen_num));
```

```

30 XMapWindow(display , win);
31 XFlush(display);
32 unsigned long valuemask = 0;
33 XGCValues values; //          initial values for the GC.
34 values.line_width = 1; //      line width for the GC.
35 values.line_style = LineSolid; // style for lines drawing and
36 values.cap_style = CapButt; //   style of the line's edge and
37 values.join_style = JoinBevel; // joined lines.
38 gc=XCreateGC(display , win , valuemask ,&values);
39
40 colormap=DefaultColormap(display , DefaultScreen(display));
41 for (i=0;i<750;i++)
42 {
43     j=i%250;
44     if (i<250)
45     {
46         color.red=65535;
47         color.green=(j*65535)/250;
48         color.blue=0;
49     }
50     else if (i<500)
51     {
52         color.red=65535-(65535*j)/250;
53         color.green=65535;
54         color.blue=(j*65535)/250;
55     }
56     else
57     {
58         color.red=0;
59         color.green=65535-(65535*j)/250;
60         color.blue=65535;
61         if (j>125) color.red=(65535*(j-125)*2)/250;
62     }
63     XAllocColor(display , colormap ,&color);
64     XSetForeground(display , gc , color.pixel);
65     XDrawLine(display , win , gc , i , 0 , i , 200);
66 }
67 XSelectInput(display , win , KeyPressMask| ButtonPressMask);
68 XSync(display , FALSE);
69 for (;;)
70 {
71     XNextEvent(display ,&ev);
72     if (ev.type==KeyPress) break;
73     else if (ev.type==ButtonPress) break;
74 }
75 }
76
77 // _____

```

Alla riga 6 includiamo lo header per la libreria *Xlib.h*, che contiene le funzioni grafiche che useremo. Il `main()` inizia con le dichiarazioni delle variabili, e le righe 15-21 dichiarano variabili di tipo struttura, con le strutture corrispondenti dichiarate nel file *Xlib.h*, normalmente contenuto nella directory `/usr/include/X11`. La struttura `XColor` è listata in 2.2

I suoi elementi sono un intero non segnato lungo (da 0 a $4294967295=2^{32} - 1$) `pixel`, che contiene la codifica numerica del colore da usare nella scrittura sul monitor, tre interi non segnati brevi (da 0 a 65535) `red`, `green` e `blue` per le intensità dei rispettivi colori. Questa struttura verrà usata dalla funzione `XAllocColor()` per costruire la codifica del colore desiderato.

Alla riga 16 viene dichiarato che la variabile `ev` è una unione del tipo `XEvent`, anche questa definita in `Xlib.h`. Questa unione è un'unione di strutture relative ad eventi (premuta o rilascio di un tasto della tastiera, premuta o rilascio di un bottone del mouse, movimenti del mouse . . .) che `X11` deve intercettare per interagire con l'utente.

Queste strutture hanno dimensioni diverse a seconda del tipo di evento, e vengono messe in una unione, a cui viene assegnato lo spazio di memoria corrispondente alla struttura più grande, in modo che `Xlib` possa sempre usare lo stesso indirizzo di memoria per contenere la struttura dell'evento. Nel listato 2.2 a noi interesserà solo la struttura del tipo `KeyPress` (premuta della tastiera). Alla riga 17 `display` è definita come l'indirizzo di una struttura di tipo `Display` che contiene tutte le informazioni sulla gestione del monitor dell'utente. Queste informazioni saranno usate da `X11` per scrivere e disegnare sul monitor stesso.

In `Xlib.h` (in realtà in `X.h`, che è incluso da `Xlib.h`) `Window` è definito come un `unsigned long`, cioè un numero intero maggiore o uguale a zero di almeno 32 bit, che serve a identificare una *finestra*. Nel sistema `X Window` ogni finestra (*finestra figlia*) è contenuta in un'altra finestra, detta la *finestra madre*. In questo modo si crea una gerarchia (o albero genealogico) di finestre. La `root_window` è la radice di questa gerarchia (di quest'albero genealogico), che corrisponde al monitor del nostro computer. All'interno della `root window` possono poi trovarsi altre finestre, come la finestra la `win`, in cui lavoreremo.

`Colormap` è definito come un intero non segnato a 32 bit, mentre `GC` (graphic context) è una struttura che contiene informazioni su come gestire la grafica. L'aspetto di qualunque cosa venga scritto o disegnato dal programma è controllato dal contesto grafico. La struttura `GC` dipende dalla particolare implementazione del sistema operativo, e non è accessibile direttamente dall'utente. Esiste però una struttura `XGCValues` (listato 2.3), che è accessibile e vedremo più sotto.

Alla riga 23 la funzione `getenv()` copia sulla variabile `display_name` il codice della variabile ambientale "DISPLAY", alla riga 24 la funzione `XOpenDisplay()` connette il nostro programma al monitor usando la variabile ambientale appena copiata da `getenv`, e copia sulla struttura puntata da `display` tutta l'informazione necessaria per la connessione. È importante notare che `display_name` avrebbe potuto riferirsi al monitor di un altro computer, o a un altro terminale, comunque connesso in rete con il computer su cui gira il nostro programma. Nel caso di un unico computer con un unico monitor `display_name` vale `:0.0`.

Alla riga 25 la macro `DefaultScreen()` fornisce il numero `screen_num` che identifica lo schermo usato dal server.

Alla riga 26 la funzione `RootWindow()` copia in `root_window` il numero identificativo del monitor su cui lavoriamo.

Finalmente, alla riga 27, la funzione `XCreateSimpleWindow()` crea la finestra `win` in cui lavoreremo e ce ne fornisce il numero identificativo. A `XCreateSimpleWindow()` dobbiamo

```

1 typedef struct {
2     unsigned long pixel;
3     unsigned short red, green, blue;
4     char flags;
5     /* do_red, do_green, do_blue */
6     char pad;
7 } XColor;

```

Listato 2.2 Struttura `XColor`

passare come argomenti l'indirizzo della struttura `display` usata, l'identificazione della root window, le coordinate x e y misurate in pixel (nel nostro caso 10 e 10) dell'angolo superiore sinistro della finestra da creare (questi numeri vanno forniti, ma verranno tranquillamente ignorati nell'esecuzione del programma!), la larghezza e l'altezza in pixel della finestra che vogliamo (in questo caso 750 e 200), la larghezza del margine della finestra (2 pixel), ed infine due numeri di tipo `unsigned long` che corrispondono alla codifica dei colori nero e bianco sullo schermo che stiamo usando. Queste due codifiche ci sono fornite dalle funzioni `BlackPixel()` e `WhitePixel()`, cui vanno fornite le informazioni su `display` e numero dello schermo.

Listato 2.3 La struttura `XGCValues`

```

1 /*
2  * Data structure for setting graphics context.
3  */
4 typedef struct {
5     int function; /* logical operation */
6     unsigned long plane_mask; /* plane mask */
7     unsigned long foreground; /* foreground pixel */
8     unsigned long background; /* background pixel */
9     int line_width; /* line width */
10    int line_style; /* LineSolid, LineOnOffDash, LineDoubleDash */
11    int cap_style; /* CapNotLast, CapButt,
12                  CapRound, CapProjecting */
13    int join_style; /* JoinMiter, JoinRound, JoinBevel */
14    int fill_style; /* FillSolid, FillTiled, FillStippled, FillOpaqueStippled */
15    int fill_rule; /* EvenOddRule, WindingRule */
16    int arc_mode; /* ArcChord, ArcPieSlice */
17    Pixmap tile; /* tile pixmap for tiling operations */
18    Pixmap stipple; /* stipple 1 plane pixmap for stippling */
19    int ts_x_origin; /* offset for tile or stipple operations */
20    int ts_y_origin;
21    Font font; /* default text font for text operations */
22    int subwindow_mode; /* ClipByChildren, IncludeInferiors */
23    Bool graphics_exposures; /* boolean, should exposures be generated */
24    int clip_x_origin; /* origin for clipping */
25    int clip_y_origin;
26    Pixmap clip_mask; /* bitmap clipping; other calls for rects */
27    int dash_offset; /* patterned/dashed line information */
28    char dashes;
29 } XGCValues;

```

Una finestra appena creata non compare automaticamente sul monitor, ma viene disegnata dalla sequenza di funzioni `XMapWindow()` alla riga 30 e `XFlush()` alla riga 31.

Alla riga 33 viene dichiarata una struttura `XGCValues`, mostrata nel listato 2.3, i cui elementi servono a controllare lo stile di scrittura e disegno. I valori di alcuni elementi della struttura sono fissati nelle righe 34-37. Per esempio, alla riga 34 viene fissato che le linee disegnate devono avere larghezza 1 pixel, mentre `LineSolid`, `CapButt` e `JoinBevel` sono costanti numeriche, corrispondenti a codifiche, definite in `Xlib.h`.

Alla riga 38 la funzione `XCreateGC()` crea il contesto grafico con i valori che noi abbiamo appena fissato. Alla riga 40 la funzione `DefaultColorMap()` ci dà il numero di codice della mappa dei colori che verrà usata sul nostro monitor.

Finalmente, il loop delle righe 41-66 disegna il nostro arcobaleno sul monitor, che apparirà come mostrato in Fig. 2.1. L'arcobaleno è costituito da 750 linee verticali, ognuna larga 1 pixel e di colore leggermente diverso dalla precedente. Come avevamo detto sopra, la variabile `color` è una struttura del tipo `XColor`, i diversi colori vengono dati attribuendo diversi valori ai suoi elementi `color.red`, `color.green` e `color.blue`. Nel loop la variabile i assume i valori da 0 a 749 compresi, mentre alla riga 43 alla variabile j viene dato il valore di $i \bmod 250$. In questo modo, posto $M = 65535$ (intensità massima), abbiamo per le componenti di `color`



Figura 2.1 Output sul monitor del listato 2.1.

$$\text{color.red} = \begin{cases} M & \text{se } 0 \leq i < 250 \\ M \left(1 - \frac{i - 250}{250}\right) & \text{se } 250 \leq i < 500 \\ 0 & \text{se } 500 \leq i < 625 \\ M \frac{i - 625}{125} & \text{se } 625 \leq i < 750 \end{cases}$$

$$\text{color.green} = \begin{cases} M \frac{i}{250} & \text{se } 0 \leq i < 250 \\ M & \text{se } 250 \leq i < 500 \\ 0 & \text{se } 500 \leq i < 750 \end{cases}$$

$$\text{color.blue} = \begin{cases} 0 & \text{se } i < 250 \\ M \frac{i - 250}{250} & \text{se } 250 \leq i < 500 \\ M & \text{se } 500 \leq i < 750 \end{cases}$$

La presenza di una componente di rosso anche per $i > 625$ è dovuta a un fenomeno “fisiologico”: i fotoni violetti hanno energia sufficiente per eccitare anche i coni del nostro occhio sensibili al rosso.

Alla riga 63 la funzione `XAllocColor()`, partendo dai valori di `color.red`, `color.green` e `color.blue` calcola la codifica numerica del nostro colore, che viene copiata in `color.pixel`. Alla riga 64 la funzione `XSetForeground()` dice che tutto quello che verrà scritto o disegnato da quel momento in poi avrà il colore codificato in `color.pixel`. Finalmente alla riga 65 la funzione `XDrawLine()` disegna la riga sulla finestra `win` del nostro monitor (`display`). Le ultime 4 variabile di `XDrawLine()` sono le coordinate x e y (in pixel) del punto di partenza del segmento e quelle del punto di arrivo. Trattandosi di un segmento verticale le due coordinate x sono uguali, e pari alla variabile i , la y va da 0 a 200 per coprire tutta la finestra in verticale.

Alla riga 67 la funzione `XSelectInput` sceglie a quali “eventi” il programma deve essere sensibile quando è attiva la finestra `win`. Gli eventi hanno una codifica numerica definita in *Xlib.h*, il valore `KeyPressMask` corrisponde alla premuta di un tasto della tastiera, `ButtonPressMask` alla premuta di un bottone del mouse. Passando alla funzione *lor* dei due valori `KeyPressMask|ButtonPressMask`, il programma sarà sensibile ad entrambi gli eventi.

Alla riga 68 la funzione `XSync()` sincronizza la scrittura sul nostro monitor con lo svolgimento del programma. Se il secondo argomento viene posto uguale a `TRUE` la catena degli eventi precedenti viene cancellata, se viene posto uguale a `FALSE` gli eventi restano in coda fino alla loro lettura.

Il loop infinito alle righe 69-74 serve a mantenere il disegno sul monitor prolungando l'esecuzione del programma. A ogni iterazione la funzione `XNextEvent()` controlla se è successo qualcosa: se sì i dati relativi all'evento vengono copiati nella struttura `ev`. Se l'evento è una premuta della tastiera (riga 72), o la premuta di un bottone del mouse (riga 73), il loop infinito viene interrotto ed il programma termina, causando la scomparsa della finestra con il nostro arcobaleno.

2.2 Definiamo la struttura `XVideoData`

Per semplificarci la vita nel seguito, definiamo la struttura listata in 2.4. Questa contiene tutto quello che potrà servirci per il resto del corso, per quel che riguarda la gestione della grafica.

Listato 2.4 La struttura `XVideoData`

```

1 typedef struct // ..... XVideoData
2 {
3     Display *display;
4     Window *win;
5     Window root_window;
6     XEvent ev;
7     XFontStruct **FontInfo;
8     Pixmap *backpix;
9     Cursor cursor;
10    int screen_num;
11    int depth;
12    int *height;int *width;
13    int *static_x;int *static_y;
14    int *UsedWin;
15    int nWin;
16    // .....
17    int left_alt_pressed;int right_alt_pressed;
18    int left_ctrl_pressed;int right_ctrl_pressed;

```

```

19  int Num_Lock_pressed;
20  int left_shift_pressed;int right_shift_pressed;
21  int *dx;
22  int *dy;
23  int *hx;
24  int *hy;
25  int *qx;
26  int *qy;
27  int *yOffs;
28  int nFonts;
29  unsigned long DefForeGround;unsigned long DefBackGround;
30  unsigned long ForeGround;unsigned long BackGround;
31  int PaintBackground;
32  char *display_name;
33  char **FontName;
34  Colormap colormap;
35  GC gc;
36  unsigned long black;
37  unsigned long brown;
38  unsigned long white;
39  unsigned long red [4];
40  unsigned long green [4];
41  unsigned long blue [4];
42  unsigned long cyan [4];
43  unsigned long yellow [4];
44  unsigned long magenta [4];
45  unsigned long gray [100];
46 } XVideoData;

```

Le prime definizioni di variabili sono analoghe a quelle all'inizio del listato 2.1. Ma alla riga 4, anziché una finestra, definiamo un vettore di finestre, in modo che il nostro programma possa utilizzarne, contemporaneamente, più di una.

Poi, alla riga 7, definiamo un puntatore a puntatori a strutture del tipo `XFontStruct` (al solito struttura definita in `Xlib.h`). Una di queste strutture contiene informazioni sui caratteri da usare scrivendo nella nostra finestra grafica. Ci servirà più di una struttura per poter usare caratteri di dimensioni diverse, e, per esempio, lettere greche.

Alla riga 8 viene definito un vettore di strutture del tipo `Pixmap`. La struttura di una pixmap è del tutto equivalente alla struttura di una finestra, e, su di essa, le funzioni grafiche possono scrivere e disegnare esattamente come su una finestra. Solo che la pixmap è inserita nella memoria non grafica del computer, e non si vede direttamente. Serve a due scopi: i) è utile costruire il disegno prima sulla pixmap, utilizzando il tempo necessario per svolgere eventuali calcoli, poi, una volta che il disegno è finito, copiarlo sulla finestra; ii) se ad un certo istante un'eventuale altra finestra presente sul monitor si sovrappone alla finestra su cui stiamo lavorando, il contenuto della nostra finestra così coperto viene definitivamente cancellato. In questo caso la nostra pixmap serve da backup (da qui il nome `backpix` della nostra variabile), ed il disegno può essere ricopiato sulla finestra appena non sarà più nascosta.

Alla riga 12 i vettori `height` e `width` sono le altezze e le larghezze delle finestre che il nostro programma gestisce (eventualmente una sola finestra).

Alla riga 13 i vettori `static_x` e `static_y` sono le coordinate x e y dell'ultimo carattere che è stato scritto su ogni finestra, utilizzati per poter riprendere la scrittura dove era stata

interrotta, in un istante successivo.

Alla riga 14 gli elementi del vettore `UsedWin` valgono 0 (FALSE) se la finestra corrispondente non è usata, 1 (TRUE) se è usata. Infine, alla riga 15, `nWin` è il numero di finestre che possiamo usare.

Alle righe 17-20 vengono definite delle variabili che ci dicono se i tasti di controllo della tastiera (*Ctrl*, Maiuscole, *Alt*, ecc.) sono premuti o meno. Notare che molti tasti di controllo compaiono sia a destra che a sinistra della tastiera, e questi tasti sono considerati tasti diversi da *X Window*. Per esempio, la variabile `left_shift_pressed` varrà 1 se il tasto delle maiuscole a sinistra è premuto, 0 se non è premuto. Analogamente per gli altri tasti di controllo. Notare che, sulle tastiere attuali, il tasto *Alt* destro è chiamato *Alt Gr*.

Alle righe 21-27 sono definiti dei vettori di interi che conterranno le dimensioni x e y dei caratteri usati per la scrittura: `dx[i]` è la larghezza del carattere dello i -esimo set di caratteri, `dy[i]` la sua altezza, `hx[i]` e `qx[i]` sono rispettivamente metà e un quarto della sua larghezza, analogamente `hy[i]` e `qy[i]` sono rispettivamente metà e un quarto dell'altezza. Invece `yOffs[i]` è la distanza verticale tra il punto più alto di un carattere "ascendente", come la *b*, la *d* o la *t*, e il suo punto più basso, o il punto più basso di un carattere né ascendente né discendente, come la *a* o la *e*. Un carattere discendente, come la *g*, la *p* o la *q* scenderà ulteriormente.

Alle righe 29 e 30 vengono definite variabili che conterranno le codifiche numeriche dei colori corrispondenti allo sfondo di default, che normalmente porremo bianco, e al colore di disegno/scrittura di default, che normalmente porremo nero, oltre ai colori di sfondo e scrittura usati.

Alla riga 31 la variabile `PaintBackground`, se posta uguale a TRUE, indica che, nella scrittura, prima di disegnare un carattere, viene riempito con il colore dello sfondo il rettangolino che lo conterrà. Se è posta uguale a FALSE il carattere viene disegnato senza alterare lo sfondo preesistente.

Alle righe 36-45 vengono definite variabili per contenere le codifiche numeriche dei colori più usati. Per ognuno dei sei colori definiti alle righe 39-44 vengono date 4 gradazioni possibili, dalla più chiara (per esempio `red[0]`) alla più scura (per esempio `red[3]`). Alla riga 45 per il grigio sono date 100 gradazioni possibili, da `gray[0]` (nero) a `gray[99]` (bianco).

La funzione `StartXWindow()`, definita nel listato 3.2, dichiara la variabile globale `xvd` come struttura di tipo `XVideodata`, e ne definisce i valori degli elementi. La funzione `StartXWindow()` deve essere chiamata all'inizio del `main()` di un programma che usa la grafica.

2.3 Semplice animazione

Come esempio scriviamo un programma di semplice animazione, `bruco.cc`, listato in 2.5. A differenza da listato 2.1 qui non abbiamo bisogno di includere `Xlib.h`, perché tutto quanto serve per la grafica è già incluso alle righe 1-4 del file `corso.h` (listato 3.1), a sua volta incluso alla riga 7 di `bruco.cc`. Il programma farà apparire un bruco che striscia nella nostra finestra.

Alla riga 9 viene dichiarata come variabile globale (siamo al di fuori della coppia di parentesi a graffa che delimitano il codice del `main`) esterna la struttura `xvd`, di tipo `XVideodata`. Questo significa che quando il nostro programma sarà compilato, in fase di *linking*, ogni volta che verrà trovato un riferimento ad un elemento di `xvd` il valore verrà cercato nella struttura dichiarata

come globale nel file `StartXWindow` (listato 3.2). Tra le dichiarazioni delle variabili, sono da notare alla riga 14 le variabili `musec1` e `musec2`, di tipo `clock_t` (in realtà equivalenti a `unsigned long`, in cui copieremo dei valori in microsecondi, e, alla riga 16, le strutture `ts1` e `ts2`, di tipo

Listato 2.5 Un semplice programma di animazione: `bruco.cc`

```

1 // ..... #include
2 #include <math.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <time.h>
7 #include <corso.h>
8
9 extern XVideoData xvd;
10
11 main()
12 {
13     int dalpha,dx,dy,i,ii, interval,ix,iy,j,key;
14     clock_t musec1,musec2;
15     char titolo[20];
16     timespec ts1,ts2;
17
18     // .....
19     ts1.tv_sec=0;
20     interval=30000000;
21     sprintf(titulo,"Bruco");
22     // ..... apri la finestra
23     StartXWindow(800,150,0,0,titulo,1);
24     // ..... scegli l'input
25     XSelectInput(xvd.display,xvd.win[0],KeyPressMask|KeyReleaseMask|ExposureMask);
26     // ..... loop
27     ii=0;
28     XSync(xvd.display,TRUE);
29     for (;;)ii++
30     {
31         XSync(xvd.display,FALSE);
32         musec1=clock();
33         i=ii%xvd.width[0];
34         dx=21+int(4.0*sin(0.5*double(i)));
35         dy=21-int(4.0*sin(0.5*double(i)));
36         dalpha=int(2560.0*(1.0+sin(0.4*double(i))));
37         // ..... pulisci la finestra
38         XSetForeground(xvd.display,xvd.gc,xvd.white);
39         XFillRectangle(xvd.display,xvd.backpix[0],xvd.gc,0,0,xvd.width[0],
40                       xvd.height[0]);
41         // ..... disegna lo sfondo
42         XSetForeground(xvd.display,xvd.gc,xvd.brown);
43         XFillRectangle(xvd.display,xvd.backpix[0],xvd.gc,0,135,xvd.width[0],20);
44         XSetForeground(xvd.display,xvd.gc,xvd.black);
45         // ..... scrivi l'intervallo
46         XPrintf(0,350,0,1,TRUE,"Intervallo: %d\n",interval);
47         // .....
48         XSetForeground(xvd.display,xvd.gc,xvd.green[2]);
49         ix=ii-dx;

```

```

50   for (j=0;j<6;j++)
51   {
52       ix=(ix+(dx-2))%xvd.width[0];
53       iy=110+int(5.0*sin(0.5*double(ix)));
54       XFillArc(xvd.display,xvd.backpix[0],xvd.gc,ix,iy,dx,dy,0,23040);
55   }
56   XSetForeground(xvd.display,xvd.gc,xvd.white);
57   XFillArc(xvd.display,xvd.backpix[0],xvd.gc,ix,iy,dx,dy,-dalpha,dalpha);
58   XSetForeground(xvd.display,xvd.gc,xvd.black);
59   XFillArc(xvd.display,xvd.backpix[0],xvd.gc,ix+12,iy+4,4,4,0,23040);
60   XDrawLine(xvd.display,xvd.backpix[0],xvd.gc,ix+10,iy,ix+20,iy-20);
61   XDrawLine(xvd.display,xvd.backpix[0],xvd.gc,ix+10,iy,ix+30,iy-20);
62   XRedraw();
63   musec2=clock();
64   ts1.tv_nsec=interval-(musec2-musec1)*1000;
65   if (ts1.tv_nsec>0) nanosleep(&ts1,&ts2);
66   if (XCheckTypedWindowEvent(xvd.display,xvd.win[0],KeyPress,&xvd.ev))
67   {
68       key=XReadAfter(0);
69       if (key==-1) continue;
70       if (key=='+')
71       {
72           interval=int(double(interval)/1.1);
73           continue;
74       }
75       if (key=='-')
76       {
77           interval=int(double(interval)*1.1);
78           continue;
79       }
80       break;
81   }
82   else if (XCheckTypedWindowEvent(xvd.display,xvd.win[0],KeyRelease,&xvd.ev))
83       key=XReadAfter(0);
84   }
85   // ..... aspetta
86   XWait(0);
87   // ..... chiudi la finestra
88   CloseXWindow();
89 }

```

timespec. Questa struttura a due elementi, **tv_sec** che corrisponde a un numero di secondi, e **tv_nsec** che corrisponde a un numero di nanosecondi. Le strutture **ts1** e **ts2** ci serviranno per regolare la cadenza dei “fotogrammi” nella nostra animazione.

Alla riga 19 l’elemento **ts1.tv_sec** viene posto uguale a 0. Alla riga 20 la variabile **interval** viene posta uguale a 30 000 000 nanosecondi, cioè a 0.03 secondi, che sarà l’intervallo iniziale tra un “fotogramma” ed il fotogramma successivo nella nostra animazione.

Alla riga 21 viene scritta la parola *Bruco* sulla stringa di caratteri **titolo**. Alla riga 23 viene chiamata la funzione

```

void StartXWindow(int width, int height, int x, int y, char *WindowName,
                  int nWin)

```


passandole i valori 800 pixel per la larghezza della finestra, 150 pixel per la sua altezza, 0 e 0 per la x e la y del suo angolo superiore sinistro (ma abbiamo già detto che questi valori sono irrilevanti). Il titolo della finestra, `Windowname`, sarà la parola *Bruco* che abbiamo copiato in `titolo`. Infine chiediamo l'uso di una sola finestra (`nWin=1`). Una volta chiamata, `StartXWindow()` genererà una finestra delle dimensioni desiderate con fondo inizialmente bianco, e tutti i valori che ci interessano per la grafica, una volta calcolati, saranno immagazzinati negli della struttura globale `xvd`.

Alla riga 25 chiediamo che il nostro programma, quando la nostra finestra `xvd.win[0]` è attiva, sia sensibile agli eventi di premuta di un bottone della tastiera (`KeyPressMask`), di rilascio di un bottone della tastiera (`KeyReleaseMask`) e di *Exposure*, cioè di sovrapposizione di finestre sul monitor.

Alla riga 27 azzeriamo la variabile `ii`, e alla riga 28 sincronizziamo una prima volta la grafica con il calcolo, cancellando, prima che inizi il nostro loop, tutti gli eventi che per qualunque motivo si trovavano nella coda di *X11*. Questo si fa ponendo la seconda variabile di `XSync()` uguale a `TRUE`.

Dalla riga 29 alla riga 84 abbiamo un loop infinito in cui, ad ogni iterazione, viene incrementata di 1 la variabile `ii`. La riga 31 serve a tenere sincronizzate grafica e svolgimento dei calcoli ad ogni iterazione del loop, senza cancellare la coda degli eventi. Alla riga 32 la funzione `clock()` copia il numero di microsecondi trascorsi dall'inizio del programma in `mussec1`. Questo ci servirà per mantenere l'immagine (verrà generato un fotogramma per iterazione del loop) sullo schermo il tempo desiderato.

Alla riga 33 `i` viene posta uguale a `ii` modulo la larghezza della finestra (`xvd.width[0]`), in modo che la `i` possa essere usata per determinare la posizione x del bruco senza mai uscire dalla finestra.

Alle righe 34-36 vengono definite delle variabili, che cambiano a ogni iterazione e serviranno più sotto per disegnare il bruco. La variabile `dx` è una larghezza, in pixel, il cui valore oscilla attorno a 21 (da 17 a 25), mentre `dy` è un'altezza, sempre con valore oscillante tra 17 e 25, ma in controfase rispetto a `dx`.

La variabile `dalpha` alla riga 36 corrisponde ad un angolo. Gli angoli, in *X Window*, sono misurati in numeri interi ottenuti moltiplicando il valore in gradi per 64. Questa unità vale così un po' meno di un primo (15/16 di primo, per la precisione). In queste unità un angolo giro vale $360 \times 64 = 23040$. Il motivo per cui gli angoli non vengono misurati in primi è che, su un calcolatore binario, la moltiplicazione, o la divisione, per 64 (potenza di 2) è più rapida di quella per 60. Poiché il valore 2560 corrisponde a 40° , l'angolo `dalpha` oscilla tra 0° e $+80^\circ$.

Alla riga 38 la funzione `XSetForeground()` pone uguale al bianco il colore di scrittura. Alla riga 39 la funzione `XFillRectangle()` colora uniformemente di bianco la pixmap di backup della nostra finestra, `xvd.backpix[0]`. Le ultime 4 variabili sono infatti le coordinate x e y , in pixel, dell'angolo superiore sinistro (x_1 e y_1) e dell'angolo inferiore destro (x_2 e y_2) del rettangolo da riempire. Ponendo $x_1 = y_1 = 0$, $x_2 = \text{xvd.width}[0]$ e $y_2 = \text{xvd.height}[0]$ si colora di bianco l'intera finestra.

Alla riga 42 si sceglie il marrone come colore di scrittura, e, alla riga 43 si riempie un rettangolo che servirà come terreno su cui camminerà il bruco. Alla riga 42 si sceglie il nero come colore di scrittura, e la funzione `XPrintf()`, descritta nel listato 3.4, scrive il valore della latenza dell'immagine, in nanosecondi, in alto al centro della pixmap.

Alla riga 48 viene scelto il colore verde `xvd.green[2]` per disegnare il bruco. Poi la variabile intera `ix` viene posta uguale alla `i` definita alla riga 33 meno la larghezza `dx`. Poi il loop alle righe 50-55 disegna i sei anelli del bruco (in realtà cinque più la testa). A ogni iterazione la variabile `ix` viene incrementata di $(dx - 2)$ pixel, e l'operazione di modulo evita che si esca dalla finestra. La variabile `dy` oscilla attorno a 100 pixel dall'alto della finestra con una semiampiezza di 5 pixel. Poi viene chiamata la funzione `XFillArc()` che disegna un arco (in realtà un settore) di ellisse circoscritto nel rettangolo con angolo superiore sinistro in (ix, iy) , larghezza `dx` e altezza `dy`. Il settore parte dall'angolo 0 in una normale rappresentazione trigonometrica ed è ampio 23040 unità, quindi 360° , e corrisponde ad un'ellisse completa. Come dice il nome della funzione, questo settore ellittico è riempito con il colore scelto.

Alla riga 56 si sceglie il colore bianco, e alla riga 57 `XFillArc()` disegna un settore di ellisse circoscritto dallo stesso rettangolo dell'ultimo ellisse disegnato, con angolo che parte da `-dalpha` ed è ampio `dalpha`. Questo corrisponde alla bocca del nostro bruco.

Alla riga 58 viene scelto il colore nero, e, nelle righe 59-61 vengono disegnati l'occhio e le antenne. Finalmente alla riga 62 la funzione `XRedraw()`, definita nel listato 3.6, copia la pixmap sulla finestra grafica visibile sul monitor. L'immagine di un fotogramma è mostrata in Fig. 2.2.

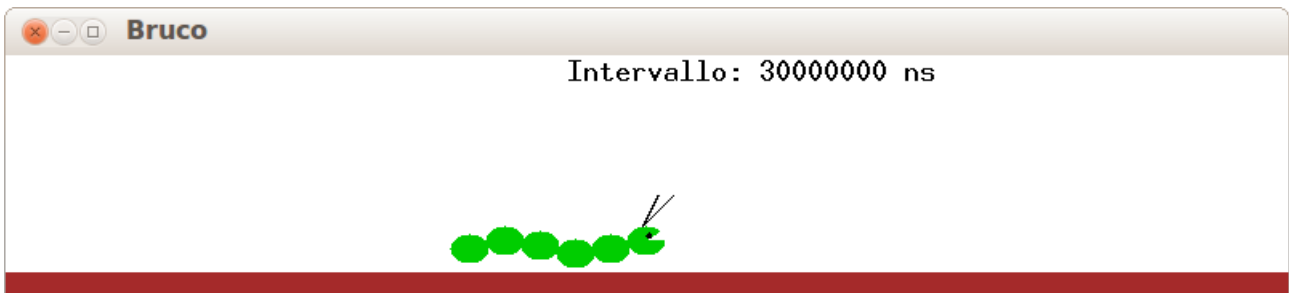


Figura 2.2 Un fotogramma dell'animazione generata dal listato 2.5.

Alla riga 63 il numero di microsecondi trascorso dall'inizio del programma viene copiato in `musec2`. Poiché vogliamo che un'immagine resti sul monitor un numero di nanosecondi pari a `interval`, alla riga 64 calcoliamo quanti nanosecondi dobbiamo ancora aspettare. Se questo numero è maggiore di zero, alla riga 65 chiamiamo la funzione `nanosleep()`, che blocca l'esecuzione del programma per il tempo fissato dalla struttura `ts1`. Se il "sonno" venisse interrotto per qualunque motivo, il tempo di sonno residuo verrebbe copiato nella struttura `ts2`, secondo argomento di `nanosleep()`.

Alla riga 66 la funzione `XCheckTypedWindowEvent()` accerta se, dalla sua ultima chiamata, sono stati registrati eventi del tipo `KeyPress`. In questo caso i dati dell'evento vengono cancellati dalla "coda" e copiati in `xvd.ev`, inoltre diventa attivo il codice delle righe 67-81. Altrimenti alla riga 82 si controlla se un bottone della tastiera, precedentemente premuto è stato rilasciato. Questo controllo è importante per distinguere, per esempio, le maiuscole dalle minuscole. Poi si passa all'iterazione successiva.

Se un tasto è stato premuto, alla riga 68 la funzione `XReadAfter()` legge da `xvd.ev` quale tasto è stato premuto. Questa funzione è listata in 3.5. Se è stato premuto un tasto di controllo (maiuscole, *Ctrl*, *Alt* ...) ne prende nota (righe 33-38 del listato 3.5) e scrive il valore `-1` in `key`. Se è stato premuto un tasto alfanumerico ne copia la codifica numerica sempre nella variabile `key` (riga 32 e righe 39-47 del listato 3.5). Tornando al listato 2.5, alle righe 70-74, se è stato premuto il tasto `+`, il tempo di latenza dell'immagine viene diminuito dividendolo per

1.1: in questo modo il bruco si muoverà più velocemente. Alla riga 73 il `continue` rimanda ad una nuova iterazione.

Alle righe 75-79, se è stato premuto il tasto `-` il tempo di latenza dell'immagine viene aumentato moltiplicandolo per 1.1: in questo modo il bruco si muoverà più lentamente. Alla riga 78 il `continue` rimanda ad una nuova iterazione.

Alla riga 80, se il tasto premuto non era né un `+` né un `-`, il comando `break` interrompe il loop, mandandoci alla riga 86.

Alle righe 82-83, se non era stato premuto un tasto la funzione `XCheckTypedWindowEvent()` controlla se un tasto è stato rilasciato. Se sì, viene chiamata di nuovo la funzione `XReadAfter()` per controllare se il tasto rilasciato era un tasto di controllo. In questo caso alle righe 22-27 del listato 3.5 si prende nota del fatto che il tasto di controllo in questione non è più premuto. Questo sempre per distinguere le maiuscole dalle minuscole alle eventuali successive chiamate di `XReadAfter()`. Poi si torna alla riga 31 e si inizia il disegno del fotogramma successivo.

Come avevamo detto più sopra, se viene premuto un tasto diverso da un tasto di controllo, un `+` o un `-`, il loop viene interrotto e, alla riga 86, viene chiamata la funzione `XWait()`, listata in 3.7. Questa entra nel loop infinito delle sue righe 22-30, che mantiene fermo sul monitor l'ultimo fotogramma disegnato. All'interno del loop infinito la funzione `XNextEvent()` aspetta che accada un evento. Se questo evento è una sovrapposizione di finestre (tipo *Expose*) alla riga 24 si provvede a ridisegnare il fotogramma, alla riga 25 si risincronizza e alla riga 26 si va ad aspettare un evento successivo. Se l'evento è stato invece una premuta di tastiera o di mouse, le righe 28 e 29 fanno uscire dal loop infinito. Nel nostro caso, in realtà, se premiamo un bottone del mouse non succederà niente perché, alla riga 25 del listato 2.5 non è stata chiesta la rilevazione di questo tipo di evento. Dal loop infinito si esce quindi solo premendo la tastiera.

Una volta terminata la attesa imposta dalla funzione `XWait()`, alla riga 88 la funzione `CloseXWindow()`, listata in 3.3, termina la sessione grafica.

Capitolo 3

Libreria

3.1 Il file delle dichiarazioni delle funzioni

Listato 3.1 File `corso.h`, contenente le dichiarazioni delle funzioni

```
1 #include <X11/Xlib.h>
2 #include <X11/keysymdef.h>
3 #include <X11/cursorfont.h>
4 #include <X11/Xutil.h>
5 #include <complex>
6 #define FALSE 0
7 #define TRUE 1
8
9 typedef std::complex<double> dComplex;
10
11 typedef struct // ..... XVideoData
12 {
13     Display *display;
14     Window *win;
15     Window root_window;
16     XEvent ev;
17     XFontStruct **FontInfo;
18     Pixmap *backpix;
19     Cursor cursor;
20     int screen_num;
21     int depth;
22     int *height;int *width;
23     int *static_x;int *static_y;
24     int *UsedWin;
25     int nWin;
26     // .....
27     int left_alt_pressed;int right_alt_pressed;
28     int left_ctrl_pressed;int right_ctrl_pressed;
29     int Num_Lock_pressed;
30     int left_shift_pressed;int right_shift_pressed;
31     int *dx;
32     int *dy;
33     int *hx;
34     int *hy;
```

```

35  int *qx;
36  int *qy;
37  int *yOffs;
38  int nFonts;
39  unsigned long DefForeGround; unsigned long DefBackGround;
40  unsigned long ForeGround; unsigned long BackGround;
41  int PaintBackground;
42  char *display_name;
43  char **FontName;
44  Colormap colormap;
45  GC gc;
46  unsigned long black;
47  unsigned long brown;
48  unsigned long white;
49  unsigned long red[4];
50  unsigned long green[4];
51  unsigned long blue[4];
52  unsigned long cyan[4];
53  unsigned long yellow[4];
54  unsigned long magenta[4];
55  unsigned long gray[100];
56 } XVideoData;
57
58 typedef struct // ..... XButton
59 {
60     int x1; int x2; int y1; int y2;
61     char *txt;
62 } XButton;
63
64 // =====
65
66 double bisection(double x1, double x2, double accuracy, double (*ptfunz)(double x));
67 void bracket(double *x1, double *x2, double (*ptfunz)(double x));
68 void brackmin(double *xa, double *xb, double *xc, double *fa, double *fb, double *fc,
69             double (*ptfunz)(double x));
70 void cfft(dComplex *z, int n, int dir);
71 void CloseXWindow();
72 void DblPtDnSrt(double *x, int *ind, int nt);
73 void dblsort(double *arr, int n);
74 void erwin(double *x, double xMax, double *eigv, int nEigen, int nStep,
75           double EigvStep, double tolerance,
76           void (*derivs)(double *x, double *y, double *dydx));
77 void fieldcpy(char *dest, char *source, int n);
78 void fft(double *z, int nm, int isign);
79 double golden(double ax, double bx, double cx, double (*ptfunz)(double x), double tol,
80             double *xmin);
81 void householder(double **a, double *d, double *e, int n);
82 void LeapFrog(double *t, double *x, double *dxdt, int nDim, double dt, int mode,
83             void (*derivs)(double *t, double *x, double *dxdt, double *d2xdt2));
84 void MatVectMult(double **a, double *b, double *c, int nDim);
85 void PlotPsi(double *psi, int nPoints, int IsEven, double Norm, int yEigv,
86             unsigned long color);
87 void RungeStep(double *x, double *y, int nVar, double step, int FirstMidLast,
88             void (*derivs)(double *x, double *y, double *dydx));

```

```

89 void RungKutCS(double *x,double *vstart,int nVar,double xMin, double xMax,
90               int nStep,double *xp,double **yp,int DeltaSave,int *nSaved,
91               void (*derivs)(double *x,double *, double *));
92 void StartXWindow(int width, int height, int x, int y,char *WindowName,int nWin);
93 double SymmWell(double *par,double *x,double **y,int nPoints,int DeltaSave,
94               double xMax,int iEv,double EigvStart,double EigvStep,double tolerance,
95               void (*derivs)(double *x,double *y,double *dydx));
96 void VectDnPtSort(double **vect,int *ind,int n,int nPos);
97 void VectPtSort(double **vect,int *ind,int n,int nPos);
98 void verlet(double *t,double *x,double *dxdt,double *d2xdt2,int nDim,double dt,
99             void (*accel)(double *t,double *x, double *dxdt,double *d2xdt2),
100            int mode);
101 void XButtonSize(char *str,int iFont,int *width,int *height);
102 void XCloseSubWin(int num);
103 int XCPrintf(int iWin,int x,int y,int iFont,int FillBack,const char *fmt,...);
104 int XCTitle(int x,int y,int iFont,unsigned long ColUp,unsigned long ColDn,
105            const char *fmt,...);
106 void XDelSubWin(int iWin);
107 int XGetAnsw(int x,int y,int iFont,int MaxLen,char *str,int iWin);
108 void XhScale(int iWin,int iFont,int FillBack,int xsize,int ox,int oy,
109            double firstpoint,double lastpoint,int numsmallticks);
110 int Xkbhit(int iWin);
111 int XMenu(int iFont,char *tit,char **ParName,char **value,int nPar,
112          int MaxLen,char **comm,int nComm,int x1,int y1);
113 void XPaintButton(XButton *bt,int iFont,int state,int iWin);
114 int XPermMenu(int iFont,char *tit,char **ParName,char **value,int nPar,
115             int MaxLen,char **comm,int nComm,int x1,int y1,
116             int *MouseX,int *MouseY,int mode,int iWin);
117 int XPrintf(int iWin,int x,int y,int iFont,int FillBack,const char *fmt,...);
118 int XReadAfter(int iWin);
119 void XReadButtKey(int iWin,int *RetKey,int *butt,int *x,int *y);
120 int XReadKey(int iWin);
121 void XRedraw();
122 int XSeqPrintf(int iFont,int FillBack,const char *fmt,...);
123 int XSubWin(int width, int height, int x, int y,char *WindowName);
124 int XTitle(int x,int y,int iFont,unsigned long ColUp, unsigned long ColDn,
125           const char *fmt,...);
126 int XUpCPrintf(int iWin,int x,int y,int iFont,int FillBack,const char *fmt,...);
127 int XUpPrintf(int iWin,int x,int y,int iFont,int FillBack,const char *fmt,...);
128 void XvScale(int iWin,int iFont,int FillBack,int ysize,int ox,int oy,
129            double firstpoint,double lastpoint,int numsmallticks);
130 void XWait(int iWin);
131 int XWarning(int x1,int y1,int iFont,const char *fmt,...);
132 int XWinPermMenu(int iFont,char *tit,char **ParName,char **value,int nPar,
133                int MaxLen,char **comm,int nComm,int mode);
134 int XWinPermMenu(int iFont,char *tit,int *ParFont,char **ParName,char **value,
135                int nPar,int MaxLen,char **comm,int nComm,int mode);

```

3.2 Le funzioni StartXWindow e CloseXWindow

Questa funzione deve essere chiamata all'inizio del `main()` di un programma che usa la grafica, e definisce praticamente tutte le variabili grafiche di cui ci sarà bisogno. Alla riga 12 la struttura `xvd`, di tipo `XVideoData` (definita a partire dalla riga 11 del listato 3.1), viene dichiarata come *variabile globale*. Questo le permette di essere usata dal `main()` e da tutte le altre funzioni che la dichiarino come `extern`.

Listato 3.2 La funzione `StartXWindow()`, che definisce le variabile grafiche

```

1  /*                                19.APR.2012                                startxwindow.cc
2                                                                                               by Giovanni Moruzzi
3      Started on 28.FEB.2005
4
5  */
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <unistd.h>
10 #include <string.h>
11
12 #include <corso.h>
13
14 XVideoData xvd;
15
16 /*
17  * function: create_simple_window. Creates a window with a white background
18  *           in the given size.
19  * input:    display, size of the window (in pixels), and location of the window
20  *           (in pixels).
21  * notes:    window is created with a black border, 2 pixels wide.
22  *           the window is automatically mapped after its creation.
23  */
24
25 void StartXWindow(int width, int height, int x, int y, char *WindowName,
26                  int nWin)
27 {
28     int i;
29     int win_border_width = 2;
30     char colstr[80];
31     XColor color;
32     Status rc;
33
34     // ..... allocate window memory
35     xvd.win=new Window[nWin];
36     xvd.backpix=new Pixmap [nWin];
37     xvd.height=new int [nWin];
38     xvd.width=new int [nWin];
39     xvd.static_x=new int [nWin];
40     xvd.static_y=new int [nWin];
41     xvd.UsedWin=new int [nWin];
42     xvd.nWin=nWin;
43     // .....
44     memset(xvd.UsedWin,0,nWin*sizeof(int));

```



```

45 xvd.UsedWin[0]=TRUE;
46 // .....
47 xvd.display_name=getenv("DISPLAY"); // address of the X display
48 if ((xvd.display=XOpenDisplay(xvd.display_name))==NULL)
49 {
50     printf("Cannot connect to X server '%s'\n",xvd.display_name);
51     exit(1);
52 }
53 xvd.screen_num=DefaultScreen(xvd.display);
54 xvd.root_window=RootWindow(xvd.display,xvd.screen_num);
55 xvd.win[0]=XCreateSimpleWindow(xvd.display,xvd.root_window,
56                               x,y,width,height,win_border_width,
57                               BlackPixel(xvd.display,xvd.screen_num),
58                               WhitePixel(xvd.display,xvd.screen_num));
59 xvd.width[0]=width;
60 xvd.height[0]=height;
61 XMapWindow(xvd.display,xvd.win[0]);
62 XFlush(xvd.display);
63
64 // handle of newly created GC.
65 unsigned long valuemask = 0; // which values in 'values' to check
66 //                               when creating the GC
67 XGCValues values; //                               initial values for the GC
68 unsigned int line_width=1; //                               line width for the GC
69 int line_style = LineSolid; //                               style for lines drawing and
70 int cap_style = CapButt; //                               style of the line's edge and
71 int join_style = JoinBevel; //                               joined lines
72
73 if ((xvd.gc=XCreateGC(xvd.display,xvd.win[0],valuemask,&values))<0)
74 {
75     printf("Cannot XCreateGC:\n");
76     exit(1);
77 }
78 // allocate foreground and background colors for this GC.
79 XSetForeground(xvd.display,xvd.gc,BlackPixel(xvd.display,xvd.screen_num));
80 XSetBackground(xvd.display,xvd.gc,WhitePixel(xvd.display,xvd.screen_num));
81 // define the style of lines that will be drawn using this GC.
82 XSetLineAttributes(xvd.display,xvd.gc,line_width,line_style,cap_style,
83                   join_style);
84 // ..... depth
85 xvd.depth=DefaultDepth(xvd.display,DefaultScreen(xvd.display));
86 // .....
87 // define the fill style for the GC. to be 'solid filling'.
88 XSetFillStyle(xvd.display,xvd.gc,FillSolid);
89 // ..... colormap
90 xvd.colormap=DefaultColormap(xvd.display,DefaultScreen(xvd.display));
91 // ..... allocate colors
92 rc=XAllocNamedColor(xvd.display,xvd.colormap,"black",&color,&color);
93 if (rc==0) printf("XAllocNamedColor failed to allocated 'black' color.\n");
94 xvd.black=color.pixel;
95 for (i=0;i<4;i++)
96 {
97     strcpy(colstr,"red"); sprintf(colstr+strlen(colstr),"%d",i+1);
98     rc=XAllocNamedColor(xvd.display,xvd.colormap,colstr,&color,&color);

```

```

99     if (rc==0) printf("Failed to allocate '%s1' color.\n", colstr);
100     xvd.red[i]=color.pixel;
101 }
102 for (i=0;i<4;i++)
103 {
104     strcpy(colstr,"green"); sprintf(colstr+strlen(colstr),"%d",i+1);
105     rc=XAllocNamedColor(xvd.display,xvd.colormap,colstr,&color,&color);
106     if (rc==0) printf("Failed to allocate '%s1' color.\n", colstr);
107     xvd.green[i]=color.pixel;
108 }
109 for (i=0;i<4;i++)
110 {
111     strcpy(colstr,"blue"); sprintf(colstr+strlen(colstr),"%d",i+1);
112     rc=XAllocNamedColor(xvd.display,xvd.colormap,colstr,&color,&color);
113     if (rc==0) printf("Failed to allocate '%s1' color.\n", colstr);
114     xvd.blue[i]=color.pixel;
115 }
116 for (i=0;i<4;i++)
117 {
118     strcpy(colstr,"cyan"); sprintf(colstr+strlen(colstr),"%d",i+1);
119     rc=XAllocNamedColor(xvd.display,xvd.colormap,colstr,&color,&color);
120     if (rc==0) printf("Failed to allocate '%s1' color.\n", colstr);
121     xvd.cyan[i]=color.pixel;
122 }
123 for (i=0;i<4;i++)
124 {
125     strcpy(colstr,"yellow"); sprintf(colstr+strlen(colstr),"%d",i+1);
126     rc=XAllocNamedColor(xvd.display,xvd.colormap,colstr,&color,&color);
127     if (rc==0) printf("Failed to allocate '%s1' color.\n", colstr);
128     xvd.yellow[i]=color.pixel;
129 }
130 for (i=0;i<4;i++)
131 {
132     strcpy(colstr,"magenta"); sprintf(colstr+strlen(colstr),"%d",i+1);
133     rc=XAllocNamedColor(xvd.display,xvd.colormap,colstr,&color,&color);
134     if (rc==0) printf("Failed to allocate '%s1' color.\n", colstr);
135     xvd.magenta[i]=color.pixel;
136 }
137 rc=XAllocNamedColor(xvd.display,xvd.colormap,"brown",&color,&color);
138 if (rc==0) printf("XAllocNamedColor failed to allocated 'brown' color.\n");
139 xvd.brown=color.pixel;
140 rc=XAllocNamedColor(xvd.display,xvd.colormap,"white",&color,&color);
141 if (rc==0) printf("XAllocNamedColor failed to allocated 'white' color.\n");
142 xvd.white=color.pixel;
143 for (i=0;i<100;i++)
144 {
145     strcpy(colstr,"gray"); sprintf(colstr+strlen(colstr),"%d",i+1);
146     rc=XAllocNamedColor(xvd.display,xvd.colormap,colstr,&color,&color);
147     if (rc==0) printf("Failed to allocate '%s1' color.\n", colstr);
148     xvd.gray[i]=color.pixel;
149 }
150 xvd.DefForeGround=xvd.ForeGround=BlackPixel(xvd.display,xvd.screen_num);
151 xvd.DefBackGround=xvd.BackGround=WhitePixel(xvd.display,xvd.screen_num);
152 xvd.PaintBackground=FALSE;

```

```

153 // ..... window name
154 XTextProperty window_name_property;
155 /* This variable will store the icon name property. */
156 // XTextProperty icon_name_property;
157 /* pointer to the size hints structure. */
158 XSizeHints* win_size_hints;
159 /* pointer to the WM hints structure. */
160 XWMHints* win_hints;
161 /* pixmap used to store the icon's image. */
162 //Pixmap icon_pixmap;
163 /* This window name and icon name strings. */
164 //char* window_name = "hello , world";
165 //char* icon_name = "small world";
166
167 if ((rc=XStringListToTextProperty(&WindowName,1,&window_name_property))==0)
168 {
169     printf("XStringListToTextProperty_ out_of_memory\n");
170     exit(1);
171 }
172 XSetWMName(xvd.display,xvd.win[0],&window_name_property);
173
174 // ..... load fonts
175 // xvd.nFonts=8;
176 xvd.nFonts=8;
177
178 xvd.FontInfo=new XFontStruct* [xvd.nFonts];
179 xvd.FontName=new char * [xvd.nFonts];
180 for (i=0;i<xvd.nFonts;i++) xvd.FontName[i]=new char [40];
181 xvd.dx=new int [xvd.nFonts];
182 xvd.dy=new int [xvd.nFonts];
183 xvd.hx=new int [xvd.nFonts];
184 xvd.hy=new int [xvd.nFonts];
185 xvd.qx=new int [xvd.nFonts];
186 xvd.qy=new int [xvd.nFonts];
187 xvd.yOffs=new int [xvd.nFonts];
188 strcpy(xvd.FontName[0],"7x14");
189 strcpy(xvd.FontName[1],"10x20");
190 strcpy(xvd.FontName[2],"12x24");
191 strcpy(xvd.FontName[3],"lucidasans-bold-24");
192 strcpy(xvd.FontName[4],"*symbol-medium-r-normal--14*");
193 strcpy(xvd.FontName[5],"*symbol-medium-r-normal--17*");
194 strcpy(xvd.FontName[6],"*symbol-medium-r-normal--24*");
195 strcpy(xvd.FontName[7],"*symbol-medium-r-normal--34*");
196 for(i=0;i<xvd.nFonts;i++)
197 { // .....
198     xvd.FontInfo[i]=XLoadQueryFont(xvd.display,xvd.FontName[i]);
199     if (!xvd.FontInfo[i])
200     {
201         printf("XLoadQueryFont: failed loading font '%s'\n",xvd.FontName[i]);
202         exit(1);
203     }
204     //xvd.dx[i]=8;
205     xvd.dx[i]=xvd.FontInfo[i]->max_bounds.width;
206     xvd.dy[i]=xvd.FontInfo[i]->ascent+xvd.FontInfo[i]->descent;

```

```

207     xvd.hx[i]=xvd.dx[i]/2;
208     xvd.hy[i]=xvd.dy[i]/2;
209     xvd.qx[i]=xvd.dx[i]/4;
210     xvd.qy[i]=xvd.dy[i]/4;
211     xvd.yOffs[i]=xvd.FontInfo[i]->ascent;
212 }
213 // ..... set active fonts
214 XSetFont(xvd.display,xvd.gc,xvd.FontInfo[1]->fid);
215 // .....
216 xvd.left_alt_pressed=FALSE;
217 xvd.right_alt_pressed=FALSE;
218 xvd.left_ctrl_pressed=FALSE;
219 xvd.right_ctrl_pressed=FALSE;
220 xvd.left_shift_pressed=FALSE;
221 xvd.right_shift_pressed=FALSE;
222 // ..... create drawing area
223 xvd.backpix[0]=XCreatePixmap(xvd.display,xvd.root_window,xvd.width[0],
224                             xvd.height[0],xvd.depth);
225 // .....
226 XSetForeground(xvd.display,xvd.gc,xvd.white);
227 XFillRectangle(xvd.display,xvd.backpix[0],xvd.gc,0,0,xvd.width[0],
228               xvd.height[0]);
229 XSetForeground(xvd.display,xvd.gc,xvd.black);
230 }

```

Listato 3.3 La funzione `CloseXwindow()`, che termina la sessione grafica

```

1 /*                                08.MAR.2005                                closerwindow.cc
2                                                                                               by Giovanni Moruzzi
3     Started on 08.MAR.2005
4
5 */
6
7 #include <stdio.h>
8 #include <stdlib.h> /* getenv(), etc. */
9 #include <unistd.h> /* sleep(), etc. */
10 #include <string.h>
11
12 #include <corso.h>
13
14 extern XVideoData xvd;
15
16 void CloseXWindow()
17 {
18     int i;
19     for (i=0;i<xvd.nWin;i++) if (xvd.UsedWin[i])
20     {
21         XFreePixmap(xvd.display,xvd.backpix[i]);
22         XDestroyWindow(xvd.display,xvd.win[i]);
23     }
24     XFreeGC(xvd.display,xvd.gc);
25     XCloseDisplay(xvd.display);
26 }

```

3.3 Funzioni grafiche

Listato 3.4 La funzione `XPrintf()`, che scrive sulle finestre grafiche

```

1  /*                                     27.JUN.2006                               xprintf.cc
2                                                                                               by Giovanni Moruzzi
3      Started on 03.MAR.2005
4
5  */
6
7  #include <stdarg.h>
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <string.h>
11
12 #include <corso.h>
13
14 extern XVideoData xvd;
15
16 int XPrintf(int iWin,int x,int y,int iFont,int FillBack,const char *fmt,...)
17 {
18     char buff[84];
19     va_list marker;
20     int len,width;
21
22     va_start(marker,fmt);
23     vsprintf(buff,fmt,marker);
24     va_end(marker);
25     len=strlen(buff);
26     width=XTextWidth(xvd.FontInfo[iFont],buff,len);
27     xvd.static_y[iWin]=y;
28     XSetFont(xvd.display,xvd.gc,xvd.FontInfo[iFont]->fid);
29     if (FillBack) XDrawImageString(xvd.display,xvd.backpix[iWin],xvd.gc,
30     x,y+xvd.yOffs[iFont],buff,len);
31     else XDrawString(xvd.display,xvd.backpix[iWin],xvd.gc,x,y+xvd.yOffs[iFont],
32     buff,len);
33     xvd.static_x[iWin]=x+width;
34     return len;
35 }

```

Questa funzione scrive sulla finestra grafica numero `iWin`, l'angolo superiore sinistro della scrittura è determinato dalle variabili `x` e `y` (in pixel), la variabile `iFont`, compresa tra 0 e `nFont-1`, sceglie il set di caratteri. Se `FillBack` vale TRUE prima della scrittura di ogni carattere viene riempito con il colore di sfondo il rettangolino che lo conterrà. Il resto funziona come la normale funzione C `printf()`.

Listato 3.5 La funzione `XReadAfter()`, che legge il carattere premuto

```

1  /*                                     12.MAR.2013                                xreadafter.cc
2                                                                                               by Giovanni Moruzzi
3  Started on 05.MAR.2005
4  */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <unistd.h>
9  #include <string.h>
10 #include <corso.h>
11
12 extern XVideoData xvd;
13
14 int XReadAfter(int iWin)
15 {
16     int key;
17     for (;;)
18     {
19         if (xvd.ev.type==KeyRelease)
20         {
21             key=XLookupKeysym(&xvd.ev.xkey,0);
22             if (key==65513) xvd.left_alt_pressed=FALSE;
23             else if (key==65514) xvd.right_alt_pressed=FALSE;
24             else if (key==65507) xvd.left_ctrl_pressed=FALSE;
25             else if (key==65508) xvd.right_ctrl_pressed=FALSE;
26             else if (key==65505) xvd.left_shift_pressed=FALSE;
27             else if (key==65506) xvd.right_shift_pressed=FALSE;
28             return -1;
29         }
30         if (xvd.ev.type==KeyPress)
31         {
32             key=XLookupKeysym(&xvd.ev.xkey,0);
33             if (key==65513) {xvd.left_alt_pressed=TRUE;return -1;}
34             if (key==65514) {xvd.right_alt_pressed=TRUE;return -1;}
35             if (key==65507) {xvd.left_ctrl_pressed=TRUE;return -1;}
36             if (key==65508) {xvd.right_ctrl_pressed=TRUE;return -1;}
37             if (key==65505) {xvd.left_shift_pressed=TRUE;return -1;}
38             if (key==65506) {xvd.right_shift_pressed=TRUE;return -1;}
39             if (xvd.left_shift_pressed || xvd.right_shift_pressed)
40                 key=XLookupKeysym(&xvd.ev.xkey,1);
41             if (xvd.right_alt_pressed) key=XLookupKeysym(&xvd.ev.xkey,2);
42             if (xvd.left_ctrl_pressed || xvd.right_ctrl_pressed)
43             {
44                 if (key==int('d')) return 65535;
45             }
46             if (key==65450) return int('*');
47             return key;
48         }
49     }
50 }

```

Listato 3.6 La funzione `XRedraw()`, che copia la grafica dalle pixmap di backup alle finestre del monitor

```
1  /*                               26.JUN.2006                               xredraw.cc
2                                     by Giovanni Moruzzi
3      Started on 14.MAR.2005
4      This function redraws the window by copying the backup pixmap
5  */
6
7  #include <stdarg.h>
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <string.h>
11
12 #include <corso.h>
13
14 extern XVideoData xvd;
15
16 void XRedraw()
17 {
18     int iWin;
19
20     // ..... copy subwindows
21     for (iWin=0;iWin<xvd.nWin;iWin++)
22     { // ..... skip unused windows
23         if (!xvd.UsedWin[iWin]) continue;
24         // ..... copy window
25         XCopyArea(xvd.display ,xvd.backpix[iWin] ,xvd.win[iWin] ,
26                 xvd.gc,0,0,xvd.width[iWin],xvd.height[iWin],0,0);
27         // ..... synchronize window
28         while(XCheckWindowEvent(xvd.display ,xvd.win[iWin] ,ExposureMask,&xvd.ev))
29         {
30             XSync(xvd.display ,FALSE);
31         }
32     }
33 }
```

Listato 3.7 La funzione `XWait()`, che mette a sospendere l'esecuzione del programma finché non viene premuto un bottone della tastiera o del mouse

```
1  /*                                02.APR.2009                                xwait.cc
2                                                                                               by Giovanni Moruzzi
3      Started on 14.MAR.2005
4  */
5
6  #include <stdarg.h>
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <string.h>
10
11 #include <corso.h>
12
13 extern XVideoData xvd;
14
15 void XWait(int iWin)
16 {
17     XSync(xvd.display, FALSE);
18     for (;;)
19     {
20         XNextEvent(xvd.display, &xvd.ev);
21         if (xvd.ev.type == Expose)
22         {
23             if (xvd.ev.xexpose.count > 0) continue;
24             XRedraw();
25             XSync(xvd.display, FALSE);
26             continue;
27         }
28         if (xvd.ev.type == KeyPress) break;
29         else if (xvd.ev.type == ButtonPress) break;
30     }
31 }
```