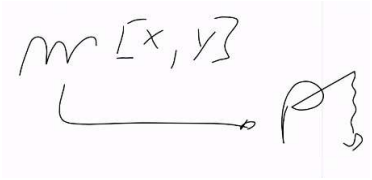
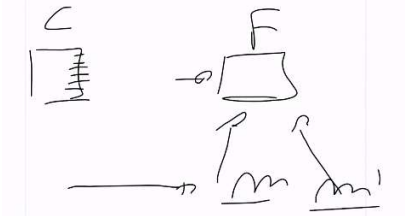
Gestione Eventi 5 WINDOWS + Messaggi evento

giovedì 27 novembre 2025 15:16

Vediamo ora i **messaggi evento** : ovvero una sorta di comunicazione/notifica asincrona . I quali messaggi non vengo processate implicitamente : il sistema deve scegliere esplicitamente di processarli. Quindi qui si parla di **polling reale** : se non vi sono thread che gestiscono messaggi , questi non verranno processati. Quindi la struttura caratteristica di un messaggio è la seguente : un numero che ne identifica il tipo e due valori numerici (parametri). In dettaglio : applicazioni rispondono in modo differente in base a questi parametri



Per processare un messaggio si deve sapere dove questi messaggi sono destinati : **finestre (mostrabile o no).** Per ogni finestra ci deve essere una **classe di finestre.**



Dove la generazione dei messaggi sono fatte dalle system call. In dettaglio:

1. **Generazione messaggi**
   a. Impacchetto sue caratteristiche in una tabella user
   b. `ATOM RegisterClass(const WNDCLASS *lpWndClass)`
   c. Dove la struttura della tabella è la seguente:

```
typedef struct _WNDCLASS {
      UINT style;
      WNDPROC lpfnWndProc;
      int cbClsExtra;
      int cbWndExtra;
      HANDLE hInstance;
      HICON hIcon;
      HCURSOR hCursor;
      HBRUSH hbrBackground;
      LPCTSTR lpszMenuName;
      LPCTSTR lpszClassName;
} WNDCLASS;
```

      i.
      ii. Style può prendere i seguenti attributi CS_HREDRAW oppure CS_VREDRAW
      iii. Mentre lpfnWndProc è l'indirizzo della funzione che fungerà da handler di tutti i messaggi evento
      iv.

- cbClsExtra: byte extra da allocare per esigenze del programamtore; tipicamente è 0
- cbWndExtra: altri byte extra da allocare per esigenze del programmatore; tipicamente è 0
- **hInstance: handler all'istanza del processo che ospita la procedura di finestra. NULL indica il processo corrente**
- hIcon: handler ad un'icona da usare per la finestra; come default usare il valore restituito dalla system call ``LoadIcon(NULL, IDI_APPLICATION"
- hCursor: handler ad un cursore da usare nella finestra; come default usare il valore restituito dalla system call ``LoadCursor(NULL, IDC_ARROW)"
- hbrBackground: handle al pennello di background
- lpszMenuName: stringa che specifica il nome del menu di default da usare. NULL se non ci sono menu
- **lpszClassName: stringa indicante il nome associato a questo tipo di finestra**

2. **Struttura dell' handler**:

```
LRESULT CALLBACK WindowProcedure(
      HWND hwnd,
      UINT uMsg,
      WPARAM wParam,
      LPARAM lParam
)
```

      a.
      b. Gli ultimi due parametri sono il payload del messaggio
      c. Quindi si porta per usare uno "switch{}" per gestire i diversi tipi di messaggio
3. **Creazione della finestra**

a.

```
HWND CreateWindow(LPCTSTR lpClassName,
                  LPTSTR lpWindowName,
                  DWORD dwStyle,
                  int x,
                  int y,
                  int nWidth,
                  int nHeight,
                  HWND hWndParent,
                  HMENU hMenu,
                  HANDLE hInstance,
                  PVOID lpParam)
```

Descrizione
• crea un nuovo oggetto finestra; NON la visualizza sullo schermo

Restituzione
• handle alla nuova finestra in caso di successo, NULL in caso di fallimento

b.

lpClassName: una stringa contenente il nome del tipo di finestra, precedentemente definito tramite RegisterClass()

lpWindowName: una stringa contenente l'intestazione della finestra

dwStyle: stile della finestra (default WS_OVERLAPPEDWINDOW)

x: posizione iniziale della finestra (coordinata x); usare CW_USEDEFAULT

y: posizione iniziale della finestra (coordinata y); usare CW_USEDEFAULT

nWidth: dimensione della finestra (coordinata x); usare CW_USEDEFAULT

nHeight: dimensione della finestra (coordinata y); usare CW_USEDEFAULT

hWndParent: handle alla finestra genitrice; NULL è il default

hMenu: handle ad un menu; se non ci sono menu usare NULL

hInstance: handle ad una istanza del processo di riferimento; NULL e' il default

lpParam: puntatore a parametri di creazione; NULL è il default

4. Polling

a.

dopo aver eseguito `RegisterClass()` e `CreateWindow()` il thread può cominciare a ricevere i messaggi evento entranti con il seguente loop:

```
while(GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
}
```

Da chiavi vituali a WM_CHAR per messaggi evento relativi ai devices

`msg` è una struttura di tipo `MSG` e `GetMessage()` è definita come:

```
INT GetMessage(LPMSG lpMsg,
               HWND hWnd,
               UINT wMsgFilterMin,
               UINT wMsgFilterMax)
```

b. Il secondo parametro di norma potrebbe essere NULL

c. Il terzo ed il quarto sono i range dei messaggi

d.

Descrizione
• riceve un messaggio nuovo. Ritorna solo se c'è un nuovo messaggio pendente o se viene ricevuto un messaggio di tipo WM_QUIT

Parametri
• lpMsg: indirizzo ad una struttura di tipo MSG
• hWnd: handle della finestra di cui si vogliono ricevere i messaggi; NULL per ricevere messaggi da tutte le finestre associate al processo
• wMsgFilterMin: valore più basso del tipo di messaggi evento da ricevere; 0 non pone limiti inferiori
• wMsgFilterMax: valore più alto del tipo di messaggi evento da ricevere; 0 non pone limiti superiori

Restituzione
• -1 se c'è un errore, 0 se viene ricevuto un messaggio di tipo WM_QUIT, un valore diverso da 0 e -1 se viene ricevuto un altro messaggio

5. **Spedizione messaggi** con codice numerico

a. Spedizione messaggio blocca finché la finestra non processa

b.

```
LRESULT SendMessage(HWND hWnd,
                    UINT Msg,
                    WPARAM wParam,
                    LPARAM lParam)
```

Descrizione
• invia un messaggio ad una finestra ; il messaggio verrà posto in testa alla coda dei messaggi-evento

Parametri
• hWnd: handle alla finestra che deve ricevere il messaggio; HWND_BROADCAST per mandare il messaggio a tutte le finestre prive di genitore (top level)
• Msg: intero che identifica il tipo di messaggio
• wParam: primo parametro del messaggio
• lParam: secondo parametro del messaggio

Restituzione
• ritorna il risultato del processamento del messaggio (true/false) ritorna quindi soltanto quando il messaggio evento è stato processato

c. **Notifica non bloccante**

```
BOOL PostMessage(HWND hWnd,
                 UINT Msg,
                 WPARAM wParam,
                 LPARAM lParam)
```

Descrizione
- invia un messaggio evento ad una finestra; il messaggio verrà posto in fondo alla coda dei messaggi evento

Parametri

i.
- hWnd: Handle alla finestra che deve ricevere il messaggio HWND_BROADCAST per mandare il messaggio a tutte le finestre prive di genitore (top level)
- Msg: intero che identifica il tipo di messaggio
- wParam: parametro del messaggio
- lParam: secondo parametro del messaggio

Restituzione
- 0 in caso di fallimento, un valore diverso da 0 in caso di successo; non attende il processamento del messaggio evento

ii. Il messaggio viene processato senza avere il blocco

6. Creazione messaggio

```
UINT RegisterWindowMessage(LPCTSTR lpString)
```

Descrizione
- crea un nuovo tipo di messaggio

a. **Parametri**
- lpString: stringa che assegna un nome al tipo di messaggio

Restituzione
- 0 indica un errore, ogni altro valore rappresenta il nuovo tipo di messaggio creato

b. Permette di scoprire nuovi codici

Vediamo un esempio:

```c
// this program polls for event-messages on a window

#include <windows.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <malloc.h>

#define READ_BUFFER 4096

#define AUTO_KILLER

#define SHOW
//#define VERBOSE

char *command1 = (char*)"command1";
char *command2 = (char*)"command2";
char *term = (char*)"term";

UINT command1_type = 0;
UINT command2_type = 0;
UINT term_type = 0;
UINT msg_type = 0;


HWND hWindow;
```

```c
LRESULT CALLBACK WndProc(HWND hWindow, UINT message, WPARAM wParam, LPARAM lParam) {

#ifdef VERBOSE
        printf("thread %d is processing an event-message\n", GetCurrentThreadId());
        fflush(stdout);
#endif

        switch(message)
        {
                case WM_CREATE:
                printf("window creation ok\n");
                fflush(stdout);
                return 0;
                case command1_type:
                printf("requested command 1\n");
                fflush(stdout);
                return 1;
                case command2_type:
                printf("requested command 2\n");
                fflush(stdout);
                return 2;
                case term_type:
                 printf("requested termination\n");
                fflush(stdout);
                PostQuitMessage(0);
                return 3;
                default:break;
        }
#ifdef SHOW
        if (message == WM_CLOSE) {
                printf("you will not close me this way!!\n");
                fflush(stdout);
                return 4;
        }
#endif

#ifdef VERBOSE
        printf("going for default  treatment\n");
#endif

        return (DefWindowProc(hWindow, message, wParam, lParam));

}
```

```c
DWORD WINAPI Killer(void * nothing) {

        int ret;
        UINT msg_type;
        char buff[READ_BUFFER];

        while (1) {
                scanf("%s", buff);

                printf("trying to kill with '%s' event-message\n", buff);

                msg_type = RegisterWindowMessage(buff);
                if (!msg_type) {
                        printf("Can't create '%s' event-message for error %u\n", buff, GetLastError());
                        fflush(stdout);
                        ExitProcess(-1);
                }
                else {
                        printf("event-message '%s' correctly registered - code is %u\n", buff, msg_type);
                }

                ret = PostMessage(HWND_BROADCAST, msg_type, 0, 0);
                printf("event-message post returned %d\n", ret);
        }

}
```

```c
void main(int argc, char *argv[]){

	struct _thread_info * thread_info = NULL;

	WNDCLASS wndclass;
	char nome_applicazione[] = "test";
	int ret;
	MSG msg;

	term_type = RegisterWindowMessage(term);
	if (!term_type) {
		printf("Can't create term message for error %d\n", GetLastError());
		fflush(stdout);
		ExitProcess(-1);
	}
	else {
		printf("term event-message correctly registered - code is %u\n", term_type);
	}

	command1_type = RegisterWindowMessage(command1);
	if (!command1_type) {
		printf("Can't create command1 message for error %d\n", GetLastError());
		fflush(stdout);
		ExitProcess(-1);
	}
	else {
		printf("command1 event-message correctly registered - code is %u\n", command1_type);
	}

	command2_type = RegisterWindowMessage(command2);
	if (!command1_type) {
		printf("Can't create command2 message for error %d\n", GetLastError());
		fflush(stdout);
		ExitProcess(-1);
	}
	else {
		printf("command2 event-message correctly registered - code is %u\n", command2_type);
	}

	wndclass.style = CS_HREDRAW | CS_VREDRAW;
	wndclass.lpfnWndProc = WndProc;
	wndclass.cbClsExtra = 0;
	wndclass.cbWndExtra = 0;
	wndclass.hInstance = NULL;
	wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
	wndclass.hCursor = LoadIcon(NULL, IDC_ARROW);
	wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
	wndclass.lpszMenuName = NULL;
	wndclass.lpszClassName = nome_applicazione;

	if (!RegisterClass(&wndclass)) {
		printf("Can't register class"); fflush(stdout);
		ExitProcess(-1);
	}
```

```c
	hWindow = CreateWindow(nome_applicazione,
		"Test su messaggi-evento",
		WS_OVERLAPPEDWINDOW,
		CW_USEDEFAULT,
		CW_USEDEFAULT,
		CW_USEDEFAULT,
		CW_USEDEFAULT,
		NULL,
		NULL,
		NULL,
		NULL);

	if (hWindow == INVALID_HANDLE_VALUE) {
		printf("Can't create window for error %d\n", GetLastError());
		fflush(stdout);
		ExitProcess(-1);
	}

#ifdef SHOW
	ShowWindow(hWindow, SW_SHOW);
#endif

#ifdef AUTO_KILLER
	if ((CreateThread(NULL, 0, Killer, NULL, 0, NULL)) == INVALID_HANDLE_VALUE) {
		printf("Can't start event-message listening thread\n");
		fflush(stdout);
		ExitProcess(-1);
	}
#endif

	printf("start polling on main tread\n");
	fflush(stdout);
	while (ret = GetMessage(&msg, NULL, 0, 0)) {
		if (ret == -1) {
			printf("event-message poll error\n");
		}
		else {
			#ifdef VERBOSE
			printf("got event-message while polling - type is %u\n", msg.message);
			printf("thread %d is dispatching an event-message\n", GetCurrentThreadId());
			fflush(stdout);
			#endif
			TranslateMessage(&msg);
			DispatchMessage(&msg);
		}
	}

	ExitProcess(0);

}
```

Quindi se la eseguo apro una finestra nella finestra. Nell'esempio command2 la fa non essere visibile, command1 si ; mentre se digito term si esce dalla fase di polling. Se includo la macro VERBOSE ad ogni evento(magari muovo il muose) viene processato e stampato. Vediamo ora due classi predefinite: **button** (possibile click sopra) **e boxes** (testo): quindi usiamo la macro WM_GETTEXT / WM_SETTEXT per ottenere o scrivere testo . Ultima cosa : possiamo anche creare dei **menu**:

```
HMENU WINAPI CreateMenu(void);

BOOL WINAPI AppendMenu( _In_     HMENU     hMenu
                        _In_     UINT      uFlags,
                        _In_     UINT_PTR  uIDNewItem,
                        _In_opt_ LPCTSTR   lpNewItem );
```

Codice o handle identificativo

Tipo di rappresentazione

Handle alla finestra da mostrare e modalita' di visualizzazione

Nome della entry

```
BOOL WINAPI ShowWindow( _In_ HWND hWnd,
                        _In_ int  nCmdShow );
```

Vediamo un esempio :

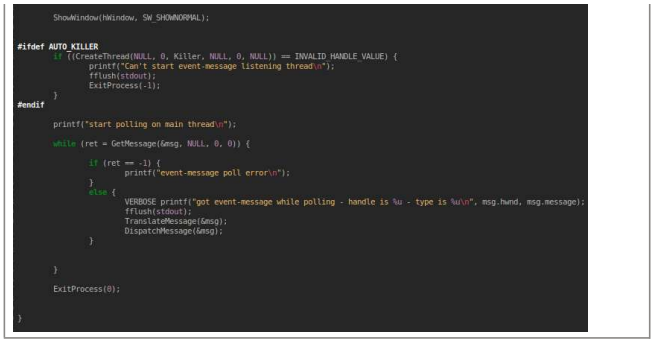
```c
// this program implements a mini editor based on windows and event-messages
// it also has a console component for inspecting the actual software execution and
// for controlling it via stdin

#include <windows.h>
#include <WinUser.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <malloc.h>

#define READ_BUFFER 4096
#define AUTO_KILLER
#define WITH_TEXT
#define WITH_BUTTON

#define VERBOSE if(0)

char text[READ_BUFFER];
HANDLE file;

char *command1 = (char*)"command1";
char *command2 = (char*)"command2";
char *term = (char*)"term";

UINT command1_type = 0;
UINT command2_type = 0;
UINT term_type = 0;
UINT msg_type = 0;


HWND hWindow;
HWND hwndButtonA;
HWND hwndButtonB;
HWND hEdit;
```

```c
LRESULT CALLBACK WndProc(HWND hWindow, UINT message, WPARAM wParam, LPARAM lParam) {
	DWORD size;
	if (message == WM_CREATE) {
		printf("window creation ok\n");
		fflush(stdout);
		return 0;
	}
	if (message == command1_type) {
		printf("requested command 1\n");
		fflush(stdout);
		return 1;
	}
	if (message == command2_type) {
		printf("requested command 2\n");
		fflush(stdout);
		return 2;
	}
	if (message == term_type) {
		printf("requested termination\n");
		fflush(stdout);
		PostQuitMessage(0);
		return 3;
	}
	if (message == WM_COMMAND) {
		VERBOSE printf("process asked to run some menu/command - param is %d\n", wParam);
		if (wParam == BN_CLICKED) {
			printf("button pressed\n");
			if ((HWND)lParam == hwndButtonA) {
				printf("button identified - load text\n");
				SendMessage(hEdit, WM_SETTEXT, READ_BUFFER, (LPARAM)text);
				fflush(stdout);
			}
			if ((HWND)lParam == hwndButtonB) {
				printf("button identified - store text\n");
				SendMessage(hEdit, WM_GETTEXT, READ_BUFFER, (LPARAM)text);
				printf("text window got message:\n%s\n", text);

				SetFilePointer(file, 0, NULL, FILE_BEGIN);
				SetEndOfFile(file);
				if (WriteFile(file, text, strlen(text), &size, NULL) == 0) {
					printf("Cannot write to file\n");
					fflush(stdout);
				}
				fflush(stdout);
			}
		}
		fflush(stdout);
#ifdef WITH_TEXT
		if (wParam == 128) {
			SendMessage(hEdit, WM_GETTEXT, READ_BUFFER, (LPARAM)text);
			printf("mineditor got request to quit message - text box had this content:\n%s\n", text);
			printf("Exiting process\n");
			fflush(stdout);
			ExitProcess(0);
		}
#endif
	}

	if (message == WM_CLOSE) {
		printf("process asked to terminate for windows closure\n");
		fflush(stdout);
		//ExitProcess(0);
		return 4;
	}

	//if (message == WM_PAINT) printf("got WM_PAINT\n");
	VERBOSE printf("going for default  treatment\n");
	return (DefWindowProc(hWindow, message, wParam, lParam));

}
```

```c
DWORD WINAPI Killer(void * nothing) {

	int ret;
	UINT msg_type;
	char buff[READ_BUFFER];

	while (1) {
		scanf("%s", buff);

		printf("trying to kill with '%s' event-message\n", buff);
		fflush(stdout);

		msg_type = RegisterWindowMessage(buff);
		if (!msg_type) {
			printf("Can't create '%s' event-message for error %u\n", buff, GetLastError());
			fflush(stdout);
			ExitProcess(1);
		}
		else {
			printf("event-message '%s' correctly registered - code is %u\n", buff, msg_type);
			fflush(stdout);
		}

		ret = PostMessage(HWND_BROADCAST, msg_type, 0, 0);
		printf("event-message post returned %d\n", ret);
		fflush(stdout);
	}

}
```

```
void main(int argc, char *argv[]){

	struct _thread_info * thread_info = NULL;

	WNDCLASS wndclass;
	char nome_applicazione[] = "test";
	int ret;
	DWORD size;
	DWORD fileSize;
	MSG msg;
	HMENU hMenu;
	HMENU MenuList;
	HMENU BMenu;


	file = CreateFile(argv[1], GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
	if (file == INVALID_HANDLE_VALUE) {
		printf("Cannot open file to edit\n");
		fflush(stdout);
		return;
	}

	fileSize = GetFileSize(file, NULL);
	if (fileSize == INVALID_FILE_SIZE) {
		printf("Failed to get file size\n");
		CloseHandle(file);
		return;
	}
	if (fileSize > (READ_BUFFER - 1)) {
		printf("File too large\n");
		CloseHandle(file);
		return;
	}

	if (ReadFile(file, text, READ_BUFFER, &size, NULL) == 0) {
		printf("Cannot read from file\n");
		fflush(stdout);
		return;
	}

	text[size] = '\0';

	term_type = RegisterWindowMessage(term);
	if (!term_type) {
		printf("Can't create term message for error %d\n", GetLastError());
		fflush(stdout);
		ExitProcess(1);
	}
	else {
		printf("term event-message correctly registered - code is %u\n", term_type);
		fflush(stdout);
	}

	command1_type = RegisterWindowMessage(command1);
	if (!command1_type) {
		printf("Can't create command1 message for error %d\n", GetLastError());
		fflush(stdout);
		ExitProcess(1);
	}
	else {
		printf("command1 event-message correctly registered - code is %u\n", command1_type);
	}

	command2_type = RegisterWindowMessage(command2);
	if (!command1_type) {
		printf("Can't create command2 message for error %d\n", GetLastError());
		fflush(stdout);
		ExitProcess(1);
	}
	else {
		printf("command2 event-message correctly registered - code is %u\n", command2_type);
		fflush(stdout);
	}
```

```
	wndclass.style = CS_HREDRAW | CS_VREDRAW;
	wndclass.lpfnWndProc = WndProc;
	wndclass.cbClsExtra = 0;
	wndclass.cbWndExtra = 0;
	wndclass.hInstance = NULL;
	wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
	wndclass.hCursor = LoadCursor(NULL, (LPTSTR)IDC_ARROW);
	wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
	wndclass.lpszMenuName = NULL;
	wndclass.lpszClassName = nome_applicazione;

	if (!RegisterClass(&wndclass)) {
		printf("Can't register class"); fflush(stdout);
		ExitProcess(1);
	}

	//managing the menu
	hMenu = CreateMenu();
	AppendMenu(hMenu, MF_STRING, 128, "Quit");
	MenuList = CreateMenu();
	AppendMenu(MenuList, MF_POPUP, (UINT_PTR)hMenu, "menu");

	hWindow = CreateWindow(nome_applicazione,
		TEXT("Windows mini editor"),
		WS_OVERLAPPEDWINDOW,
		10, 20, 650, 600,
		NULL,
		MenuList,
		NULL,
		NULL);
	if (hWindow == INVALID_HANDLE_VALUE) {
		printf("Can't create window for error %d\n", GetLastError());
		fflush(stdout);
		ExitProcess(-1);
	}

#ifdef WITH_BUTTON
	hwndButtonA = CreateWindow(
		"BUTTON",  // Predefined class
		"Load text",      // Button text
		WS_TABSTOP | WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,  // Styles
		10,         // x position
		20,         // y position
		100,        // Button width
		60,        // Button height
		hWindow,    // Parent window
		NULL,       // No menu.
		(HINSTANCE)GetWindowLong(hWindow, GWLP_HINSTANCE),
		NULL);      // Pointer not needed.

	hwndButtonB = CreateWindow(
		"BUTTON",  // Predefined class
		"Store text",      // Button text
		WS_TABSTOP | WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,  // Styles
		10,         // x position
		80,         // y position
		100,        // Button width
		60,        // Button height
		hWindow,    // Parent window
		NULL,       // No menu.
		(HINSTANCE)GetWindowLong(hWindow, GWLP_HINSTANCE),
		NULL);      // Pointer not needed.

#endif

#ifdef WITH_TEXT
	hEdit = CreateWindowEx(WS_EX_CLIENTEDGE,
		"EDIT",
		"",
		WS_CHILD | WS_VISIBLE |
		ES_MULTILINE | ES_AUTOVSCROLL | ES_AUTOHSCROLL,
		120,
		20,
		500,
		500,
		hWindow,
		NULL,
		GetModuleHandle(NULL),
		NULL);
#endif
```



```
        ShowWindow(hWindow, SW_SHOWNORMAL);


#ifdef AUTO_KILLER
        if (!(CreateThread(NULL, 0, Killer, NULL, 0, NULL)) == INVALID_HANDLE_VALUE) {
                printf("Can't start event-message listening thread\n");
                fflush(stdout);
                ExitProcess(-1);
        }
#endif

        printf("start polling on main thread\n");

        while (ret = GetMessage(&msg, NULL, 0, 0)) {

                if (ret == -1) {
                        printf("event-message poll error\n");
                }
                else {
                        VERBOSE_printf("got event-message while polling - handle is %u - type is %u\n", msg.hwnd, msg.message);
                        fflush(stdout);
                        TranslateMessage(&msg);
                        DispatchMessage(&msg);
                }

        }

        ExitProcess(0);

}
```

Quindi con questo esempio va a simulare una text area : leggo e scrivo stringhe su file. Se bottone leggi leggo stringhe da file, altrimenti le scrivo su file