



UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Dispense del corso di Fondamenti di Informatica 1

Il linguaggio C - Altre librerie

Aggiornamento del 13/11/2020

La libreria assert.h

- La libreria assert.h contiene una macro chiamata **assert** che può essere usata in fase di debug per controllare alcune condizioni la cui verifica è necessaria per il programma.
- La sua sintassi è:

```
assert ( <int-expression> ) ;
```

- La macro assert può essere utilizzata come una funzione, ad esempio:

```
int x = 3;  
assert(x >= 0);  
double r = radice(x);
```

- Se la condizione non è verificata, ossia l'espressione ha valore uguale a 0, l'esecuzione termina con un errore (un *assertion failure*), altrimenti procede normalmente se l'espressione ha un valore diverso da 0.
- La condizione usata come parametro per una assert può essere qualsiasi espressione di tipo intero.

La libreria `assert.h`

- La definizione della macro `assert` dipende però dalla definizione di un'altra macro chiamata `NDEBUG`. Quando essa è definita `assert` viene *disabilitata* (ossia viene sostituita con un'espressione che non fa nulla, in questo caso con `(void) 0`).
- La libreria standard però non definisce mai `NDEBUG` nei suoi file... chi lo definisce allora??
- È il compilatore che si occupa di gestire `NDEBUG`, a seconda della modalità in cui si sta compilando!
- Quando si compila in modalità *debug* essa non è definita e tutti gli `assert` sono abilitati, quando si compila in modalità *release* invece essa viene definita e tutti gli `assert` vengono disabilitati.

La libreria float.h

- La libreria float.h contiene delle macro che rappresentano le caratteristiche dei tipi floating point (quindi float, double e long double).
- Dato che i tipi floating point sono composti da diverse parti (segno, esponente e mantissa) a seconda della precisione (singola, doppia o più) le diverse parti possono assumere diversi valori minimi e massimi.
- In float.h sono definite varie macro che rappresentano valori massimi e minimi di ogni parte dei tipi floating point o i massimi e minimi rappresentabili con essi.
- Elencarle tutte sarebbe estremamente noioso, vediamo quindi solo alcune, le altre sono tutte disponibili sulla reference.
 - `FLT_MIN` e `FLT_MAX`: valori minimi e massimi assumibili da un float.
 - `FLT_MIN_EXP` e `FLT_MAX_EXP`: valori minimi e massimi assumibili dall'esponente in un float.
 - `FLT_MANT_DIG` e `FLT_DIG`: numero di cifre che compongono la mantissa, rispettivamente in binario e in decimale.
- Sono anche definite le macro per i double e i long double, i nomi sono gli stessi delle precedenti, basta sostituire «FLT» con «DBL» o «LDBL».

La libreria limits.h

- Se float.h definisce i limiti dei tipi floating point, limits.h definisce quelli dei tipi interi, sempre utilizzando delle macro.
- In questo caso avremo delle macro i cui nomi seguiranno questo schema:

`<tipo>_<limite>`

- Dove `tipo` è uno dei tipi interi fondamentali e `limite` può essere MIN o MAX, ad esempio:
 - `UINT_MAX`: massimo valore assumibile da un `int` senza segno.
 - `CHAR_MIN`: minimo valore assumibile da un `char` con segno.
 - `ULLONG_MAX`: massimo valore assumibile da un `unsigned long long int`.
 - `SHRT_MIN`: minimo valore assumibile da uno `short`.

Ancora su float.h e limits.h

- Perché dovremmo utilizzare queste macro quando sappiamo benissimo quanti byte occupano i vari tipi?
- Perché è meglio usare queste macro invece che scrivere direttamente nel codice i valori che ci interessano? Se volessimo usare il valore massimo rappresentabile in un intero senza segno, non potremmo direttamente scrivere `0xffffffff` nel codice?
- Perché le macro contenute in queste due librerie sono *implementation defined*!
- Se la dimensione di un intero passasse da 32 a 16 bit cambiando architettura un eventuale valore massimo scritto a mano da noi potrebbe non essere più valido, creando così un bug.
- A seconda dell'architettura le dimensioni dei tipi potrebbero cambiare, utilizzando queste macro siamo sicuri di utilizzare sempre i valori giusti.
- Nota: per il compilatore di Visual Studio non c'è differenza tra `double` e `long double`, entrambi usano 64 bit.

Ancora su stdlib.h

- Finora abbiamo visto solamente alcuni degli elementi presenti in stdlib.h.
- In questa libreria sono presenti anche altre funzioni utili, che non hanno a che fare con l'allocazione dinamica della memoria.
- stdlib.h contiene anche funzioni:
 - Funzioni per la conversione di stringhe in tipi interi e floating point.
 - Funzioni per la generazione di numeri casuali.
 - Utilità per la gestione di un programma.
 - Alcune funzioni per l'aritmetica intera.
- Vediamo ora alcune di esse più nel dettaglio.

Conversione tra stringhe e numeri

- Potrebbe capitare di dover convertire una stringa contenente la rappresentazione di un numero, intero o con la virgola, nella corrispondente variabile di tipo intero o con la virgola.
- Scrivere le funzioni per fare queste conversioni ogni volta sarebbe un lavoro inutile e noioso, per questo nella standard library sono state previste delle funzioni apposite, che si dividono in due gruppi.
- Il primo è costituito da funzioni con il nome `ato*`():
 - `atof()`: per conversione in numeri floating point.
 - `atoi()`, `atol()`, `atoll()`: per conversione in numeri interi.
- Il secondo invece da funzioni con il nome `strto*`():
 - `strtod()`, `strtof()`, `strtold()`: per conversioni in floating point.
 - `strtol()`, `strtoll()`, `strtoul()`, `strtoull()`: per conversioni in interi.
- La differenza tra le funzioni di questi due gruppi sta nelle dichiarazioni delle funzioni, come vedremo tra poco.

Funzioni ato*()

- Le funzioni del gruppo ato*() hanno tutte un unico parametro, una stringa, e a seconda della conversione che effettuano ritornano un tipo diverso.
- L'unica conversione da stringa a floating point è effettuata da atof(), la cui dichiarazione è:

```
double atof(const char *str);
```

- Essa ritorna il double rappresentato testualmente nella stringa `str`.
- Le altre tre funzioni si occupano della conversione di stringhe in tipi interi:

```
int atoi(const char *str);
```

```
long int atol(const char *str);
```

```
long long int atoll(const char *str);
```

Funzione atof()

- Le stringhe passate alla funzione atof(), per dare luogo a una conversione valida, devono rispettare alcune regole.
- La funzione prima scarta tutti i whitespaces (spazi, tab e a capo) fino al primo carattere non whitespace. Poi a partire da quel carattere ci può essere un +/- opzionale e poi tutti i caratteri che rappresentano il numero.
- La rappresentazione può essere la stessa che si usa per un literal di tipo floating point in C.
- Dopo i caratteri che rappresentano il numero ce ne possono essere altri ma la funzione li ignora.
- Quando una conversione fallisce la funzione ritorna 0.0.
- Quando si prova a convertire un numero che non può essere rappresentato con quel tipo di dato si genera un undefined behavior.

Funzione atof()

- Ecco alcuni esempi di chiamate alla funzione atof():

```
double d1 = atof("24.2");  
double d2 = atof(" -24.ccc");  
double d3 = atof("cc-24.2");  
double d4 = atof(" -24.4e-2");
```

- Quanto varranno d1, d2, d3 e d4 dopo l'esecuzione?
- A parte d3 tutti gli altri verranno correttamente convertiti nei rispettivi valori in double rappresentati nelle stringhe.
- Nel caso di d3 il valore restituito dalla funzione sarà 0.0 in quanto la stringa inizia con delle lettere e non è una rappresentazione valida di un numero floating point.

Funzioni atoi(), atol(), atoll()

- Le funzioni atoi(), atol() e atoll() funzionano in modo analogo ad atof().
- Anche in questo caso la stringa passata deve rispettare praticamente le stesse regole già viste per atof(), con la differenza che la rappresentazione del numero non è una di tutte le possibili per i literals, ma solamente quella che usa solo cifre decimali precedute da un +/- opzionale.
- Ad esempio:

```
int i1 = atoi("s354");  
long i2 = atol("45u");  
long long i3 = atoll(" 45aaaa");  
int i4 = atoi("0xfa");
```

- In questo caso sia `i1` che `i4` saranno 0, dato che le due relative stringhe non sono valide per la conversione.

Funzioni strt*()

- Tutte le funzioni della famiglia strt*() svolgono la stessa funzione delle ato*() ma la loro dichiarazione è diversa.
- Per le conversioni di stringhe in numeri floating point esistono:

```
double strtod (const char* str, char** endptr);  
float strtof (const char* str, char** endptr);  
long double strtold (const char* str, char** endptr);
```

- Mentre per le conversioni di stringhe in numeri interi esistono:

```
long int strtol (const char* str, char** endptr, int base);  
long long int strtoll (const char* str, char** endptr, int base);  
unsigned long int strtoul (const char* str, char** endptr, int base);  
unsigned long long int strtoull (const char* str, char** endptr, int base);
```

- Tutte le funzioni hanno un parametro `char **endptr` e quelle per interi hanno un ulteriore parametro `int base`.
- A cosa servono questi due nuovi parametri?

Funzioni `strto*()`

- Il parametro `endptr` è un *puntatore a puntatore* che deve puntare a un puntatore già allocato il quale, al termine della conversione punterà al primo carattere della stringa successivo al numero appena convertito.
- Puntando a un elemento della stringa precedente esso può di nuovo essere usato come stringa per essere passato a una funzione `strto*()`! L'esempio successivo chiarirà meglio questa cosa.
- Se `endptr` è un puntatore nullo allora non viene usato.
- Il parametro `base` presente nelle funzioni per conversioni in numeri interi serve ad indicare la base in cui è rappresentato il numero nella stringa.
- Se `base` è 0 allora il formato in cui è scritto il numero nella stringa viene usato per determinarne la base, (un prefisso "0" per indicare numeri in ottale e "0x" o "0X" per numeri in esadecimale).
- Anche per queste funzioni se la conversione fallisce viene ritornato uno 0 e se il numero letto nella stringa non può essere rappresentato nel tipo richiesto si genera un undefined behavior.

Funzioni strt*()

- Per fare un esempio consideriamo il seguente codice:

```
char *start = "23.5e-2 56.4 390.2e2", *end;  
float f1 = strttof(start, &end);  
float f2 = strttof(end, NULL);
```

- Nella prima riga definisco due puntatori, il primo inizializzato con l'indirizzo di un stringa costante e il secondo non inizializzato.
- Nella seconda riga chiamo strttof() passandole `start` e un *puntatore al puntatore* `end`.
- Dopo la chiamata, `end` punterà al char successivo all'ultimo usato per la conversione del numero, quindi in questo caso allo spazio che c'è dopo "23.5e-2".
- Nella terza riga possiamo quindi riutilizzare `end` passandolo come primo parametro a strttof(). `end` punterà quindi a una stringa che contiene:
" 56.4 390.2e2".

Attenzione: La stringa è sempre la stessa!

- Dopo le due chiamate `f1` e `f2` varranno rispettivamente 0.235 e 56.4.

Generazione di numeri pseudo casuali

- Come già detto, `stdlib.h` contiene anche funzioni per la generazione di numeri casuali, più precisamente due funzioni, chiamate `rand()` e `srand()`.
- La funzione `rand()` ha la seguente dichiarazione:

```
int rand (void);
```

e ritorna un numero intero pseudo-casuale nel range che va da 0 a `RAND_MAX` (`RAND_MAX` è una macro definita sempre in `stdlib.h`).

- Il numero è detto pseudo-casuale perché l'algoritmo che viene usato per generarlo è in grado di generare una sequenza di numeri non legati tra loro ma ha bisogno di essere inizializzato usando un *seed*, altrimenti la sequenza sarà sempre uguale!
- Un *seed* non è altro che un altro intero che viene usato dall'algoritmo come numero di partenza (da cui la parola *seed*, ossia «seme» in inglese).

Generazione di numeri pseudo casuali

- Il seed dell'algoritmo si può impostare attraverso l'altra funzione che abbiamo nominato, `srand()`, la cui dichiarazione è:

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

- La funzione non fa altro che prendere il parametro che le viene passato e usarlo come seed dell'algoritmo di generazione.
- Se il seed determina tutti i numeri casuali generati successivamente da `rand()` sarebbe l'ideale se fosse anch'esso il più casuale possibile!
- Il modo più banale per decidere il seed è quello di usare il tempo corrente.
- Il tempo corrente può essere ottenuto attraverso la funzione `time()`.
- Essa è definita nella libreria `time.h` e ritorna il tempo corrente in secondi a partire dalla mezzanotte del 1 gennaio 1970.
- Potremo scrivere ad esempio:

```
srand((unsigned int) time(NULL));
```

Generazione di numeri pseudo casuali

- L'importante è inizializzare il seed solamente una volta all'inizio del programma e poi fare solo chiamate a rand().
- Ad esempio:

```
#include <stdlib.h>
#include <time.h>

int main(void) {
    srand((unsigned int)time(NULL));

    int r1 = rand();
    int r2 = rand();

    // altre operazioni...

    return 0;
}
```

La funzione `exit()`

- Finora abbiamo visto solo un modo per terminare un programma, lasciare che esso esegua tutte le istruzioni fino al `return` statement del `main`.
- Ma è possibile anche decidere volontariamente di terminare un programma in qualsiasi momento.
- Questo è possibile chiamando una funzione presente in `stdlib.h`:

```
void exit (int exit_code);
```

- Il parametro `exit_code` rappresenta il valore che sarebbe stato ritornato dal `return` statement del `main`.
- Il valore di `exit_code` può essere un qualsiasi intero ma solitamente il suo valore fornisce informazioni sullo stato del programma e potenzialmente sul motivo per cui l'esecuzione è terminata.

La funzione `exit()`

- La convenzione per trattare gli exit code dei programmi è la seguente:
 - 0: il programma è terminato correttamente senza errori.
 - Qualsiasi altro valore: il programma è terminato con un errore, il numero ritornato può anche indicare di quale errore si tratta.
- Sempre in `stdlib.h` sono state definite due macro per questo scopo:
 1. `EXIT_SUCCESS`: viene sostituita dal valore 0 e serve per indicare che il programma sta uscendo con successo, senza errori.
 2. `EXIT_FAILURE`: viene generalmente sostituita dal valore 1 e viene usata per indicare che il programma è terminato con un errore.

La funzione div()

- La funzione `div()` esegue la divisione intera tra due numeri interi, più precisamente due `int`.

```
div_t div (int numer, int denom);
```

- Come si può notare il suo tipo di ritorno è `div_t`, questo non è altro che una struct che contiene quoziente e resto, la sua definizione è la seguente:

```
typedef struct _div_t {  
    int quot;  
    int rem;  
} div_t;
```

- Dove `quot` contiene il quoziente e `rem` il resto della divisione.
- Nella standard library ne viene fatto il typedef quindi non c'è bisogno di scrivere struct davanti al nome del tipo.

La funzione abs()

- La funzione `abs()` ritorna il valore assoluto di un numero intero, la sua dichiarazione è la seguente:

```
int abs (int n);
```

- Il parametro `n` è il numero di cui determinare il valore assoluto.
- Ad esempio:

```
int a;  
a = abs(-48);
```

- Dopo la seconda riga `a` vale 48, ossia il valore assoluto di -48.

La libreria math.h

- La libreria math.h contiene tutte le funzioni matematiche di base.
- A differenza di quelle poche contenute in stdlib.h, che lavorano su interi, quelle di math.h lavorano su numeri floating point.
- Questa libreria contiene varie categorie di funzioni:
 - Funzioni trigonometriche.
 - Funzioni per esponenziali e logaritmi.
 - Potenze e radici.
 - Funzioni di arrotondamento.
 - Funzioni di massimo e minimo e altre funzioni utili
 - Macro per rappresentare facilmente nan e infiniti.
 - Macro per controllare nan e infiniti.
- Non elencheremo tutte le funzioni di math.h perché sarebbe estremamente noioso, in ogni caso la reference è sempre disponibile.

Trigonometria

- In `math.h` sono contenute tutte le funzioni trigonometriche, di seguito ne elenchiamo alcune:

```
double sin (double x);
```

```
double cos (double x);
```

```
double tan (double x);
```

- Queste funzioni fanno esattamente quello che ci si aspetterebbe da loro, calcolano il seno/coseno/tangente di un `double`.
- Ovviamente esistono anche le relative funzioni inverse:

```
double asin (double x);
```

```
double acos (double x);
```

```
double atan (double x);
```

- Nota: tutte le funzioni trigonometriche lavorano in *radianti*, non in gradi!

Esponenziali e logaritmi

- In `math.h` sono disponibili anche tutte le funzioni per calcolare esponenziali e logaritmi, ad esempio:

```
double exp (double x);  
double log (double x);
```

che calcolano rispettivamente e^x e $\ln(x)$.

- Esistono anche altre funzioni per calcolare i logaritmi:

```
double log10 (double x);  
double log2 (double x);
```

che calcolano rispettivamente $\log_{10}(x)$ e $\log_2(x)$.

- Ed esiste anche un'altra funzione per calcolare gli esponenziali:

```
double exp2 (double x);
```

che calcola, invece che e^x , il valore di 2^x .

Potenze

- La funzione per calcolare una potenza qualsiasi è la seguente:

```
double pow(double base, double exponent);
```

- Che calcola esattamente $base^{exponent}$.
- Le due funzioni per calcolare radici presenti in math.h sono le due seguenti:

```
double sqrt(double x);
```

```
double cbrt(double x);
```

- Che calcolano rispettivamente \sqrt{x} e $\sqrt[3]{x}$.

Arrotondamenti

- In `math.h` sono disponibili anche funzioni per eseguire degli arrotondamenti:

```
double ceil (double x);
```

```
double floor (double x);
```

- `ceil()` arrotonda x all'intero superiore e `floor()` arrotonda x all'intero inferiore.
- Esistono anche altre funzioni più generiche di arrotondamento, tra cui:

```
double round (double x);
```

```
long int lround (double x);
```

```
long long int llround (double x);
```

- La funzione `round()` ritorna x arrotondato all'intero più vicino. Le sue due varianti `lround()` e `llround()` fanno la stessa cosa ma castano il risultato rispettivamente a `long int` e `long long`.

Valore assoluto per floating point

- In `math.h` esiste anche una versione floating point di `abs()` visto in `stdlib.h`.

```
double fabs (double x);
```

- Non fa altro che ritornare in un `double` il valore assoluto del `double x`, ad esempio:

```
double a;  
a = fabs(-32.5);
```

- Dopo l'esecuzione della seconda riga `a` varrà 32.5.

Resto per floating point

- In `math.h` esiste una versione floating point dell'operatore %:

```
double fmod(double x, double y);  
float fmodf(float x, float y);
```

- Queste funzioni ritornano il «resto con la virgola» della divisione x/y , ovvero il valore $x - n * y$ con n uguale a x/y senza la parte frazionaria.

```
double a;  
a = fabs(-32.5, 5);
```

- Dopo l'esecuzione della seconda riga `a` varrà 2.5, perché il 5 sta nel 32.5 6 volte (-30) e quindi «resta» 2.5.

Not-a-number e infiniti

- In `math.h` vengono anche messe a disposizione due macro importanti per rappresentare facilmente i nan e gli infiniti.
- Questi vengono rappresentati nelle variabili floating point usando combinazioni particolari di esponente e mantissa.
- Ad esempio, per numeri floating point a 32 bit, nan e infiniti sono rappresentati dalle combinazioni:

	Esponente	Mantissa
Infiniti	255	0
Nan	255	Non zero

- Per comodità, e quindi per non dover scrivere tali float a mano, si possono usare le due macro:
 - `NAN`: rappresenta il valore per il nan.
 - `INFINITY`: rappresenta il valore per gli infiniti.

Not-a-number e infiniti

- Esistono anche delle macro per controllare se un numero floating point ricade in uno dei due casi elencati in precedenza:

`isinf(x)`

`isnan(x)`

- Queste due macro lavorano come se fossero funzioni.
- Entrambe hanno un solo parametro `x` che deve essere un'espressione di tipo floating point.
- Vengono valutate come espressioni intere che valgono un valore diverso da zero (`true`) se `x` verifica la condizione che si stava esaminando oppure zero (`false`).

La libreria `errno.h`

- La libreria `errno.h` contiene varie macro che servono per controllare errori generati durante l'esecuzione di un programma.
- Può capitare, durante l'esecuzione di un programma, che alcune delle funzioni, ad esempio quelle di `math.h`, generino degli errori, ad esempio provando a calcolare la radice di un numero negativo.
- La libreria `errno.h` ci fornisce strumenti per controllare questi errori.
- La macro più importante è:

`errno`

- Questa viene espansa a un valore intero con segno modificabile che contiene un codice di errore.
- Nel caso non si siano verificati errori, `errno` vale 0.
- Le funzioni che usano `errno` non lo impostano a 0 se tutto è andato bene, quindi se si vuole controllarne il valore è necessario azzerarlo prima di chiamare la funzione interessata, altrimenti potremmo trovarci ad analizzare un codice di errore impostato da un'altra funzione.

La libreria errno.h

- In errno.h sono anche contenute altre macro che identificano i codici di errore più comuni:
 - EDOM: argomento di una funzione matematica fuori dal dominio della funzione.
 - EILSEQ: sequenza di byte non ammessa.
 - ERANGE: Risultato troppo grande per essere memorizzato.
- Una volta ottenuto un codice di errore è possibile stamparne una descrizione testuale se necessario.
- Per fare questo si può utilizzare la funzione **strerror()**:

```
char* strerror(int errnum);
```

- Questa funzione ritorna un puntatore a una stringa che non deve essere modificata che contiene la descrizione dell'errore.

Esempio

- Facciamo un esempio di come si può utilizzare `errno` per controllare un errore:

```
#include <math.h>
#include <errno.h>
```

```
int main(void)
{
    errno = 0;
    double res = sqrt(-1);
    if (errno != 0){
        fprintf(stderr, "ERROR: %s", strerror(errno));
        return EXIT_FAILURE;
    }

    return 0;
}
```

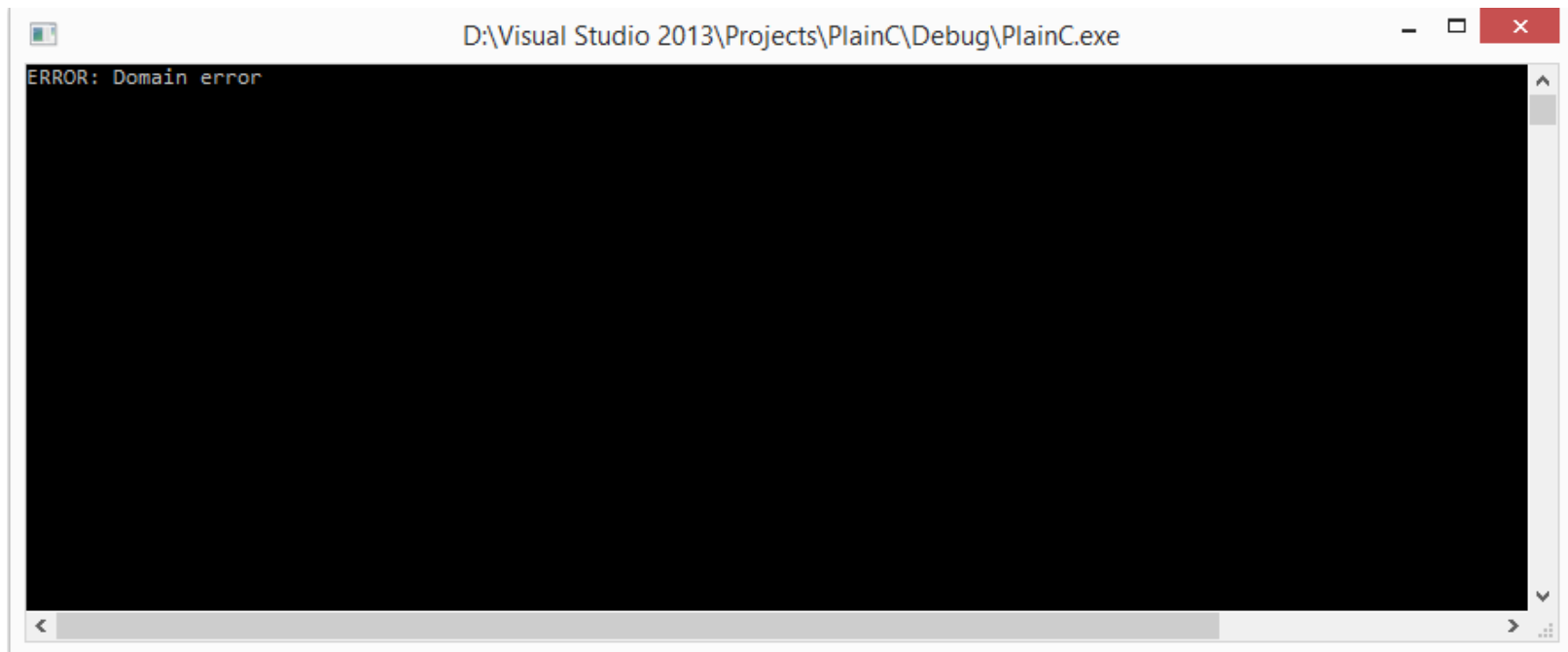
Azzero `errno` prima di chiamare la funzione da controllare.

Chiamo la funzione `sqrt()` con il parametro fuori dal dominio della funzione.

Controllo se `errno` è diverso da 0 e in tal caso stampo l'errore corrispondente su `stderr` usando la funzione `strerror()`.

Esempio

- In questo caso ovviamente si genera un errore.
- Se controllassimo il valore di `errno` dopo la chiamata a `sqrt()` scopriremmo che vale 33.
- Stampando su `stderr` il messaggio di errore ritroveremmo sulla console quanto segue:



A screenshot of a Windows console window titled "D:\Visual Studio 2013\Projects\PlainC\Debug\PlainC.exe". The console output shows the text "ERROR: Domain error" in red. The window has a standard Windows title bar with minimize, maximize, and close buttons. The console area is black with white text, and there are scrollbars on the right and bottom.

```
ERROR: Domain error
```