Capitolo 69

Libreria C, con qualche estensione POSIX

Complessivamente, la libreria C è ciò che consente l'uso di funzioni, macroistruzioni e macro-variabili definite dallo standard (ed eventualmente dalle estensioni presenti nel proprio contesto). Generalmente le funzioni vengono fornite già compilate all'interno di una libreria dinamica o statica (per esempio possono essere i file '`/lib/libc.so`' o '`/usr/lib/libc.a`'), ma dal punto di vista formale, la libreria standard è percepita attraverso i file di intestazione.

Per la precisione, lo standard stabilisce che si debba fare riferimento a delle «intestazioni» nel sorgente di un programma scritto in linguaggio C, ma il contesto particolare può essere tale per cui queste potrebbero non esistere fisicamente come ci si attenderebbe da un sistema operativo tradizionale. Anche per questo, nella documentazione standard ci si riferisce solo a intestazioni, senza precisare che debba trattarsi di file.

In pratica, i file di intestazione, o ciò che ne fa la funzione, sono sempre necessari e al loro interno si dichiarano le macro-variabili, le macroistruzioni e i prototipi delle funzioni, le quali normalmente sono già precompilate in un file separato. A ogni modo, di norma il compilatore è predisposto per utilizzare automaticamente i file precompilati necessari.

Nei capitoli successivi vengono descritti alcuni dei file di intestazione previsti dallo standard del linguaggio, mostrando come potrebbero essere realizzati e, in alcuni casi, anche fornendo una soluzione completa per le funzioni (gli esempi dovrebbero essere disponibili a partire da *allegati/c/*).

La libreria C viene estesa dallo standard POSIX con componenti aggiuntivi. In alcuni casi, nei capitoli successivi, si fa riferimento anche a estensioni POSIX, con le annotazioni appropriate al riguardo. Va però osservato che per scrivere un programma in linguaggio C, che abbia la massima portabilità fra sistemi operativi molto differenti tra loro, occorre evitare il più possibile le estensioni di qualunque genere.

Ciò che non si vede negli esempi dei capitoli successivi è la tecnica comune che si usa per evitare di includere ricorsivamente lo stesso file di intestazione più volte: si associa a ogni file una macro-variabile e se all'inizio della lettura questa non risulta dichiarata, il

contenuto viene acquisito, altrimenti viene ignorato semplicemente, perché deve essere già stato incluso in precedenza. L'esempio seguente riguarda il file '`limits.h`':

```
#ifndef _LIMITS_H
#define _LIMITS_H       1
    …
    contenuto_del_file
    …
#endif // _LIMITS_H
```

In pratica viene verificato se la macro-variabile ***_LIMITS_H*** è già stata definita; se lo è, il contenuto del file viene ignorato. Se invece la macro-variabile non è stata dichiarata, questa allora viene dichiarata e quindi si procede con il lavoro normale del file.

Tabella 69.2. File di intestazione standard.

Intestazione	Descrizione	Riferimenti
`assert.h`	Verifica diagnostica di un'espressione (asserzione).	sezione 69.2
`complex.h`	Aritmetica complessa.	--
`ctype.h`	Classificazione dei caratteri.	sezione 69.7
`errno.h`	Definizione degli errori.	sezione 69.5
`fenv.h`	Gestione di valori in virgola mobile.	--
`float.h`	Limiti dei valori in virgola mobile.	--
`inttypes.h`	Estensione di '`stdint.h`'.	sezione 69.10
`iso646.h`	Macro-variabili da usare in sostituzione di vari operatori.	sezione 69.11
`limits.h`	Limiti per i numeri interi.	sezione 69.3
`locale.h`	Gestione della configurazione locale (nel senso di «localizzazione»).	sezione 69.6
`math.h`	Funzioni matematiche comuni.	--
`setjmp.h`	Funzionalità per il salto incondizionato.	--
`signal.h`	Gestione dei segnali.	sezione 69.15
`stdarg.h`	Gestione degli argomenti variabili.	sezione 69.8
`stdbool.h`	Tipo e valori booleani.	sezione 69.12
`stddef.h`	Definizioni comuni; in particolare i tipi '**`size_t`**', '**`wchar_t`**' e il puntatore nullo '**`NULL`**'.	sezione 69.13
`stdint.h`	Definizioni di interi con un rango prestabilito, assieme ai valori minimi e massimi.	sezione 69.4
`stdio.h`	Gestione di input e output dei dati.	sezione 69.17
`stdlib.h`	Funzioni, macro e tipi di utilità generale.	sezione 69.9
`string.h`	Gestione delle stringhe.	sezione 69.14
`tgmath.h`	macroistruzioni matematiche, indipendenti dal tipo.	--
`time.h`	Gestione di date e orari.	sezione 69.16
`wchar.h`	Gestione facilitata di caratteri estesi.	--
`wctype.h`	Classificazione dei caratteri estesi.	--

69.1 Funzionalità di libreria non dichiarate

Può succedere che il compilatore, per assolvere a funzionalità che figurano essere indipendenti da librerie, debba invece avvalersi di funzioni esterne che non sono previste dallo standard. In particolare, questo problema può verificarsi di fronte alla necessità di svolgere calcoli al di fuori della portata normale del microprocessore.

A titolo di esempio, il compilatore GNU C per la piattaforma x86-32 prevede un tipo intero '**`long long int`**' da 64 bit. Quando si vuole ottenere una divisione intera o il resto di una divisione con variabili di questo tipo, il compilatore GNU C richiama rispettivamente le funzioni ***__udivdi3()*** e ***__umoddi3()***. In generale il problema non si avverte, ma se si vuole scrivere la propria libreria C, senza tali funzioni, in pratica non è possibile usare questo tipo intero molto grande.

Si vedano eventualmente i listati della sezione 95.2, relativi a os32, in cui si realizzano queste funzioni con il solo ausilio del linguaggio C.

69.2 File «assert.h»

Il file '`assert.h`' della libreria standard definisce la macroistruzione ***assert()***, da usare per generare informazioni diagnostiche, sulla base dell'esito della valutazione di un'espressione.

La macroistruzione ***assert()*** viene definita in due modi alternativi, in base alla presenza o meno della macro-variabile ***NDEBUG***. Per la precisione, in presenza della macro-variabile ***NDEBUG*** la macroistruzione ***assert()*** deve risultare inerte.

```
#include <stdio.h>
#ifdef NDEBUG
#define assert(ignore) ((void)0)
#else
#define assert(ASSERTION)                                \
      ({if ((ASSERTION)==0)                              \
        fprintf (stderr,                                 \
                 "Assertion failed: " # ASSERTION        \
                 ", function %s, file %s, line %u.\n", \
                 __func__, __FILE__, __LINE__);})
#endif
```

69.2.1 Utilizzo

La macroistruzione ***assert()*** va usata con la sintassi seguente, dove il parametro indica un'espressione di tipo non specificato, purché di tipo scalare:

```
void assert (espressione);
```

Se l'espressione si traduce in un valore *Falso*, ovvero pari a zero, la macroistruzione emette, attraverso lo standard error, un messaggio contenente l'espressione stessa e altre indicazioni. Precisamente, oltre all'espressione deve apparire: il nome della funzione in cui ci si trova, il nome del file (sorgente) e il numero della riga.

Tuttavia, se la macro-variabile ***NDEBUG*** risulta definita, prima dell'inclusione del file '`assert.h`', la macroistruzione ***assert()*** deve essere trasformata dal compilatore come un'istruzione inerte, ovvero l'equivalente di '**`((void) 0)`**'.

Segue un l'esempio di un programma completo in cui si utilizza ***assert()***:

```
#include <stdio.h>
int
main (int argc, char *argv[])
{
    assert (123==124);
    return 0;
}
```

L'espressione verificata da ***assert()*** non può essere vera, pertanto, se non è stata dichiarata la macro-variabile ***NDEBUG***, questo programma dovrebbe produrre un messaggio come quello seguente:

```
Assertion failed: 123==124, function main, file assert.c,
line 5.
```

69.3 File «limits.h»

Il file '`limits.h`' della libreria standard definisce delle macro-variabili che riepilogano i limiti dei valori rappresentabili con le variabili scalari intere. Lo standard prescrive dei limiti minimi per la

conformità, ma le realizzazioni comuni consentono mediamente di rappresentare valori più grandi (in senso assoluto), a parità di tipo di intero. Infatti, i limiti effettivi dipendono principalmente dalla dimensione della parola del microprocessore e dal modo in cui si rappresentano i valori negativi. Si può osservare che nelle architetture comuni, in cui i valori negativi si rappresentano con il complemento a due, il valore negativo più grande (in senso assoluto) di una variabile è pari a una unità in più rispetto al valore positivo massimo (per esempio il tipo '**signed char**' va solitamente da -128 a 127).

L'esempio proposto si riferisce a un'architettura a 32 bit con i valori negativi rappresentati attraverso il complemento a due.

```
#define CHAR_UNSIGNED   0

#define CHAR_BIT        8

#define SCHAR_MIN       (-0x80)
#define SCHAR_MAX         0x7F
#define UCHAR_MAX         0xFF

#ifdef CHAR_UNSIGNED
#   define CHAR_MIN     0
#   define CHAR_MAX     UCHAR_MAX
#else
#   define CHAR_MIN     SCHAR_MIN
#   define CHAR_MAX     SCHAR_MAX
#endif

#define MB_LEN_MAX      16

#define SHRT_MIN        (-0x8000)
#define SHRT_MAX          0x7FFF
#define USHRT_MAX         0xFFFF

#define INT_MIN         (-0x80000000)
#define INT_MAX           0x7FFFFFFF
#define UINT_MAX          0xFFFFFFFFU

#define LONG_MIN        (-0x80000000L)
#define LONG_MAX          0x7FFFFFFFL
#define ULONG_MAX         0xFFFFFFFFUL

#define LLONG_MIN       (-0x8000000000000000LL)
#define LLONG_MAX         0x7FFFFFFFFFFFFFFFLL
#define ULLONG_MAX        0xFFFFFFFFFFFFFFFFULL
```

Tabella 69.7. Macro-variabili standard per la rappresentazione dei limiti riferiti a variabili scalari intere.

Macro-variabile	Descrizione
`CHAR_BIT`	Indica la quantità di bit utilizzata per rappresentare il tipo '**char**', con o senza segno. In altri termini è l'unità di memorizzazione più piccola con cui si può gestire l'insieme di caratteri minimo. Di norma si tratta di 8 bit.
`SCHAR_MIN` `SCHAR_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile '**signed char**'.
`UCHAR_MAX`	Indica il valore massimo rappresentabile in una variabile '**unsigned char**'. Il valore minimo è zero.
`CHAR_MIN` `CHAR_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile '**char**'. Questi valori dipendono dal fatto che il tipo '**char**' sia da intendere equivalente a un tipo '**unsigned char**' o '**signed char**', da cui ereditano i limiti.
`MB_LEN_MAX`	Indica la quantità massima di byte che possono essere usati per rappresentare un carattere multibyte, qualunque sia la configurazione locale.
`SHRT_MIN` `SHRT_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile '**short int**'.
`USHRT_MAX`	Indica il valore massimo rappresentabile in una variabile '**unsigned short int**'. Il valore minimo è zero.
`INT_MIN` `INT_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile '**int**'.

Macro-variabile	Descrizione
`UINT_MAX`	Indica il valore massimo rappresentabile in una variabile '**unsigned int**'. Il valore minimo è zero.
`LONG_MIN` `LONG_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile '**long int**'.
`ULONG_MAX`	Indica il valore massimo rappresentabile in una variabile '**unsigned long int**'. Il valore minimo è zero.
`LLONG_MIN` `LLONG_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile '**long long int**'.
`ULLONG_MAX`	Indica il valore massimo rappresentabile in una variabile '**unsigned long long int**'. Il valore minimo è zero.

Eventualmente si veda la realizzazione di questo file nei sorgenti di os32 (listato 95.1.6).

69.3.1 Confronto tra architetture

Per avere un'idea di come potrebbero svilupparsi i valori del file '`limits.h`' tra le varie architetture, viene mostrata una tabella in cui si possono paragonare quelli minimi stabiliti dallo standard con quelli usati nei sistemi GNU/Linux con architetture x86-32 e x86-64. Per semplicità si indicano solo i valori senza segno:

Macro-variabile	Standard	GNU/Linux x86-32	GNU/Linux x86-64
`UCHAR_MAX`	2^8-1	2^8-1	2^8-1
`USHRT_MAX`	$2^{16}-1$	$2^{16}-1$	$2^{16}-1$
`UINT_MAX`	$2^{16}-1$	$2^{32}-1$	$2^{32}-1$
`ULONG_MAX`	$2^{32}-1$	$2^{32}-1$	$2^{64}-1$
`ULLONG_MAX`	$2^{64}-1$	$2^{64}-1$	$2^{128}-1$

69.3.2 Estensioni POSIX

Per lo standard POSIX, il file '`limits.h`' serve anche per annotare limiti numerici relativi al funzionamento del sistema operativo, come per esempio la quantità massima di file aperti simultaneamente per ogni processo.

Un gruppo di macro-variabili definite nel file '`limits.h`', caratterizzate per avere il prefisso ***_POSIX_***..., ***_POSIX2_***... e ***_XOPEN_***..., definisce dei limiti minimi di compatibilità con lo standard. Per esempio, la macro-variabile ***_POSIX_LINK_MAX*** deve tradursi nel numero 8 e stabilisce che deve essere consentita la creazione di almeno otto collegamenti fisici per ogni file, in qualunque sistema POSIX.

Un secondo gruppo di macro-variabili definite nel file '`limits.h`', dichiara i limiti massimi effettivi del sistema, riconducibili ai minimi già fissati nel primo gruppo già descritto. Per esempio, la macro-variabile ***LINK_MAX*** indica il numero massimo effettivo di collegamenti fisici per file, tenendo conto che deve essere necessariamente maggiore o uguale al valore di ***_POSIX_LINK_MAX***.

Le macro-variabili del secondo gruppo sono facoltative, in quanto i limiti effettivi del sistema, per le varie voci, possono dipendere da fattori dinamici di funzionamento. In ogni caso, devono essere garantiti i valori minimi delle macro-variabili del primo gruppo.

Nell'ambito delle dichiarazioni che fanno già parte dello standard C, va osservato che lo standard POSIX richiede che il byte sia esattamente di 8 bit, pertanto la macro-variabile ***CHAR_BIT*** deve tradursi necessariamente nel numero otto. Inoltre, si aggiungono anche qui alcune macro-variabili:

```
#define WORD_BIT   32
#define LONG_BIT   32
#define SSIZE_MAX  0x7FFFFFFFL
```

Tabella 69.10. Alcune macro-variabili aggiunte dallo standard POSIX.

Macro-variabile	Descrizione
`WORD_BIT`	Rappresenta la quantità di bit utilizzata per rappresentare il tipo '**`int`**', con o senza segno. Il minimo valore accettabile è 32.
`LONG_BIT`	Rappresenta la quantità di bit utilizzata per rappresentare il tipo '**`long int`**', con o senza segno. Il minimo valore accettabile è 32.
`SSIZE_MAX`	Rappresenta il valore positivo massimo che possa esprimere una variabile di tipo '**`ssize_t`**' (un tipo come '**`size_t`**', ma con segno). Il minimo valore accettabile è quello della macro-variabile ***_POSIX_SSIZE_MAX***, ovvero $7FFF_{16}$.

69.4 File «stdint.h»

Il file '`stdint.h`' della libreria standard definisce principalmente dei tipi interi, alternativi a quelli tradizionali, riferiti in modo più diretto al rango. Assieme a questi tipi interi definisce anche delle macro-variabili che consentono di conoscere i limiti esatti di tali tipi, oltre ad altre macro-variabili con i limiti di tipi interi speciali, dichiarati in altri file (si veda eventualmente la realizzazione di questo file nei sorgenti di os32, listato 95.1.13).

Lo standard prescrive che alcuni dei tipi definiti nel file '`stdint.h`' siano opzionali, purché sia rispettato un certo ordine (per esempio, se viene definito un tipo intero con segno, deve essere prevista anche una versione di quel tipo senza segno e viceversa). A tale proposito, le macro-variabili con le quali si possono verificare i limiti, servono anche per consentire al programmatore di verificare la disponibilità o meno del tipo relativo, attraverso istruzioni del precompilatore del tipo '**`#ifdef`**'.

L'esempio proposto si riferisce a un elaboratore x86-32 ed è abbastanza conforme alla configurazione che si può trovare in un sistema GNU/Linux.

69.4.1 Tipi interi ad ampiezza esatta

Lo standard prescrive un gruppo facoltativo di tipi interi il cui rango è definito precisamente dal nome. Si tratta dei tipi '**`int`*`n`*`_t`**' (con segno) e '**`uint`*`n`*`_t`**' (senza segno), dove ***n*** esprime la quantità di bit che compone l'intero. Si tratta necessariamente di tipi facoltativi, perché non è possibile stabilire in modo sicuro che in ogni architettura siano gestibili tipi interi di una data quantità di bit; per esempio, in una certa achitettura «X» potrebbero essere gestiti tipi interi a 8, 16 e 32 bit, mentre in un'architettura «Y» i tipi disponibili effettivamente potrebbero essere a 8, 16, 24 e 32 bit.

Nei sistemi POSIX, questi tipi sono invece obbligatori, costringendo così ad avere byte esattamente di otto bit.

```
typedef signed char             int8_t;
typedef short int               int16_t;
typedef int                     int32_t;        // x86-32
typedef long long int           int64_t;        // x86-32

typedef unsigned char           uint8_t;
typedef unsigned short int      uint16_t;
typedef unsigned int            uint32_t;       // x86-32
typedef unsigned long long int  uint64_t;       // x86-32
```

Le macro-variabili usate per definire i limiti di questi valori interi hanno nomi del tipo '**`INT`*`n`*`_MIN`**', '**`INT`*`n`*`_MAX`**' e '**`UINT`*`n`*`_MAX`**', per indicare rispettivamente: il valore minimo dei tipi con segno; il valore massimo dei tipi con segno; il valore massimo dei tipi senza segno. Lo standard prescrive precisamente questi valori minimi e massimi, intendendo implicitamente che i valori negativi si rappresentino con il complemento a due:

Macro-variabile	Valore standard
`INT`***`n`***`_MIN`	$-(2^{n-1})$
`INT`***`n`***`_MAX`	$2^{n-1}-1$
`UINT`***`n`***`_MAX`	$2^{n}-1$

```
#define INT8_MIN                (-0x80)
#define INT16_MIN               (-0x8000)
#define INT32_MIN               (-0x80000000)
#define INT64_MIN               (-0x8000000000000000LL)

#define INT8_MAX                  0x7F
#define INT16_MAX                 0x7FFF
#define INT32_MAX                 0x7FFFFFFF
#define INT64_MAX                 0x7FFFFFFFFFFFFFFFLL

#define UINT8_MAX                 0xFF
#define UINT16_MAX                0xFFFF
#define UINT32_MAX                0xFFFFFFFFU
#define UINT64_MAX                0xFFFFFFFFFFFFFFFFULL
```

69.4.2 Tipi interi di rango minimo

Un gruppo richiesto espressamente dallo standard riguarda tipi interi il cui rango sia tale da garantire la rappresentazione di almeno ***n*** bit, utilizzando comunque la quantità minima possibile di bit. In questo caso i nomi sono '**`int_least`*`n`*`_t`**' per i tipi con segno e '**`uint_least`*`n`*`_t`**' per quelli senza segno. Lo standard prescrive che siano previsti necessariamente i tipi a 8, 16, 32 e 64 bit, mentre ammette che ne siano disponibili anche altri.

```
typedef signed char             int_least8_t;
typedef short int               int_least16_t;
typedef int                     int_least32_t;  // x86-32
typedef long long int           int_least64_t;  // x86-32

typedef unsigned char           uint_least8_t;
typedef unsigned short int      uint_least16_t;
typedef unsigned int            uint_least32_t; // x86-32
typedef unsigned long long int  uint_least64_t; // x86-32
```

Le macro-variabili usate per definire i limiti di questi valori interi hanno nomi del tipo '**`INT_LEAST`*`n`*`_MIN`**', '**`INT_LEAST`*`n`*`_MAX`**' e '**`UINT_LEAST`*`n`*`_MAX`**', per indicare rispettivamente: il valore minimo dei tipi con segno; il valore massimo dei tipi con segno; il valore massimo dei tipi senza segno. Lo standard attribuisce questi limiti in modo indipendente dalla rappresentazione dei valori negativi, ma tali limiti possono essere estesi in senso assoluto:

Macro-variabile	Valore standard che può essere superato in termini assoluti
`INT_LEAST`***`n`***`_MIN`	$-(2^{n-1}-1)$
`INT_LEAST`***`n`***`_MAX`	$2^{n-1}-1$
`UINT_LEAST`***`n`***`_MAX`	$2^{n}-1$

```
#define INT_LEAST8_MIN          (-0x80)
#define INT_LEAST16_MIN         (-0x8000)
#define INT_LEAST32_MIN         (-0x80000000)
#define INT_LEAST64_MIN         (-0x8000000000000000LL)

#define INT_LEAST8_MAX            0x7F
#define INT_LEAST16_MAX           0x7FFF
#define INT_LEAST32_MAX           0x7FFFFFFF
#define INT_LEAST64_MAX           0x7FFFFFFFFFFFFFFFLL

#define UINT_LEAST8_MAX           0xFF
#define UINT_LEAST16_MAX          0xFFFF
#define UINT_LEAST32_MAX          0xFFFFFFFFU
#define UINT_LEAST64_MAX          0xFFFFFFFFFFFFFFFFULL
```

Ai tipi interi di rango minimo sono associate anche delle macroistruzioni, il cui scopo è quello di consentire la rappresentazione corretta dei valori costanti:

```
INTn_C(valore)
```

```
UINTn_C(valore)
```

In pratica, per indicare il valore costante 1234567890, precisando che va inteso come un tipo '**uint_least64_t**', si deve scrivere: '**UINT64_C(1234567890)**'. Il valore costante in sé, può essere espresso in qualunque modo, purché sia ammissibile nel contesto comune.

```
#define INT8_C(VAL)     VAL
#define INT16_C(VAL)    VAL
#define INT32_C(VAL)    VAL             // x86-32
#define INT64_C(VAL)    VAL ## LL       // x86-32

#define UINT8_C(VAL)    VAL
#define UINT16_C(VAL)   VAL
#define UINT32_C(VAL)   VAL ## U        // x86-32
#define UINT64_C(VAL)   VAL ## ULL      // x86-32
```

È evidente che, nel caso dell'esempio mostrato, '**UINT64_C(1234567890)**' corrisponde a '**1234567890ULL**'.

69.4.3 Tipi interi «veloci»

Un altro gruppo di tipi richiesti dallo standard è quello il cui rango è tale da consentire la rappresentazione di almeno ***n*** bit, utilizzando la quantità minima di bit che garantisce tempi ottimali di elaborazione. In questo caso i nomi sono '**int_fastn_t**' per i tipi con segno e '**uint_fastn_t**' per quelli senza segno. Lo standard prescrive che siano previsti necessariamente i tipi a 8, 16, 32 e 64 bit, mentre ammette che ne siano disponibili anche altri.

```
typedef signed char             int_fast8_t;
typedef int                     int_fast16_t;   // x86-32
typedef int                     int_fast32_t;   // x86-32
typedef long long int           int_fast64_t;   // x86-32

typedef unsigned char           uint_fast8_t;
typedef unsigned int            uint_fast16_t;  // x86-32
typedef unsigned int            uint_fast32_t;  // x86-32
typedef unsigned long long int  uint_fast64_t;  // x86-32
```

Come suggerisce l'esempio, è ragionevole pensare che, dove possibile, il rango usato effettivamente sia quello del tipo intero normale.

Le macro-variabili usate per definire i limiti di questi valori interi hanno nomi del tipo '**INT_FASTn_MIN**', '**INT_FASTn_MAX**' e '**UINT_FASTn_MAX**', per indicare rispettivamente: il valore minimo dei tipi con segno; il valore massimo dei tipi con segno; il valore massimo dei tipi senza segno. Lo standard attribuisce questi limiti in modo indipendente dalla rappresentazione dei valori negativi, ma tali limiti possono essere estesi in senso assoluto:

Macro-variabile	Valore standard che può essere superato in termini assoluti
INT_FASTn_MIN	$-(2^{n-1}-1)$
INT_FASTn_MAX	$2^{n-1}-1$
UINT_FASTn_MAX	$2^{n}-1$

```
#define INT_FAST8_MIN           (-0x80)
#define INT_FAST16_MIN          (-0x80000000)
#define INT_FAST32_MIN          (-0x80000000)
#define INT_FAST64_MIN          (-0x8000000000000000LL)

#define INT_FAST8_MAX             0x7F
#define INT_FAST16_MAX            0x7FFFFFFF
#define INT_FAST32_MAX            0x7FFFFFFF
#define INT_FAST64_MAX            0x7FFFFFFFFFFFFFFFLL

#define UINT_FAST8_MAX            0xFF
#define UINT_FAST16_MAX           0xFFFFFFFFU
#define UINT_FAST32_MAX           0xFFFFFFFFU
#define UINT_FAST64_MAX           0xFFFFFFFFFFFFFFFFULL
```


69.4.4 Tipi interi per rappresentare dei puntatori

Sono previsti due tipi opzionali interi, adatti a contenere il valore di un puntatore, garantendo che la conversione da e verso '**void ***' avvenga sempre correttamente. Per la precisione si tratta di '**intptr_t**' e '**uintptr_t**', dove il primo è un intero con segno, mentre il secondo è senza segno.

```
typedef int                     intptr_t;       // x86-32
typedef unsigned int            uintptr_t;      // x86-32
```

Le macro-variabili usate per definire i limiti di questi valori interi sono ***INTPTR_MIN***, ***INTPTR_MAX*** e ***UINTPTR_MAX***, per indicare rispettivamente: il valore minimo con segno, il valore massimo con segno e il valore massimo senza segno. Lo standard attribuisce dei limiti riferiti ad architetture in grado di indirizzare al massimo con 16 bit e ovviamente vanno adattati alla realtà dell'architettura effettiva:

Macro-variabile	Valore standard che può essere superato in termini assoluti
INTPTR_MIN	$-(2^{15}-1)$
INTPTR_MAX	$2^{15}-1$
UINTPTR_MAX	$2^{16}-1$

```
#define INTPTR_MIN              (-0x80000000)
#define INTPTR_MAX                0x7FFFFFFF
#define UINTPTR_MAX               0xFFFFFFFFU
```

69.4.5 Tipi interi di rango massimo

Per poter rappresentare in modo indipendente dall'architettura degli interi di rango massimo, sono previsti due tipi specifici, richiesti espressamente dallo standard: '**intmax_t**' e '**uintmax_t**'. Le macro-variabili che definiscono i limiti sono: ***INTMAX_MIN***, ***INTMAX_MAX*** e ***UINTMAX_MAX***. Lo standard prescrive che si tratti di variabili con un rango di almeno 64 bit.

```
typedef long long int           intmax_t;       // x86-32
typedef unsigned long long int  uintmax_t;      // x86-32
```

Macro-variabile	Valore standard che può essere superato in termini assoluti
INTMAX_MIN	$-(2^{63}-1)$
INTMAX_MAX	$2^{63}-1$
UINTMAX_MAX	$2^{64}-1$

Anche ai tipi interi di rango massimo sono associate delle macroistruzioni per facilitare la rappresentazione corretta dei valori costanti:

```
INTMAX_C(valore)
```

```
UINTMAX_C(valore)
```

In pratica, per indicare il valore costante 1234567890, precisando che va inteso come un tipo '**uintmax_t**', si deve scrivere: '**UINTMAX_C(1234567890)**'. Il valore costante in sé, può essere espresso in qualunque modo, purché sia ammissibile nel contesto comune.

```
#define INTMAX_C(VAL)   VAL ## LL       // x86-32
#define UINTMAX_C(VAL)  VAL ## ULL      // x86-32
```

A questo punto, anche la definizione dei valori minimi e massimi diventa più agevole:

```
// x86-32
#define INTMAX_MIN      (-INTMAX_C(0x8000000000000000))
#define INTMAX_MAX       (INTMAX_C(0x7FFFFFFFFFFFFFFF))
#define UINTMAX_MAX     (UINTMAX_C(0xFFFFFFFFFFFFFFFF))
```

69.4.6 Limiti per altri tipi interi

Altri tipi interi dichiarati al di fuori del file ‘`stdint.h`’ hanno i limiti definiti qui. Ne viene mostrata solo una tabella riepilogativa.

Macro-variabile	Descrizione
`PTRDIFF_MIN` `PTRDIFF_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile ‘**`ptrdiff_t`**’.
`SIG_ATOMIC_MIN` `SIG_ATOMIC_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile ‘**`sig_atomic_t`**’.
`SIZE_MAX`	Indica il valore massimo rappresentabile in una variabile ‘**`size_t`**’, la quale è destinata a contenere valori senza segno.
`WCHAR_MIN` `WCHAR_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile ‘**`wchar_t`**’.
`WINT_MIN` `WINT_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile ‘**`wint_t`**’.

L’esempio seguente riporta i valori usati in un sistema GNU/Linux con architettura x86-32, a parte il caso di ‘**`wchar_t`**’ e ‘**`wint_t`**’ che si intendono rispettivamente a 32 bit senza segno e 64 bit con segno:

```
#define PTRDIFF_MIN             (-0x80000000)   // x86-32
#define PTRDIFF_MAX               0x7FFFFFFF    // x86-32

#define SIG_ATOMIC_MIN          (-0x80000000)   // x86-32
#define SIG_ATOMIC_MAX            0x7FFFFFFF    // x86-32

#define SIZE_MAX                  0xFFFFFFFFU   // x86-32

#define WCHAR_MIN                 0x00000000
#define WCHAR_MAX                 0xFFFFFFFFU   // x86-32

#define WINT_MIN        (-0x8000000000000000LL) // x86-32
#define WINT_MAX          0x7FFFFFFFFFFFFFFFLL  // x86-32
```

69.5 File «errno.h»

Il file ‘`errno.h`’ della libreria standard definisce principalmente delle macro-variabili per rappresentare simbolicamente delle situazioni di errore. Queste macro-variabili si espandono in un numero intero e positivo, di tipo ‘**`int`**’, ma la corrispondenza tra l’errore simbolico rappresentato dalla macro-variabile e il numero in cui questa si deve espandere, dipende dalle convenzioni del sistema operativo (si veda eventualmente la realizzazione del file ‘`errno.h`’ nei sorgenti di os32, sezione 95.5, tenendo conto che lì si aggiungono delle funzioni non standard, con le quali è più facile l’individuazione della posizione del sorgente in cui l’errore si è manifestato).

Lo standard del linguaggio prescrive poche macro-variabili, da cui dipendono le librerie standard, mentre tutte le altre sono competenza delle convenzioni del sistema operativo.

Oltre alle macro-variabili che rappresentano le situazioni di errore previste, il file ‘`errno.h`’ deve dichiarare ***errno***, in qualità di espressione che si traduca in una variabile scalare. In pratica può trattarsi di una variabile esterna o di una macro-variabile che si traduce in qualunque cosa consenta di assegnarvi un valore. Il valore iniziale che si può leggere da ***errno*** è zero (con cui si intende l’assenza di qualunque tipo di situazione di errore) e viene modificato dalle funzioni che, di volta in volta, possono avere bisogno di annotare uno stato di errore.

Si osservi che i numeri che si vedono associati alle macro-variabili sono stati tratti, come esempio, dalla configurazione di un sistema GNU/Linux.

```
extern int errno;

#define EDOM    33
#define EILSEQ  84
#define ERANGE  34
```


Il nome ***errno***, in un sistema che consenta l’esecuzione di programmi suddivisi in più thread, deve tradursi in un’espressione tale da rappresentare una variabile scalare individuale per ogni thread, in modo che i thread non possano interferire tra di loro a questo proposito. Evidentemente, l’esempio mostrato non offre questa accortezza.

Tabella 69.31. Macro-variabili standard per la rappresentazione di situazioni di errore.

Macro-variabile	Descrizione
`EDOM`	Errore di dominio: l’argomento di una funzione matematica ha un valore al di fuori del dominio previsto.
`EILSEQ`	Errore di codifica: la sequenza dei byte che deve rappresentare una certa codifica contiene un errore.
`ERANGE`	Errore nell’intervallo di valori: il risultato di un’espressione matematica non può essere rappresentato nell’intervallo di valori previsto (ovvero nella variabile che deve riceverlo).

Il file di intestazione ‘`errno.h`’ di un sistema POSIX è più articolato, in quanto contiene un elenco numeroso di macro-variabili. Valgono naturalmente le stesse considerazioni per la variabile globale ***errno***, a proposito dei thread multipli. Nell’esempio successivo, i numeri associati alle varie macro-variabili non fanno riferimento ad alcun sistema operativo reale; va osservato inoltre che ogni sistema POSIX aggiunge propri tipi di errore, necessari per le proprie caratteristiche specifiche.

```
extern int errno;

#define E2BIG            1 // Argument list too long.
#define EACCES           2 // Permission denied.
#define EADDRINUSE       3 // Address in use.
#define EADDRNOTAVAIL    4 // Address not available.
#define EAFNOSUPPORT     5 // Address family not supported.
#define EAGAIN           6 // Resource unavailable, try
                           // again.
#define EALREADY         7 // Connection already in
                           // progress.
#define EBADF            8 // Bad file descriptor.
#define EBADMSG          9 // Bad message.
#define EBUSY           10 // Device or resource busy.
#define ECANCELED       11 // Operation canceled.
#define ECHILD          12 // No child processes.
#define ECONNABORTED    13 // Connection aborted.
#define ECONNREFUSED    14 // Connection refused.
#define ECONNRESET      15 // Connection reset.
#define EDEADLK         16 // Resource deadlock would occur.
#define EDESTADDRREQ    17 // Destination address required.
#define EDOM            18 // Mathematics argument out of
                           // domain of function.
#define EDQUOT          19 // Reserved.
#define EEXIST          20 // File exists.
#define EFAULT          21 // Bad address.
#define EFBIG           22 // File too large.
#define EHOSTUNREACH    23 // Host is unreachable.
#define EIDRM           24 // Identifier removed.
#define EILSEQ          25 // Illegal byte sequence.
#define EINPROGRESS     26 // Operation in progress.
#define EINTR           27 // Interrupted function.
#define EINVAL          28 // Invalid argument.
#define EIO             29 // I/O error.
#define EISCONN         30 // Socket is connected.
#define EISDIR          31 // Is a directory.
#define ELOOP           32 // Too many levels of symbolic
                           // links.
#define EMFILE          33 // Too many open files.
#define EMLINK          34 // Too many links.
#define EMSGSIZE        35 // Message too large.
#define EMULTIHOP       36 // Reserved.
#define ENAMETOOLONG    37 // Filename too long.
#define ENETDOWN        38 // Network is down.
#define ENETRESET       39 // Connection aborted by network.
#define ENETUNREACH     40 // Network unreachable.
#define ENFILE          41 // Too many files open in system.
#define ENOBUFS         42 // No buffer space available.
```

```
#define ENODATA         43 // No message is available on the
                           // stream head read queue.
#define ENODEV          44 // No such device.
#define ENOENT          45 // No such file or directory.
#define ENOEXEC         46 // Executable file format error.
#define ENOLCK          47 // No locks available.
#define ENOLINK         48 // Reserved.
#define ENOMEM          49 // Not enough space.
#define ENOMSG          50 // No message of the desired
                           // type.
#define ENOPROTOOPT     51 // Protocol not available.
#define ENOSPC          52 // No space left on device.
#define ENOSR           53 // No stream resources.
#define ENOSTR          54 // Not a stream.
#define ENOSYS          55 // Function not supported.
#define ENOTCONN        56 // The socket is not connected.
#define ENOTDIR         57 // Not a directory.
#define ENOTEMPTY       58 // Directory not empty.
#define ENOTSOCK        59 // Not a socket.
#define ENOTSUP         60 // Not supported.
#define ENOTTY          61 // Inappropriate I/O control
                           // operation.
#define ENXIO           62 // No such device or address.
#define EOPNOTSUPP      63 // Operation not supported on
                           // socket.
#define EOVERFLOW       64 // Value too large to be stored
                           // in data type.
#define EPERM           65 // Operation not permitted.
#define EPIPE           66 // Broken pipe.
#define EPROTO          67 // Protocol error.
#define EPROTONOSUPPORT 68 // Protocol not supported.
#define EPROTOTYPE      69 // Protocol wrong type for
                           // socket.
#define ERANGE          70 // Result too large.
#define EROFS           71 // Read-only file system.
#define ESPIPE          72 // Invalid seek.
#define ESRCH           73 // No such process.
#define ESTALE          74 // Reserved.
#define ETIME           75 // Stream ioctl() timeout.
#define ETIMEDOUT       76 // Connection timed out.
#define ETXTBSY         77 // Text file busy.
#define EWOULDBLOCK     78 // Operation would block (may
                           // be the same as EAGAIN).
#define EXDEV           79 // Cross-device link.
```

69.6 File «locale.h»

Il file '`locale.h`' della libreria standard definisce delle macro-variabili, un tipo di struttura e alcune funzioni, relative alla gestione della configurazione locale del programma. Se non si fa uso di tale configurazione, il proprio programma opera in quella che è nota come «configurazione locale C», ovvero il minimo indispensabile.

Nell'ambito di un sistema operativo Unix o simile, la configurazione locale avviene principalmente attraverso l'impostazione di variabili di ambiente il cui nome inizia per '**`LC_`**…'. Tuttavia, questa configurazione non viene ereditata automaticamente dal programma scritto in linguaggio C, perché questo deve acquisirla espressamente, se vuole.

La definizione della configurazione locale avviene attraverso una stringa contenente delle sigle che esprimono la lingua, la nazionalità e la codifica da utilizzare per rappresentare i caratteri. Per esempio, '**`it_CH.UTF-8`**' rappresenta la lingua italiana, la nazionalità svizzera e la codifica UTF-8.

69.6.1 Impostazione della configurazione locale

Per modificare l'impostazione della configurazione locale del proprio programma, si utilizza la funzione ***setlocale()*** che richiede l'indicazione di un numero intero, a rappresentare la ***categoria*** nella quale intervenire. La categoria viene definita formalmente attraverso delle macro-variabili il cui nome inizia per ***LC_***… e si tratta degli stessi nomi usati nelle variabili di ambiente di un sistema Unix o simile. Viene proposto un esempio di dichiarazione delle macro-variabili indispensabili, ma l'associazione al numero varia molto da un sistema all'altro:


```
LC_ALL          0
LC_COLLATE      1
LC_CTYPE        2
LC_MONETARY     3
LC_NUMERIC      4
LC_TIME         5
```

La macro variabile successiva è un'estensione usata nei sistemi POSIX:

```
LC_MESSAGES     6
```

La funzione ***setlocale()*** con cui si cambia la configurazione locale ha il prototipo seguente:

```
char *setlocale (int category, const char *locale);
```

Si prevedono due situazioni diverse di utilizzo della funzione. Per cominciare può essere usata per interrogare la configurazione attuale, come nell'esempio seguente:

```
…
char *p;
p = setlocale (LC_COLLATE, NULL);
…
```

Fornendo un puntatore nullo, al posto della stringa che deve indicare la configurazione locale, si ottiene un puntatore alla stringa che descrive quella attuale. In questo caso, se si utilizza la categoria '**`LC_ALL`**', si ottiene una stringa che descrive tutte le altre categorie, ammesso che ci siano delle differenze. Se invece la funzione non è in grado di dare questa informazione, si ottiene semplicemente un puntatore nullo.

Naturalmente, la funzione serve anche per cambiare la configurazione locale, specificando in tal caso la stringa che la descrive. Per esempio, nel modo seguente si interviene nella categoria '**`LC_NUMERIC`**':

```
…
char *p;
p = setlocale (LC_NUMERIC, "it_IT.UTF-8");
…
```

Anche in questo caso si ottiene un puntatore che descrive la categoria scelta, ma se l'operazione fallisce, si ottiene invece il puntatore nullo.

Normalmente è più probabile che, nell'impostazione della configurazione locale si voglia indicare una modalità unica per tutte le categorie; pertanto, in tal caso va usato '**`LC_ALL`**':

```
…
p = setlocale (LC_ALL, "it_IT.UTF-8");
…
```

Viene mostrato un esempio di programma completo, in cui si imposta prima la configurazione locale complessiva, poi se ne cambia una e quindi si interroga la situazione:

```
#include <stdio.h>
#include <locale.h>

int main (void)
{
    setlocale (LC_ALL, "it_IT.UTF-8");
    setlocale (LC_MONETARY, "en_US.UTF-8");

    printf ("LC_COLLATE:  \"%s\"\n",
            setlocale (LC_COLLATE, NULL));
    printf ("LC_CTYPE:    \"%s\"\n",
            setlocale (LC_CTYPE, NULL));
    printf ("LC_MONETARY: \"%s\"\n",
            setlocale (LC_MONETARY, NULL));
    printf ("LC_NUMERIC:  \"%s\"\n",
            setlocale (LC_NUMERIC, NULL));
    printf ("LC_TIME:     \"%s\"\n",
            setlocale (LC_TIME, NULL));
    printf ("LC_ALL:      \"%s\"\n",
            setlocale (LC_ALL, NULL));
#ifdef LC_MESSAGES
    printf ("LC_MESSAGES: \"%s\"\n",
            setlocale (LC_MESSAGES, NULL));
#endif

    return 0;
}
```

Ecco cosa si può ottenere:

```
LC_COLLATE:  "it_IT.UTF-8"
LC_CTYPE:    "it_IT.UTF-8"
LC_MONETARY: "en_US.UTF-8"
LC_NUMERIC:  "it_IT.UTF-8"
LC_TIME:     "it_IT.UTF-8"
LC_ALL:      "LC_CTYPE=it_IT.UTF-8;LC_NUMERIC=it_IT.UTF-8;↩
↪LC_TIME=it_IT.UTF-8;LC_COLLATE=it_IT.UTF-8;LC_MONETARY=en_US.UTF-8
↪LC_MESSAGES=it_IT.UTF-8"
LC_MESSAGES: "it_IT.UTF-8"
```

Per fare sì che il programma erediti la configurazione locale dal contesto in cui si trova a funzionare (quindi dalla configurazione locale del sistema operativo), si può indicare la stringa nulla al posto della definizione:

```
…
p = setlocale (LC_ALL, "");
…
```

Pertanto, questo è il modo appropriato per iniziare la configurazione all'interno di un programma.

69.6.2 Composizione dei valori numerici

Il modo in cui si rappresenta testualmente un valore numerico, con o senza indicazione della valuta (la moneta), dipende dalla configurazione locale. La funzione ***localeconv()*** restituisce il puntatore a una struttura che contiene i dettagli riguardo alle modalità di rappresentazione dei valori numerici, secondo la configurazione locale. L'utilizzo consentito di questa struttura si limita all'interrogazione dei valori, perché la modifica dipende dalla gestione della configurazione locale.

```
struct lconv {char *decimal_point;
              char *thousands_sep;
              char *grouping;
              char *mon_decimal_point;
              char *mon_thousands_sep;
              char *mon_grouping;
              char *positive_sign;
              char *negative_sign;
              char *currency_symbol;
              char frac_digits;
              char p_cs_precedes;
              char n_cs_precedes;
              char p_sep_by_space;
              char n_sep_by_space;
              char p_sign_posn;
              char n_sign_posn;
              char *int_curr_symbol;
              char int_frac_digits;
              char int_p_cs_precedes;
              char int_n_cs_precedes;
              char int_p_sep_by_space;
              char int_n_sep_by_space;
              char int_p_sign_posn;
              char int_n_sign_posn;
};
```


A titolo di esempio vengono descritti solo alcuni membri della struttura; per gli altri si deve consultare la documentazione dello standard:

Membro	Descrizione
`decimal_point`	Stringa contenente il carattere usato per separare la parte decimale in un numero per uso generale.
`thousand_sep`	Stringa contenente il carattere usato per separare le cifre della parte intera di un numero per uso generale.

`positive_sign`
 	
`negative_sign`
	Stringa contenente il carattere usato per rappresentare il segno, positivo o negativo, di un numero usato per le valute.

`int_curr_symbol`	Stringa composta da quattro caratteri, di cui i primi tre indicano la sigla internazionale della valuta ('**`USD`**', '**`EUR`**', ecc.) e il quarto è solo un carattere di separazione da usare tra tale sigla e il valore numerico a cui questa si riferisce.

I membri che rappresentano delle stringhe (puntatori a carattere), quando si riferiscono a dati facoltativi, possono essere vuoti (nel senso di stringhe nulle). I membri di tipo '**`char`**' vengono usati in modo numerico.

```
struct lconv *localeconv (void);
```

Come si vede dal prototipo, la funzione ***localeconv()*** serve esclusivamente per ottenere il puntatore alla struttura '**`lconv`**', da usare per la sua consultazione. Viene mostrato un esempio molto semplice per il suo utilizzo:

```
#include <stdio.h>
#include <locale.h>

int main (void)
{
    struct lconv *lc;

    setlocale (LC_ALL, "it_IT.UTF-8");
    lc = localeconv ();

    printf ("decimal_point:\t\t\"%s\"\n",
            lc->decimal_point);
    printf ("thousand_sep:\t\t\"%s\"\n",
            lc->thousands_sep);
    printf ("grouping:\t\t\"%s\"\n",
            lc->grouping);
    printf ("mon_decimal_point:\t\"%s\"\n",
            lc->mon_decimal_point);
    printf ("mon_thousands_sep:\t\"%s\"\n",
            lc->mon_thousands_sep);
    printf ("mon_grouping:\t\t\"%s\"\n",
            lc->mon_grouping);
    printf ("positive_sign:\t\t\"%s\"\n",
            lc->positive_sign);
    printf ("negative_sign:\t\t\"%s\"\n",
            lc->negative_sign);
    printf ("currency_symbol:\t\"%s\"\n",
            lc->currency_symbol);              // Multibyte.
    printf ("frac_digits:\t\t%i\n",
            lc->frac_digits);
    printf ("p_cs_precedes:\t\t%i\n",
            lc->p_cs_precedes);
    printf ("n_cs_precedes:\t\t%i\n",
            lc->n_cs_precedes);
    printf ("p_sep_by_space:\t\t%i\n",
            lc->p_sep_by_space);
    printf ("n_sep_by_space:\t\t%i\n",
            lc->n_sep_by_space);
    printf ("p_sign_posn:\t\t%i\n",
            lc->p_sign_posn);
    printf ("n_sign_posn:\t\t%i\n",
```

```
            lc->n_sign_posn);
    printf ("int_curr_symbol:\t\"%s\"\n",
            lc->int_curr_symbol);
    printf ("int_frac_digit:\t\t%i\n",
            lc->int_frac_digit);
    printf ("int_p_cs_precedes:\t\t%i\n",
            lc->int_p_cs_precedes);
    printf ("int_n_cs_precedes:\t\t%i\n",
            lc->int_n_cs_precedes);
    printf ("int_p_sep_by_space:\t\t%i\n",
            lc->int_p_sep_by_space);
    printf ("int_n_sep_by_space:\t\t%i\n",
            lc->int_n_sep_by_space);
    printf ("int_p_sign_posn:\t\t%i\n",
            lc->int_p_sign_posn);
    printf ("int_n_sign_posn:\t\t%i\n",
            lc->int_n_sign_posn);

    return 0;
}
```

Il risultato che si ottiene dovrebbe essere molto simile a quello seguente:

```
decimal_point:          ","
thousand_sep:           ""
grouping:               ""
mon_decimal_point:      ","
mon_thousands_sep:      "."
mon_grouping:           ""
positive_sign:          ""
negative_sign:          "-"
currency_symbol:        "€"
frac_digits:            2
p_cs_precedes:          1
n_cs_precedes:          1
p_sep_by_space:         1
n_sep_by_space:         1
p_sign_posn:            1
n_sign_posn:            1
int_curr_symbol:        "EUR "
int_frac_digit:         2
int_p_cs_precedes:      1
int_n_cs_precedes:      1
int_p_sep_by_space:     1
int_n_sep_by_space:     1
int_p_sign_posn:        1
int_n_sign_posn:        4
```

Si osservi che, nell'esempio, la stringa a cui si accede tramite il membro '**`currency_symbol`**' è una sequenza «multibyte», nel senso che utilizza più byte per rappresentare un solo carattere.

69.7 File «ctype.h»

Il file '`ctype.h`' della libreria standard definisce alcune funzioni per la classificazione e la trasformazione dei caratteri (intesi come '**`char`**'). Gli esempi proposti qui riguardano esclusivamente l'insieme di caratteri corrispondente alla codifica ASCII e, di conseguenza, la configurazione locale '**`C`**'. Tuttavia va ricordato che il linguaggio C non impone che l'insieme di caratteri minimo sia descritto attraverso la codifica ASCII, mentre così è invece nello standard POSIX. (si veda eventualmente la realizzazione del file '`ctype.h`' nei sorgenti di os32, listato 95.1.5).

Le funzioni di questo file hanno in comune il parametro, costituito da un valore intero di tipo '**`int`**', usato per rappresentare il carattere.[1] Le funzioni del tipo ***is...()*** restituiscono un valore intero diverso da zero (corrispondente a *Vero*) se la condizione riferita al carattere fornito si verifica. Le funzioni ***to...()*** restituiscono un valore intero, corrispondente al carattere fornito e trasformato nel modo richiesto, se ciò è possibile.

Listato 69.47. Prototipi delle funzioni dichiarate nel file '`ctype.h`'.

```
int isalnum  (int c);
int isalpha  (int c);
int isblank  (int c);
int iscntrl  (int c);
int isdigit  (int c);
int isgraph  (int c);
int islower  (int c);
int isprint  (int c);
int ispunct  (int c);
int isspace  (int c);
int isupper  (int c);
int isxdigit (int c);
int tolower  (int c);
int toupper  (int c);
```

Listato 69.48. Prototipi aggiuntivi dello standard POSIX.

```
int isascii  (int c);           // POSIX
int toascii  (int c);           // POSIX
```

69.7.1 Funzioni «is...()»

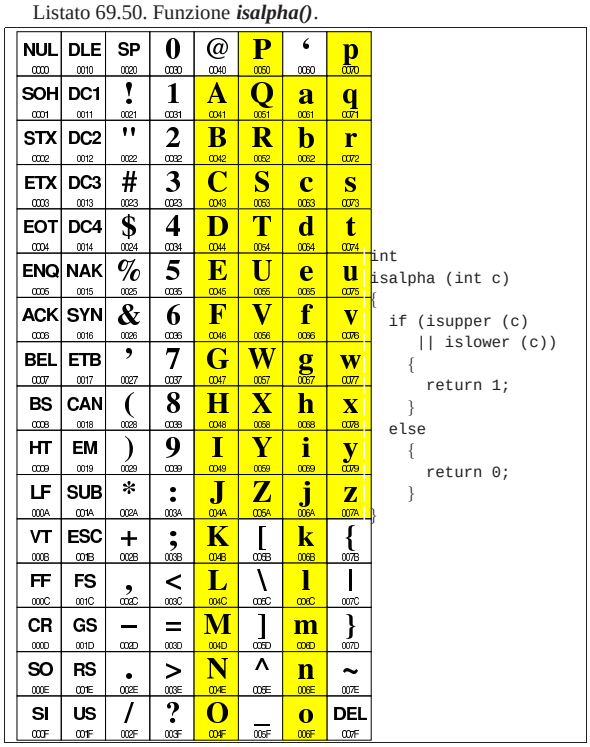
Il gruppo di funzioni ***is...()*** restituisce un valore intero diverso da zero (corrispondente a *Vero*) se la condizione riferita al carattere fornito si verifica. Vengono proposte le varie soluzioni, affiancando la tabella ASCII con i caratteri validi evidenziati.

Listato 69.49. Funzione ***isalnum()***.

NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	‘ 0060	p 0070
SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
ETX 0003	DC3 0013	# 0023	3 0033	C 0043	S 0053	c 0063	s 0073
EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
BEL 0007	ETB 0017	’ 0027	7 0037	G 0047	W 0057	g 0067	w 0077
BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
CR 000D	GS 001D	- 002D	= 003D	M 004D	] 005D	m 006D	} 007D
SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
SI 000F	US 001F	/ 003F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

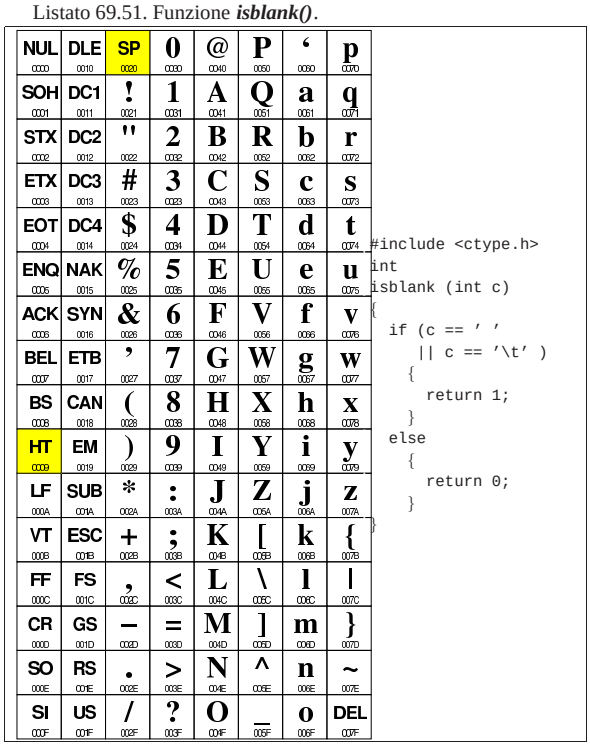
```
#include <ctype.h>
int
isalnum (int c)
{
  if (isalpha (c)
      || isdigit (c))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.50. Funzione ***isalpha()***.



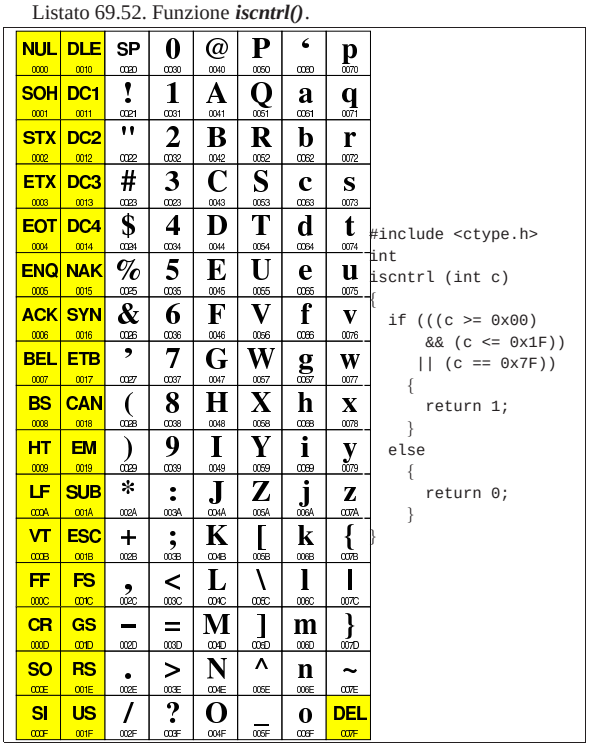
```
int
isalpha (int c)
{
  if (isupper (c)
      || islower (c))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.51. Funzione ***isblank()***.



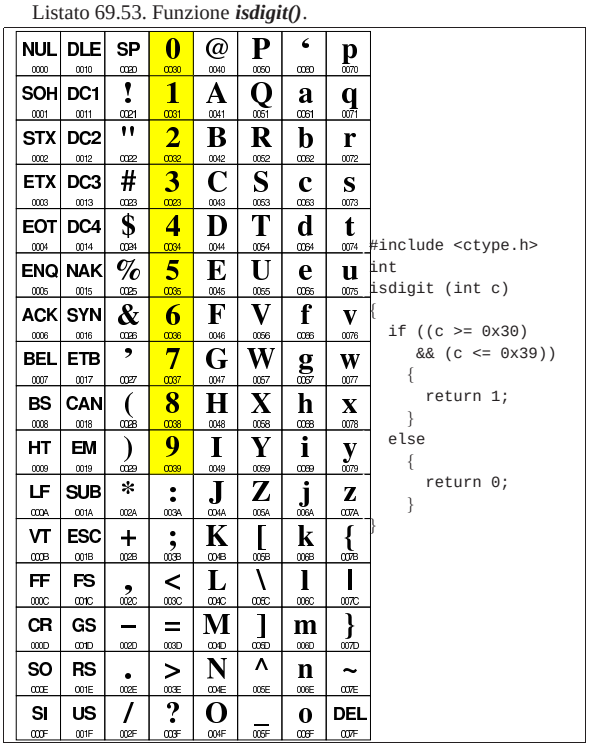
```
#include <ctype.h>
int
isblank (int c)
{
  if (c == ' '
      || c == '\t' )
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.52. Funzione ***iscntrl()***.



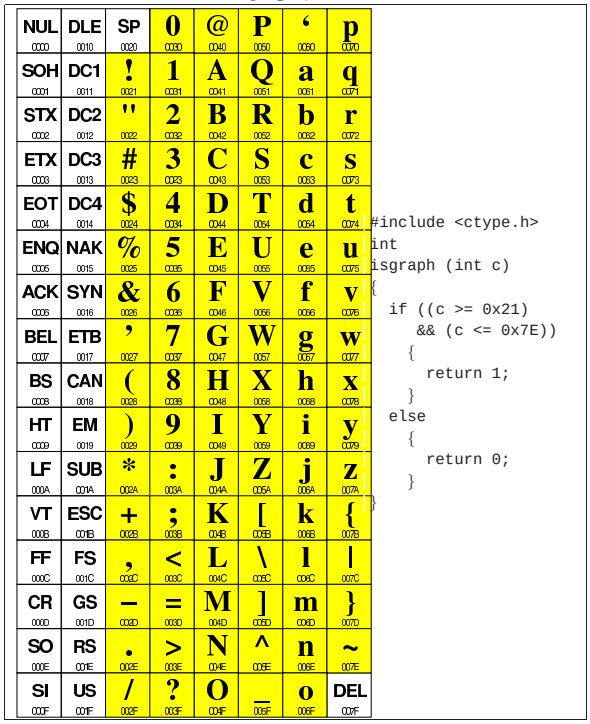
```
#include <ctype.h>
int
iscntrl (int c)
{
  if (((c >= 0x00)
       && (c <= 0x1F))
      || (c == 0x7F))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.53. Funzione ***isdigit()***.



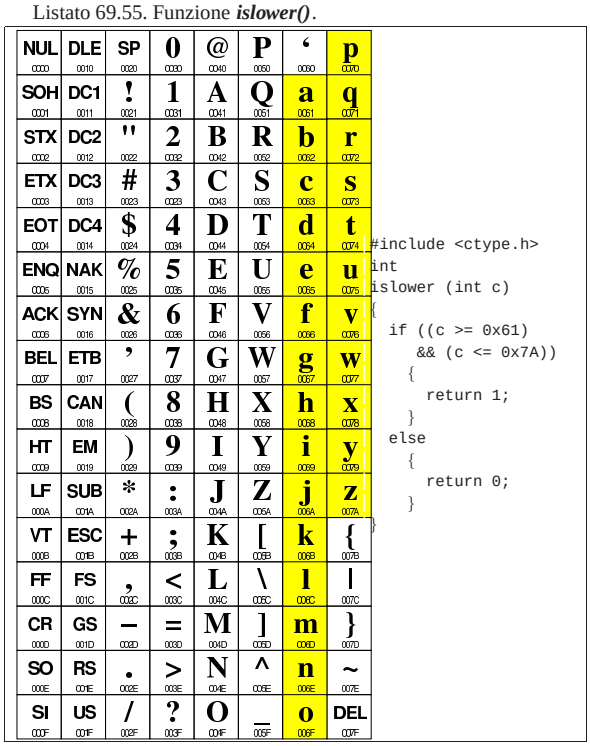
```
#include <ctype.h>
int
isdigit (int c)
{
  if ((c >= 0x30)
      && (c <= 0x39))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.54. Funzione ***isgraph()***.



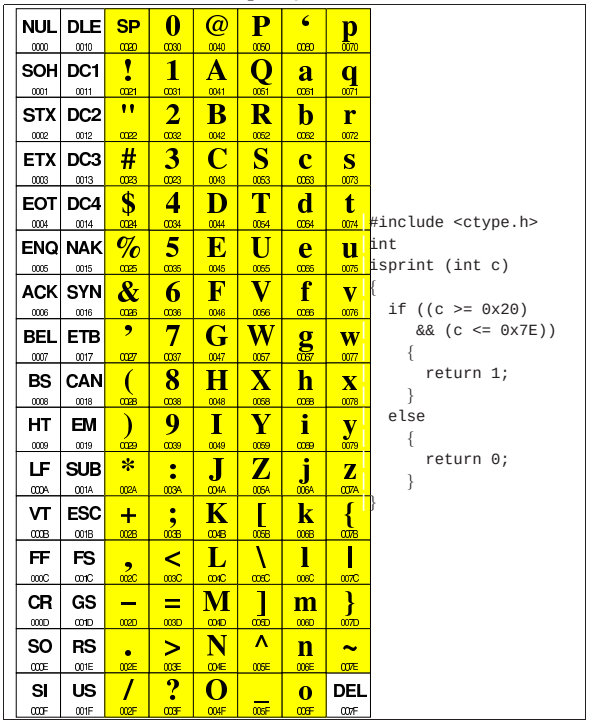
```
#include <ctype.h>
int
isgraph (int c)
{
  if ((c >= 0x21)
      && (c <= 0x7E))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.55. Funzione ***islower()***.



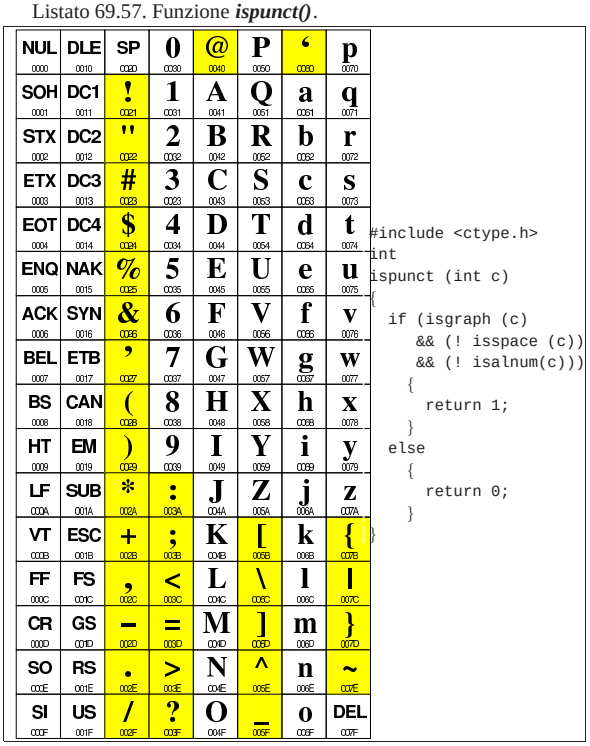
```
#include <ctype.h>
int
islower (int c)
{
  if ((c >= 0x61)
      && (c <= 0x7A))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.56. Funzione ***isprint()***.



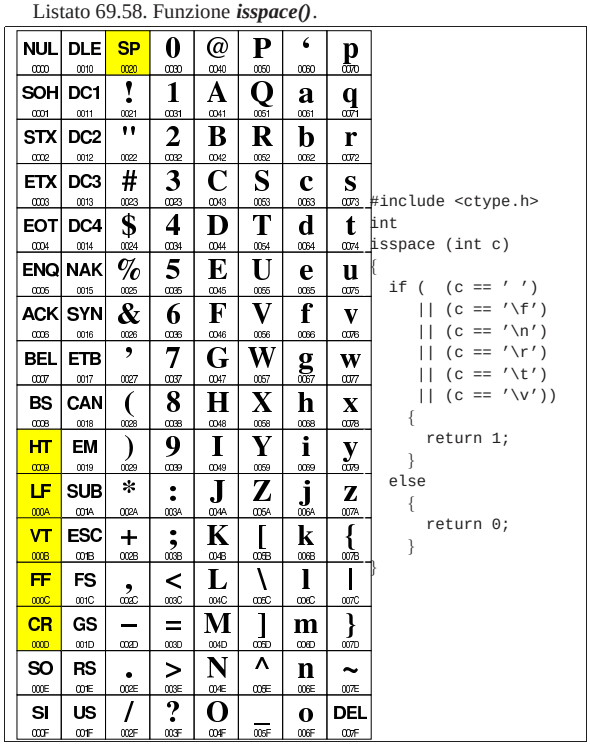
```
#include <ctype.h>
int
isprint (int c)
{
  if ((c >= 0x20)
      && (c <= 0x7E))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.57. Funzione ***ispunct()***.



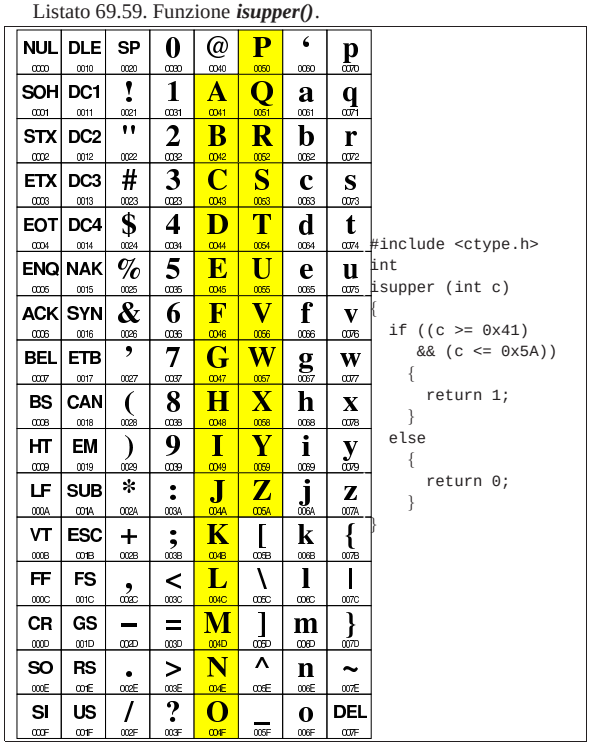
```
#include <ctype.h>
int
ispunct (int c)
{
  if (isgraph (c)
      && (! isspace (c))
      && (! isalnum(c)))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.58. Funzione ***isspace()***.



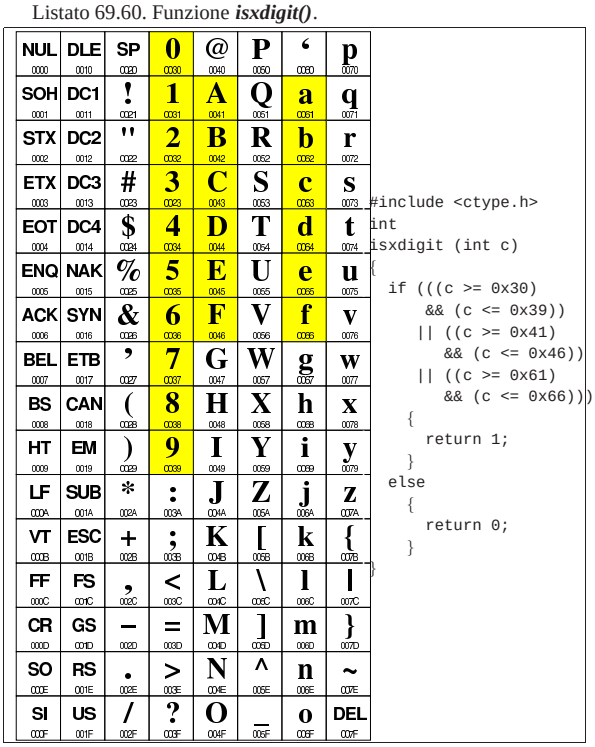
```
#include <ctype.h>
int
isspace (int c)
{
  if (  (c == ' ')
      || (c == '\f')
      || (c == '\n')
      || (c == '\r')
      || (c == '\t')
      || (c == '\v'))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.59. Funzione ***isupper()***.



```
#include <ctype.h>
int
isupper (int c)
{
  if ((c >= 0x41)
     && (c <= 0x5A))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.60. Funzione ***isxdigit()***.



```
#include <ctype.h>
int
isxdigit (int c)
{
  if (((c >= 0x30)
      && (c <= 0x39))
     || ((c >= 0x41)
        && (c <= 0x46))
     || ((c >= 0x61)
        && (c <= 0x66)))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.61. Nello standard POSIX si aggiunge la funzione ***isascii()***.

```
#include <ctype.h>
int
isascii (int c)
{
  if ((c >= 0x00)
     && (c <= 0x7F))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

69.7.2 macroistruzioni «is...()»

In alternativa a delle funzioni vere e proprie, si possono realizzare semplicemente delle macroistruzioni per verificare le condizioni riferite al carattere. Il listato seguente è conforme a quanto già visto nella sezione precedente:

```
#define isblank(C)  ((int) (C == ' ' || C == '\t'))
#define isspace(C)  ((int) (C == ' ' \
                            || C == '\f' \
```

```
                          || C == '\n' \
                          || C == '\n' \
                          || C == '\r' \
                          || C == '\t' \
                          || C == '\v'))
#define isdigit(C)  ((int) (C >= '0' && C <= '9'))
#define isxdigit(C) ((int) ((C >= '0' && C <= '9') \
                            || (C >= 'A' && C <= 'F') \
                            || (C >= 'a' && C <= 'f')))
#define isupper(C)  ((int) (C >= 'A' && C <= 'Z'))
#define islower(C)  ((int) (C >= 'a' && C <= 'z'))
#define iscntrl(C)  ((int) ((C >= 0x00 && C <= 0x1F)
                            || C == 0x7F))
#define isgraph(C)  ((int) (C >= 0x21 && C <= 0x7E))
#define isprint(C)  ((int) (C >= 0x20 && C <= 0x7E))
#define isalpha(C)  (isupper (C) || islower (C))
#define isalnum(C)  (isalpha (C) || isdigit (C))
#define ispunct(C)  (isgraph (C) && (!isspace (C))
                                 && (!isalnum (C)))
#define isascii(C)  ((int) (C >= 0x00 && C <= 0x7F))
```

69.7.3 Funzioni di conversione

Le due funzioni ***tolower()*** e ***toupper()*** si occupano di convertire un carattere, rispettivamente, in minuscolo o in maiuscolo.

Listato 69.63. Funzione ***tolower()***.

NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	‘ 0060	p 0070
SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
ETX 0003	DC3 0013	# 0023	3 0033	C 0043	S 0053	c 0063	s 0073
EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
BEL 0007	ETB 0017	’ 0027	7 0037	G 0047	W 0057	g 0067	w 0077
BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
CR 000D	GS 001D	– 002D	= 003D	M 004D	] 005D	m 006D	} 007D
SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

```
#include <ctype.h>
int
tolower (int c)
{
  if (isupper (c))
    {
      return
        (c + 0x20);
    }
  else
    {
      return c;
    }
}
```

Listato 69.64. Funzione ***toupper()***.

NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	‘ 0060	p 0070
SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
ETX 0003	DC3 0013	# 0023	3 0033	C 0043	S 0053	c 0063	s 0073
EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
BEL 0007	ETB 0017	’ 0027	7 0037	G 0047	W 0057	g 0067	w 0077
BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
CR 000D	GS 001D	– 002D	= 003D	M 004D	] 005D	m 006D	} 007D
SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

```
#include <ctype.h>
int
toupper (int c)
{
  if (islower (c))
    {
      return
        (c - 0x20);
    }
  else
    {
      return c;
    }
}
```

Lo standard POSIX aggiunge anche la funzione ***toascii()*** che si limita ad azzerare i bit più significativi, dopo il settimo.

Listato 69.65. Funzione ***toascii()***.

```
#include <ctype.h>
int
toascii (int c)
{
  return (c & 0x7F);
}
```

69.7.4 macroistruzioni di conversione

Anche le funzioni ***toupper()***, ***tolower()*** e ***toascii()*** possono essere rappresentate agevolmente in forma di macroistruzioni. Il listato seguente è conforme a quanto già visto nella sezione precedente:

```
#define tolower(C)  (isupper (C) ? ((C) + 0x20) : (C))
#define toupper(C)  (islower (C) ? ((C) - 0x20) : (C))
#define toascii(C)  (C & 0x7F)
```

Ma nel caso dello standard POSIX, in questo caso vanno ancora aggiunge due macroistruzioni, a cui non fanno capo funzioni con lo stesso nome:

```
#define _tolower(C) (isupper (C) ? ((C) + 0x20) : (C))
#define _toupper(C) (islower (C) ? ((C) - 0x20) : (C))
```

69.7.5 Esempio di utilizzo delle funzioni

Viene proposto un programma molto semplice che utilizza tutte le funzioni dichiarate nel file ‘`ctype.h`’, ma solo secondo lo standard C:

```
#include <stdio.h>
#include <ctype.h>

int
main (int argc, char *argv[])
{
  int c;
  for (c = 0; c <= 0x7F; c++)
    {
      printf ("%02x", c);
      printf ("\t"); if (iscntrl (c))  printf ("cntrl");
                     if (isprint (c))  printf ("print");
      printf ("\t"); if (isblank (c))  printf ("blank");
```

```
                    if (isgraph (c))  printf ("graph");
      printf ("\t"); if (isspace (c))  printf ("space");
                    if (ispunct (c))  printf ("punct");
                    if (isupper (c))  printf ("upper");
                    if (islower (c))  printf ("lower");
                    if (isdigit (c))  printf ("digit");
      printf ("\t"); if (isalnum (c))  printf ("alnum");
      printf ("\t"); if (isxdigit (c)) printf ("xdigit");
      printf ("\t"); if (isalpha (c))  printf ("alpha");
      printf ("\n");
    }
  printf ("\n");
  printf ("ASCII:\n");
  for (c = 0; c <= 0x7F; c++)
    {
      if (isprint (c)) printf ("%c", c);
    }
  printf ("\n");
  printf ("\n");
  printf ("to upper:\n");
  for (c = 0; c <= 0x7F; c++)
    {
      if (isprint (c)) printf ("%c", toupper (c));
    }
  printf ("\n");
  printf ("\n");
  printf ("to lower:\n");
  for (c = 0; c <= 0x7F; c++)
    {
      if (isprint (c)) printf ("%c", tolower (c));
    }
  printf ("\n");
  return 0;
}
```

Una volta compilato il programma, avviandolo si deve ottenere un testo come quello che si vede nell'estratto seguente:

```
…
44      print   graph   upper   alnum   xdigit  alpha
45      print   graph   upper   alnum   xdigit  alpha
46      print   graph   upper   alnum   xdigit  alpha
47      print   graph   upper   alnum           alpha
48      print   graph   upper   alnum           alpha
…
ASCII:
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJ↩
↪KLMNOPQRSTUVWXYZ[\]^_‘abcdefghijklmnopqrstuvwxyz{|}~

to upper:
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJ↩
↪KLMNOPQRSTUVWXYZ[\]^_‘ABCDEFGHIJKLMNOPQRSTUVWXYZ{|}~

to lower:
!"#$%&'()*+,-./0123456789:;<=>?@abcdefghij↩
↪klmnopqrstuvwxyz[\]^_‘abcdefghijklmnopqrstuvwxyz{|}~
```

69.8 File «stdarg.h»

Il file '`stdarg.h`' della libreria standard definisce principalmente delle macroistruzioni per gestire gli argomenti variabili passati a una funzione, assieme a un tipo di variabile, '**`va_list`**', specifico per gestire il puntatore a tali parametri non dichiarati (si veda eventualmente la realizzazione del file '`stdarg.h`' nei sorgenti di os32, listato 95.1.10).

Tabella 69.70. macroistruzioni standard per la gestione di argomenti variabili.

Macroistruzione	Descrizione
`void va_start (va_list` ***ap***`,` `  `***parametro_n***`);`	Inizializza la variabile ***ap***, di tipo '**`va_list`**', in modo che punti all'area di memoria immediatamente successiva al parametro indicato, il quale deve essere l'ultimo.
tipo `va_arg (va_list` ***ap***`,` ***tipo***`);`	Restituisce il contenuto dell'area di memoria a cui punta ***ap***, utilizzando il tipo indicato, incrementando contestualmente il puntatore in modo che, al termine, si trovi nell'area di memoria immediatamente successiva.
`void va_copy (va_list` ***dst***`,` `  va_list` ***org***`);`	Copia il puntatore ***org*** nella variabile ***dst***.
`void va_end (va_list` ***ap***`);`	Conclude l'utilizzo del puntatore ***ap***.

69.8.1 Realizzazione

Il listato successivo è tutto ciò che serve per realizzare la libreria:

```
typedef unsigned char * va_list;

#define va_start(AP, LAST) \
    ((void) ((AP) = ((va_list) &(LAST)) + (sizeof (LAST))))

#define va_end(AP) ((void) ((AP) = 0))
#define va_copy(DEST, SRC) \
    ((void) ((DEST) = (va_list) (SRC)))

#define va_arg(AP, TYPE) \
    (((AP) = (AP) + (sizeof (TYPE))), \
        *((TYPE *) ((AP) - (sizeof (TYPE)))))
```

Delle macroistruzioni mostrate nell'esempio, la più difficile da interpretare potrebbe essere ***va_arg***, la quale deve restituire il valore dell'area di memoria puntata inizialmente, ma garantendo di lasciare il puntatore pronto per l'area successiva. In pratica, prima viene incrementato il puntatore per l'area successiva, quindi viene dereferenziato ricalcolando lo spazio necessario a raggiungere la posizione precedente. In altri termini è come scrivere:

```
…
va_list ap;
…
//
// va_start
//
ap = (va_list) &ultimo_parametro;
ap = ap + (sizeof (tipo_ultimo_parametro);
…
//
// va_arg
//
ap = ap + (sizeof tipo_successivo);
var = (tipo_successivo *)
      (ap - (sizeof (tipo_successivo)));
…
//
// va_end
//
ap = 0;
…
```

69.8.2 Esempio di utilizzo delle macro

Viene riproposto un programma molto semplice, già apparso in altri capitoli, per dimostrare l'utilizzo delle macroistruzioni dichiarate nel file '`stdarg.h`'.

```
#include <stdio.h>
#include <stdarg.h>

void
f (int w,...)
{
  long double x;     // Dichiara le variabili che servono
  long long int y;   // a contenere gli argomenti per i
  int z;             // quali mancano i parametri formali.
```

```
  va_list ap;       // Dichiara il puntatore agli argomenti.

  va_start (ap, w); // Posiziona il puntatore dopo la fine
                    // di «w».

  x = va_arg (ap, long double); // Estrae l'argomento
                                // successivo portando
                                // avanti il puntatore
                                // di conseguenza.

  printf ("w = %i; ", w);   // Mostra il valore del primo
                            // parametro.
  printf ("x = %Lf; ", x);  // Mostra il valore
                            // dell'argomento successivo.

  y = va_arg (ap, long long int);  // Estrapola e mostra
  printf ("y = %lli; ", y);        // il terzo argomento.

  z = va_arg (ap, int);            // Estrapola e mostra
  printf ("z = %i\n", z);          // il quarto argomento.

  va_end (ap);                     // Conclude la scansione.

  return;
}

int
main (int argc, char *argv[])
{
  f (10, (long double)12.34, (long long int)13, 14);
  return 0;
}
```

Avviando il programma di esempio si deve visualizzare il messaggio seguente:

```
w = 10; x = 12.340000; y = 13; z = 14
```

69.8.3 Promozione

Va ricordato che gli argomenti delle chiamate alle funzioni vengono adattati in modo tale da facilitare l'uso della pila dei dati. Pertanto, i valori che prevedono una rappresentazione in memoria troppo piccola, subiscono quella che è nota come «promozione».

La funzione che ha degli argomenti variabili, dovrebbe gestire solo valori che non possono subire una trasformazione di questo tipo, altrimenti, quando poi utilizza la macroistruzione ***va_arg*** deve indicare un tipo adeguato alla promozione che si prevede sia applicata ai valori degli argomenti.

A questo proposito si può notare che nell'esempio di utilizzo che appare nella sezione 69.8.2, non si fa mai uso di tipi di dati di rango inferiore a '**int**'.

69.9 File «stdlib.h»

Il file '`stdlib.h`' della libreria standard definisce alcuni tipi di dati, varie funzioni di utilità generale e alcune macro-variabili. Viene proposto un esempio di questo file, e di alcune delle funzioni a cui si riferisce, indicando per le altre solo i prototipi (si veda eventualmente la realizzazione del file '`stdlib.h`' e di alcune delle sue funzioni, nei sorgenti di os32, sezione 95.19).

Lo standard POSIX estende significativamente il contenuto del file '`stdlib.h`', ma qui non si fa riferimento ad alcuna di tali estensioni.

69.9.1 Tipi di dati speciali

I tipi di dati che il file '`stdlib.h`' definisce sono '**size_t**' e '**wchar_t**', già descritti nel file '`stddef.h`' (sezione 69.13), oltre a '**div_t**', '**ldiv_t**' e '**lldiv_t**'. I tipi '...**div_t**' sono delle strutture il cui scopo è quello di contenere il risultato di una divisione, espresso come quoziente e resto. Questi tipi di dati si usano per contenere il valore restituito dalle funzioni ***div()***, ***ldiv()*** e ***lldiv()***. La distinzione tra i tre tipi deriva dalla capienza dei membri della struttura. Ecco come potrebbero essere dichiarati:


```
typedef struct {
    int quot;
    int rem;
} div_t;
typedef struct {
    long int quot;
    long int rem;
} ldiv_t;
typedef struct {
    long long int quot;
    long long int rem;
} lldiv_t;
```

69.9.2 Macro-variabili

Il file '`stdlib.h`' dichiara nuovamente la macro-variabile ***NULL***, come già avviene nel file '`stddef.h`' (sezione 69.13); inoltre definisce quelle seguenti:

Macro	Descrizione

`EXIT_SUCCESS`
 	
`EXIT_FAILURE`
	Rappresentano un numero intero che possa essere usato come argomento della funzione ***exit()***, per rappresentare il successo o l'insuccesso dell'attività svolta. Normalmente, i valori in cui si espandono le due macro-variabili sono rispettivamente zero e uno.

`RAND_MAX`	Rappresenta il valore massimo che possa essere generato dalla funzione ***rand()*** e generalmente corrisponde al valore massimo che può assumere un numero intero di tipo '**int**'.
`MB_CUR_MAX`	Rappresenta la quantità massima di byte che possono essere utilizzati in una sequenza multibyte, in base alla configurazione locale. Questo valore è un intero che deve essere di tipo '**size_t**' è non può superare il limite rappresentato dalla macro-variabile ***MB_LEN_MAX*** (dichiarata nel file '`limits.h`', descritto nella sezione 69.3).

Merita un po' di attenzione la macro-variabile ***MB_CUR_MAX***. La sequenza multibyte è una sequenza di byte che, in base alla configurazione locale, deve essere interpretata come un carattere singolo. Per esempio, questo meccanismo si utilizza nella codifica UTF-8 e in altre; ma proprio perché esistono più metodi alternativi, per quanto superati possano essere rispetto a UTF-8, la configurazione locale stabilisce le regole particolari per interpretare tali sequenze e i limiti rispetto a queste. Pertanto, la macro-variabile ***MB_CUR_MAX*** dovrebbe espandersi in una funzione che restituisce il valore desiderato, in relazione alla configurazione locale che si trova a essere attiva in un certo momento. Per semplicità, nell'esempio che viene proposto si associa il valore di questa macro-variabile a quello massimo accettabile in assoluto.

```
#define EXIT_FAILURE    1
#define EXIT_SUCCESS    0

#define RAND_MAX        INT_MAX

#define MB_CUR_MAX      ((size_t) MB_LEN_MAX)  // [1]
//
// [1] Sarebbe meglio una funzione.
//
```

Nell'esempio proposto viene usata la macro-variabile ***MB_LEN_MAX***, pertanto, in questo modo si rende necessaria l'inclusione del file '`limits.h`' che deve contenere la sua dichiarazione.

69.9.3 Conversioni numeriche

Un gruppo di funzioni del file '`stdlib.h`' permette di convertire una stringa in un valore numerico. In particolare, le funzioni con nomi del tipo ***ato...()*** (*ASCII to ...*) non eseguono controlli particolari e

non modificano la variabile ***errno*** (sezione 69.5); invece, le funzioni con nomi ***strto...()*** (*string to ...*) sono più sofisticate.

Le funzioni ***ato...()*** interpretano una stringa e convertono il suo contenuto in un numero intero o in un numero a virgola mobile. Le funzioni sono ***atoi()***, ***atol()***, ***atoll()*** e ***atof()***, che convertono rispettivamente in un tipo '**int**', '**long int**', '**long long int**' e '**double**'. Ecco i prototipi:

```
int           atoi  (const char *nptr);
long int      atol  (const char *nptr);
long long int atoll (const char *nptr);
double        atof  (const char *nptr);
```

Viene proposta una soluzione per queste funzioni di conversione:

```
#include <stdlib.h>
#include <ctype.h>
int
atoi (const char *nptr)
{
  int i;
  int sign = +1;
  int n;

  for (i = 0; isspace (nptr[i]); i++)
    {
      ;       // Si limita a saltare gli spazi iniziali.
    }

  if (nptr[i] == '+')
    {
      sign = +1;
      i++;
    }
  else if (nptr[i] == '-')
    {
      sign = -1;
      i++;
    }

  for (n = 0; isdigit (nptr[i]); i++)
    {
      n = (n * 10) + (nptr[i] - '0'); // Accumula il valore.
    }

  return sign * n;
}
```

Logicamente, le funzioni ***atol()*** e ***atoll()*** sono praticamente uguali, con la differenza che la variabile automatica ***n*** deve essere dello stesso tipo restituito dalla funzione; pertanto si passa alla soluzione proposta per la funzione ***atof()***:

```
#include <stdlib.h>
#include <ctype.h>
double
atof (const char *nptr)
{
  int i;
  int sign = +1;
  double n;           // Il risultato sarà:  n / d.
  double d;           //

  for (i = 0; isspace (nptr[i]); i++)
    {
      ;       // Si limita a saltare gli spazi iniziali.
    }

  if (nptr[i] == '+')
    {
      sign = +1;
      i++;
    }
  else if (nptr[i] == '-')
    {
      sign = -1;
      i++;
    }

  for (n = 0.0; isdigit (nptr[i]); i++)
    {
      n = (n * 10.0) + (nptr[i] - '0');     // Accumula
                                            // il valore.
    }

  if (nptr[i] == '.')
    {
      i++;
    }

  for (d = 1.0; isdigit (nptr[i]); i++)
    {
      // La variabile "d" viene inizializzata in ogni caso.

      n = (n * 10.0) + (nptr[i] - '0');
      d = d * 10.0;   // Tiene conto di quanto dovrà essere
                      // diviso il risultato.
    }

  return sign * n / d;
}
```


Le funzioni ***strto...()*** sono più complesse rispetto a quelle ***ato...()***. Per dare una descrizione sommaria, si può osservare che, oltre alla stringa da scandire ricevono un puntatore di puntatore all'ultimo elemento utile di tale stringa;[2] se poi questo è nullo, la scansione avviene normalmente nella stringa, entro il limite del carattere nullo di terminazione. Se fallisce il riconoscimento del valore da tradurre, il puntatore all'inizio della stringa viene copiato nell'area di memoria a cui punta il puntatore di puntatore, a meno che questo, inizialmente, sia già nullo (potrebbe essere nullo il puntatore principale o il contenuto dell'area a cui punta).

In caso di errore nell'interpretazione del valore, queste funzioni utilizzano la variabile ***errno*** per annotare il tipo di problema riscontrato.

I valori che possono essere convertiti sono esprimibili in notazione decimale o esadecimale; inoltre, le funzioni che convertono in valori a virgola mobile, accettano una notazione esponenziale e delle parole chiave per rappresentare l'infinito e NaN (*Not a number*). Nel caso particolare delle funzioni che convertono in un numero intero, esiste un terzo parametro per specificare la base di numerazione attesa.

Vengono presentati solo i prototipi di queste funzioni:

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

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

Per una descrizione completa si vedano le pagine di manuale *strtof(3)*, *strtod(3)*, *strtold(3)*, *strtol(3)*, *strtoll(3)*, *strtoul(3)* e *strtoull(3)*, oltre alla documentazione standard citata alla fine del capitolo.

L'esempio seguente mostra l'uso delle funzioni ***ato...()***:

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

int main (void)
{
  int             i;
  long int       li;
  long long int lli;
  double          d;
  char          s[] = "  -987654.3210";
```

```
  i = atoi (s);
  li = atol (s);
  lli = atoll (s);
  d = atof (s);

  printf ("\"%s\" = %i, %li, %lli, %f\n", s, i, li, lli, d);

  return 0;
}
```

Il risultato che ci si attende di visualizzare è questo:

```
"  -987654.3210" = -987654, -987654, -987654, -987654.321000
```

69.9.4 Funzioni per la generazione di numeri in modo pseudo-casuale

La libreria standard deve disporre, nel file '`stdlib.h`', di due funzioni per la generazione di numeri pseudo-casuali. Si tratta precisamente di ***rand()*** che restituisce un numero intero casuale (di tipo '**`int`**', ma sono ammessi solo valori positivi) e di ***srand()*** che serve a cambiare il «seme» di generazione di tali numeri. Lo standard prescrive anche che per uno stesso seme, la sequenza di numeri pseudo-casuali sia la stessa e che il seme predefinito iniziale sia pari a uno.

Nella descrizione dello standard si fa riferimento al fatto che il valore che può essere generato deve andare da zero a '**`RAND_MAX`**', escludendo quindi valori negativi. Considerando che la funzione ***rand()*** restituisce un valore di tipo '**`int`**' e che questo non può essere negativo, significa che '**`RAND_MAX`**' deve essere inferiore o uguale al massimo numero positivo rappresentabile con il tipo '**`int`**'.

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

Viene proposta una soluzione molto semplice e anche molto scadente sul piano della sequenza casuale generata. Tuttavia garantisce che il valore ottenuto vada effettivamente da zero a '**`RAND_MAX`**' incluso:

```
#include <stdlib.h>
static unsigned int _srand = 1; // Il rango di «_srand» deve
                                // essere maggiore o uguale
                                // a quello di «RAND_MAX»
                                // e di «unsigned int».
int
rand (void)
{
  _srand = _srand * 1234567 + 12345;
  return _srand % ((unsigned int) RAND_MAX + 1);
}

void
srand (unsigned int seed)
{
  _srand = seed;
}
```

L'esempio seguente consente di verificare sommariamente il lavoro delle funzioni per la generazione di numeri casuali. Per la precisione, si vogliono ottenere valori che vanno da 0 a 99 inclusi:

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

int main (void)
{
  int r;
  int i;
  int j;
  const int max = 99;

  srand (123);

  for (i = 0; i < 25; i++)
  {
      for (j = 0; j < 26; j++)
        {
          r = (rand () % (max + 1));
          printf ("%2d ", r);
        }
      printf ("\n");
  }

  return 0;
}
```

Si dovrebbe ottenere un risultato simile a quello seguente:

```
86 67 22 15 22 47 94 91 10 99  6 31 22 51 98 11 14 11 22  3
22 11 82  7 54 11 62 75 10 63 50 75 98 91 90 79 74 87 18 15
34  7 54 71 42 47 10 23 82 67 50 19 14 31 34 27 58 71 38 43
14 87 38 71 82 95 50 99  2  7 30  7 98 87  2 19 46 71 78 83
34 43 58 99 70 79 30 75 58 99 46 31 78 91 34 79 98 75 14 99
66 63 82 99  2 51 46 11 74 39 90 31 54 63 78 39 14 31 50 55
26 87  2 51 70 15 98 67 54 27 34 55 14  7 74 71 42  7 30 87
70 67 34 15 26 47 90 91 30 19 66 27 86 15 26 15  6 99 46  7
14 51 22 43 54 95 58  7 18 15 50 83 26 71  2 27 38 71 30  7
94 19  6 47 62 19 90 39 54 39 42 15 86 91 82 87 38 27 34 87
74 87 50 87 82 67 78 63 22  3 26 87 86 11 94 71 26 35 10 83
78 71 46 63 82 47 74 63 70 11 54 95 62 31 74 87 30 71 54 15
18 19 14 79 86 23 58  7 98 47 10 63 46 39 26 19 10 67 54 99
 6 47 70 11 26  7 66 23 10 51 66 31 58 91 74 87 22 35 74 11
 6 39 58 39 26 35 94 55 82 47 78 67 98 91 30 67 18 75 74 83
18 23 54 51 26 91 38 11 22 23 66 91 18 99 74 27 82 99 62 35
58 35 54 95  2 95 86 95 82 47 90 51 90 51 54 23 62 91 90 99
18 19 62 51 10 59 26 39 82 71 94 83 98 27 38  7 22 15 90 79
18 31 58 71 22  3 58 39 98 63 62 55 38 51 86 51  2  7 86 87
94 43 54 67 62 23 42 31 10  3 42 47 74 87 26 27 54 43 54 95
18 31 34 27 58 47 66 47 70 75 22 35 82  7 54  7 62 19 18 91
22 63 94 83 98 27 86 59 78 91 78 35 62 67 90 67 62 35 78 95
86 99 54 47 14  3 62 19 58 63 74 11 62 99 46 71 86  7 34  3
34 59 18  3 62 43 62 27 26 95 46  3 22 15 54 35  6 75 86 27
58 51 70 71 82 15 50 39 38 91 14 99 66 67 66 91 54 31 34  3
30 55 98 27 50  7 38 11 26 11 42 47 26 31 10 91 90 15 54 19
50 87 70 35 14 79 10 47 74 55 34 51 54 95 58 23 46 59 78 91
38  3 94 51 70  3 54 51 86 51 14 95 70 55 50 99 70 63 86 15
 6 35 98 67 74 91 54 99 30  7 94 35 10 43 54 55 50 11 58 75
26 27 78 99 98 55 94 79 26 83 34 87 38 47  6 55 74 23 10 11
26 87  6 59 10 71 86 31 78 55 30 39 66 47 46 11 30 47  2 11
74 55 42 59  2 59 42 63 78 71  2 19  6 83 30 23  2 31 54 47
62 31 26 67  6 67 70 63 14 91
```

69.9.5 Funzioni standard per la generazione di numeri pseudo-casuali

Il documento che descrive lo standard descrive una versione della funzione ***rand()*** che corrisponde al listato successivo. I valori usati nei calcoli sono tali da essere adatti a un contesto in cui i limiti degli interi sono quelli minimi previsti.

Listato 69.89. Esempio di realizzazione delle funzioni ***rand()*** e ***srand()***, tratto dal documento che descrive lo standard del linguaggio.

```
// RAND_MAX assumed to be 32767

static unsigned long int next = 1;

int
rand(void)
{
  next = next * 1103515245 + 12345;
  return (unsigned int)(next/65536) % 32768;
}

void
srand(unsigned int seed)
{
  next = seed;
}
```

69.9.6 Amministrazione della memoria

Un gruppo di funzioni dichiarate nel file '`strlib.h`' consente di utilizzare dinamicamente la memoria. Si tratta di ***malloc()***, ***calloc()***, ***realloc()*** e ***free()***. Le prime tre funzioni restituiscono un puntatore di tipo '**`void *`**' all'area di memoria allocata, oppure il puntatore nullo nel caso l'operazione di allocazione fallisca; la funzione ***free()*** libera

un'area di memoria allocata, indicando come argomento il puntatore che inizialmente la rappresentava.

```
void *malloc  (size_t size);
void *calloc  (size_t nmemb, size_t size);
void *realloc (void *ptr, size_t size);
void free     (void *ptr);
```

Rispetto ai prototipi mostrati, la funzione ***malloc()*** richiede l'allocazione di una quantità di byte espressa dal parametro ***size***; ***calloc()*** richiede una quantità di ***nmemb*** elementi da ***size*** byte (pertanto serve solo a facilitare l'allocazione di uno spazio necessario a un array); ***realloc()*** richiede la riallocazione della memoria già allocata precedentemente a partire dall'indirizzo ***ptr*** per avere ***size*** byte, con l'intento di non perdere le informazioni precedenti (a meno che si tratti di una riduzione della dimensione); infine, ***free()*** si limita a deallocare la memoria a cui punta ***ptr***.

Nel linguaggio C, la memoria deve essere allocata e liberata espressamente, in quanto non esiste alcun sistema automatico al riguardo.

La gestione della memoria dipende strettamente dal sistema operativo, pertanto la realizzazione delle funzioni non può essere generalizzata. Per i dettagli che riguardano il comportamento di queste funzioni nel proprio sistema operativo vanno consultate le pagine di manuale *malloc(3)*, *calloc(3)*, *realloc(3)* e *free(3)*.

69.9.7 Conclusione forzata del programma

Alcune funzioni si occupano di interrompere il funzionamento del programma al di fuori della conclusione naturale della funzione ***main()***. In generale si possono distinguere i casi in cui la conclusione del programma viene gestita in modo gentile, oppure viene forzata brutalmente.

Per una conclusione corretta di un programma, è possibile predisporre un elenco di funzioni da eseguire automaticamente nel momento della conclusione. Ciò avviene attraverso la funzione ***atexit()*** che accumula un elenco di puntatori a funzione; successivamente, attraverso la chiamata alla funzione ***exit()*** si ottiene l'esecuzione delle funzioni dell'elenco, senza argomenti, secondo l'ordine di inserimento. Quindi, la funzione ***exit()*** conclude con la chiusura dei file e con la restituzione del valore passatole come argomento.

Una conclusione brutale si ottiene con la funzione ***_Exit()***, che si limita a terminare il programma, ma senza fare altro, soprattutto senza garantire che i file aperti siano chiusi correttamente.

Per ottenere una conclusione brutale del funzionamento di un programma si può usare anche la funzione ***abort()*** che però è legata alla gestione dei segnali (sezione 69.15) e qui non viene spiegato il suo utilizzo.

```
int  atexit (void (*func) (void));
void exit   (int status);
void _Exit  (int status);
void abort  (void);
```

La funzione ***atexit()*** riceve come unico argomento il puntatore a una funzione, la quale non restituisce alcun valore (di tipo '**void**') e non si attende alcun argomento (ancora il tipo '**void**'). La funzione ***atexit()*** restituisce un valore numerico da intendere come *Vero* o *Falso*, per comunicare il successo o l'insuccesso dell'operazione, dato che la quantità di puntatori a funzione che possono essere accumulati può avere un limite.

Per una descrizione completa dell'uso di queste funzioni si vedano le pagine di manuale *abort(3)*, *atexit(3)*, *exit(3)* e *_Exit(3)*.

69.9.8 Funzioni di comunicazione con l'ambiente

Nel file '`stdlib.h`' sono dichiarate due funzioni per interagire con il sistema operativo, ***getenv()*** e ***system()***, dove la prima consente di interrogare le variabili di ambiente (nel senso inteso nei sistemi

Unix ed equivalenti) e la seconda consente di eseguire dei comandi attraverso la shell.

Nel documento che descrive lo standard del linguaggio C, il concetto viene generalizzato, ma in pratica, il contesto da cui derivano queste funzioni è quello dei sistemi Unix.

```
char *getenv (const char *name);
int   system (const char *string);
```

La funzione ***getenv()*** si aspetta di ricevere come argomento il nome di una variabile di ambiente (o di qualcosa di comparabile, nel contesto di un altro tipo di sistema operativo), restituendo il puntatore al contenuto di tale variabile. La funzione ***system()*** può essere usata indicando un puntatore nullo e in tal caso restituisce un valore diverso da zero se il sistema operativo è in grado di recepire dei comandi testuali. Se invece viene passata una stringa, la funzione tenta di farla eseguire come comando del sistema operativo: in un sistema Unix o equivalente si tratta di un comando che deve essere eseguito da '`/bin/sh`'. L'esito della funzione ***system()*** dipende da quello del comando impartito e generalmente si ottiene lo stesso valore restituito dal comando eseguito.

Si vedano le pagine di manuale *getenv(3)* e *system(3)*.

69.9.9 Funzioni di ricerca e riordino

Il file '`stdlib.h`' prevede la dichiarazione di due funzioni per il riordino degli array e per la ricerca all'interno di array ordinati. Si tratta precisamente delle funzioni ***qsort()*** e ***bsearch()***, dove i nomi richiamano evidentemente gli algoritmi tradizionali noti come *quick sort* e *binary search*.

Le funzioni della libreria standard generalizzano il problema dell'ordinamento e della ricerca utilizzando puntatori di tipo '**void ***' e scandendo la memoria a blocchi di una dimensione determinata. Ma dal momento che l'area di memoria da scandire non ha la personalità di un array di un qualche tipo, occorre fornire a entrambe queste funzioni il puntatore a una funzione diversa, in grado di confrontare due valori nel contesto di proprio interesse.

```
void qsort (void *base,
            size_t nmemb,
            size_t size,
            int (*compar) (const void *, const void *));

void *bsearch (const void *key,
               const void *base,
               size_t nmemb,
               size_t size,
               int (*compar) (const void *, const void *));
```

Prima di descrivere il significato dei parametri delle due funzioni, conviene vedere un esempio in cui queste si utilizzano. Per la precisione viene scandito un piccolo array di elementi di tipo '**int**': prima viene ordinato, poi si cerca un elemento al suo interno.

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

int confronta (const void *a, const void *b)
{
  int x = *((int *) a);
  int y = *((int *) b);
  return x - y;
}

int main (void)
{
  int a[] = {3, 1, 5, 2};
  int cercato = 5;
  void *p;

  qsort (&a[0], 4, sizeof (int), confronta);
  printf ("%i %i %i %i\n", a[0], a[1], a[2], a[3]);

  p = bsearch (&cercato, &a[0], sizeof (int), 4, confronta);

  printf ("&a[0] = %u; \"%i\" si trova in %u.\n",
          (unsigned int) &a[0], cercato, (unsigned int) p);
```

```
  return 0;
}
```

Nell'esempio viene dichiarata la funzione ***confronta()*** che riceve due argomenti e restituisce un valore che può essere: minore, pari o maggiore di zero, se il primo argomento, rispetto al secondo, è minore, pari o maggiore. Questo è il modo in cui deve comportarsi la funzione da passare come argomento a ***qsort()*** e a ***bsearch()***, tenendo conto che è da tali funzioni che riceve gli argomenti.

La funzione ***qsort()*** vuole ricevere il puntatore alla prima posizione in memoria da riordinare (il parametro ***base***), la quantità degli elementi da riordinare (***nmemb***, ovvero *Number of memory blocks*), la dimensione di tali elementi (***size***) e la funzione da usare per la loro comparazione.

La funzione ***bsearch()*** vuole ricevere il puntatore alla chiave di ricerca (il parametro ***key***), il puntatore alla prima posizione in memoria da scandire (***base***), la quantità degli elementi da scandire (***nmemb***), la dimensione di tali elementi (***size***) e la funzione da usare per la loro comparazione, tenendo conto che questa riceve la chiave di ordinamento come primo argomento.

L'esempio mostrato esegue un ordinamento crescente e il testo visualizzato che si ottiene deve essere simile a quello seguente:

```
1 2 3 5
&a[0] = 3218927260; "5" si trova in 3218927272.
```

È sufficiente invertire il risultato della funzione di comparazione per ottenere un ordinamento decrescente e per scandire un array ordinato in modo decrescente:

```
int confronta (const void *a, const void *b)
{
  int x = *((int *) a);
  int y = *((int *) b);
  return y - x;
}
```

In tal caso, il testo che viene emesso deve essere simile a quello seguente:

```
5 3 2 1
&a[0] = 3218593340; "5" si trova in 3218593340.
```

69.9.10 Funzioni per l'aritmetica con i numeri interi

Un gruppo di funzioni il cui nome termina per *...**abs()*** si occupa di calcolare il valore assoluto di un numero intero. Le funzioni sono precisamente: ***abs()*** per gli interi di tipo '**int**', ***labs()*** per gli interi di tipo '**long int**' e ***llabs()*** per gli interi di tipo '**long long int**'.

```
int abs             (int j);
long int labs       (long int j);
long long int llabs (long long int j);
```

Evidentemente, la realizzazione di queste funzioni è estremamente banale. Viene presentato solo il caso di ***abs()***:

```
#include <stdlib.h>
int
abs (int j)
{
  if (j < 0)
    {
      return -j;
    }
  else
    {
      return j;
    }
}
```

Un gruppo di funzioni il cui nome termina per *...**div()*** si occupa di dividere due interi, calcolando il quoziente e il resto. Le funzioni sono precisamente: ***div()*** per gli interi di tipo '**int**', ***ldiv()*** per gli interi di tipo '**long int**' e ***lldiv()*** per gli interi di tipo '**long long int**'. Il risultato viene restituito in una variabile strutturata che contiene sia il quoziente, sia il resto: rispettivamente si tratta dei tipi '**div_t**', '**ldiv_t**' e '**lldiv_t**'.

```
div_t   div   (int numer, int denom);
ldiv_t  ldiv  (long int numer, long int denom);
lldiv_t lldiv (long long int numer, long long int denom);
```

I tre tipi creati appositamente per contenere il risultato di queste funzioni contengono i membri '**quot**' e '**rem**' che rappresentano, rispettivamente, il quoziente e il resto. Anche la realizzazione di queste funzioni è molto semplice banale. Viene presentato solo il caso di ***div()***:

```
#include <stdlib.h>
div_t
div (int numer, int denom)
{
  div_t d;
  d.quot = numer / denom;
  d.rem = numer % denom;
  return d;
}
```

69.9.11 Funzioni per la gestione di caratteri estesi e sequenze multibyte

Il linguaggio C distingue tra una gestione dei caratteri basata sul byte, tale da consentire la gestione di un insieme minimo, come quello della codifica ASCII, e una gestione a byte multipli, o multibyte. Per esempio, la codifica UTF-8 è ciò che si intende per «multibyte», ma esistono anche altre codifiche che sfruttano questo meccanismo.

Quando il contesto richiede l'interpretazione dei byte secondo una codifica multibyte, è necessario stabilire un punto di riferimento per iniziare l'interpretazione e occorre poterne conservare lo stato quando la lettura di un carattere viene interrotta e ripresa a metà. Nella documentazione dello standard, nell'ambito delle sequenze multibyte, lo stato viene definito *shift state*.

Per gestire internamente la codifica universale, il C utilizza un tipo specifico, '**wchar_t**', corrispondente a un intero di rango sufficiente a rappresentare tutti i caratteri che si intendono gestire. Di conseguenza, le stringhe letterali, precedute dalla lettera '**L**' (per esempio '**L"àèìòùé"**'), sono array di elementi '**wchar_t**'.

Le funzioni che riguardano la gestione di caratteri estesi e sequenze multibyte del file '**stdlib.h**', servono principalmente per convertire sequenze multibyte nel tipo '**wchar_t**' e viceversa. Tuttavia, occorre tenere presente che la configurazione locale deve essere tale da prevedere l'uso di caratteri da rappresentare attraverso sequenze multibyte, altrimenti le conversioni diventano prive di utilità.

Listato 69.102. Prototipi delle funzioni relative alla gestione multibyte.

```
int    mblen    (const char *s, size_t n);
int    mbtowc   (wchar_t *restrict pwc,
                 const char *restrict s,
                 size_t n);
int    wctomb   (char *s, wchar_t wc);
size_t mbstowcs (wchar_t *restrict pwcs,
                 const char *restrict s,
                 size_t n);
size_t wcstombs (char *restrict s,
                 const wchar_t *restrict pwcs,
                 size_t n);
```

69.9.12 Funzione «mblen()»

La funzione ***mblen()*** si usa normalmente per contare quanti byte sono presenti nella stringa ***s*** fornita come primo argomento, per comporre il primo carattere (multibyte) della stringa stessa, limitando la scansione a un massimo di ***n*** byte (il secondo argomento richiesto). Se al posto di indicare una stringa si fornisce il puntatore nullo, si ottiene un valore che può essere uno o zero, a seconda che sia prevista o meno una codifica multibyte con una gestione dello stato (*shift state*).

```
#include <locale.h>
#include <stdio.h>
```

```
#include <stdlib.h>

int main (void)
{
  int n;
  setlocale (LC_ALL, "en_US.UTF-8");
  n = mblen (NULL, 0);
  printf ("Gestione dello stato: %i\n", n);
  n = mblen ("€", 8);
  printf ("Il carattere %s richiede %i byte.\n", "€", n);
  return 0;
}
```

L'esempio mostrato dovrebbe chiarire alcune cose. La funzione richiede un argomento di tipo stringa di caratteri, perché un argomento di tipo '**char**' singolo, non consentirebbe di annotare una sequenza multibyte. Le sequenze multibyte sono stringhe normali, trattate come tali, salvo quando è necessario interpretare il loro contenuto; a questo proposito, si vede che la funzione ***printf()*** riceve una stringa multibyte e si limita a trattarla come una stringa normale.

Se la codifica in cui è scritto il sorgente è la stessa usata dal programma durante il suo funzionamento, si può ottenere il testo seguente:

```
Gestione dello stato: 0
Il carattere € richiede 3 byte.
```

Nell'uso normale della funzione ***mblen()***, se la stringa che si fornisce contiene una sequenza multibyte errata o incompleta, il valore restituito è `-1`.

69.9.13 Funzioni «mbtowc()» e «wctomb()»

Le due funzioni ***mbtowc()*** e ***wctomb()*** si compensano a vicenda, fornendo il mezzo elementare di conversione dei caratteri da una sequenza multibyte a un numero intero di tipo '**wchar_t**' e viceversa. Quando a queste funzioni, al posto del puntatore alla stringa multibyte, si fornisce il puntatore nullo, si ottiene un funzionamento analogo a quello di ***mblen()***, con un valore pari a uno se la configurazione locale prevede l'uso di sequenze multibyte e una gestione dello stato, oppure zero se questo problema non sussiste. Inoltre, per entrambe le funzioni, se la sequenza multibyte è errata o incompleta, si ottiene la restituzione del valore `-1`. Infine, se la conversione ha successo, si ottiene la quantità dei byte che compongono la sequenza multibyte (di origine o di destinazione, a seconda della funzione usata).

Viene mostrato un esempio molto semplice che dimostra l'uso delle due funzioni. In particolare viene convertita la sequenza multibyte che rappresenta la lettera «ä» in un numero '**wchar_t**', quindi il numero viene incrementato e riconvertito in una nuova sequenza multibyte, per ottenere il carattere «å».

```
#include <locale.h>
#include <stdlib.h>
#include <stdio.h>

int main (void)
{
  int n;
  wchar_t wc;
  char mb[20] = {}; // Inizializza l'array a zero.

  setlocale (LC_ALL, "en_US.UTF-8");

  n = mbtowc (&wc, NULL, 8);
  printf ("Gestione dello stato: %i\n", n);
  n = wctomb (NULL, wc);
  printf ("Gestione dello stato: %i\n", n);

  n = mbtowc (&wc, "ä", 8);
  printf ("Il carattere \"ä\" si rappresenta con %i byte ",
          n);
  printf ("in una sequenza multibyte e con il numero %i ",
          (int) wc);
  printf ("in una variabile di tipo \"wchar_t\".\n");
```



```
  wc++;
  n = wctomb (mb, wc);
  printf ("Il carattere \"%s\" si rappresenta con %i byte ",
          mb, n);
  printf ("in una sequenza multibyte e con il numero %i ",
          (int) wc);
  printf ("in una variabile di tipo \"wchar_t\".\n");

  return 0;
}
```

Si dovrebbe ottenere un testo come quello seguente:

```
Gestione dello stato: 0
Gestione dello stato: 0
Il carattere "ä" si rappresenta con 2 byte in una ↩
↪sequenza multibyte e con il numero 228 ↩
↪in una variabile di tipo "wchar_t".
Il carattere "å" si rappresenta con 2 byte in una ↩
↪sequenza multibyte e con il numero 229 ↩
↪in una variabile di tipo "wchar_t".
```

Per gli approfondimenti eventuali, si vedano le pagine di manuale *mbtowc(3)* e *wctomb(3)*.

69.9.14 Funzioni «mbstowcs()» e «wcstombs()»

Le funzioni ***mbstowcs()*** e '**wcstombs**' servono rispettivamente per convertire una stringa multibyte in un stringa estesa (un array di elementi '**wchar_t**') e per fare l'opposto. Entrambe le funzioni richiedono tre argomenti: l'array di destinazione, l'array di origine e la quantità di elementi da utilizzare nell'array di destinazione. Entrambe le funzioni restituiscono un numero che esprime la quantità di elementi di destinazione convertiti, escluso ciò che costituisce il carattere nullo di terminazione. Entrambe restituiscono un valore pari a '**(size_t) (-1)**' se la conversione produce un errore.[3]

Per convertire correttamente una stringa (multibyte o estesa), occorre che il numero di elementi di destinazione previsto includa anche il carattere nullo di terminazione. L'esempio seguente dovrebbe aiutare a comprendere il problema:

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
  size_t n;
  wchar_t wca[] = {1, 2, 3, 4, 5, 6, 7};
  wchar_t wcb[] = {1, 2, 3, 4, 5, 6, 7};
  char mba[] = "**************";
  char mbb[] = "**************";

  setlocale (LC_ALL, "en_US.UTF-8");

  n = mbstowcs (wca, "ääå", 3);
  printf ("mbstowcs: %i: %i %i %i %i %i\n", n,
          wca[0], wca[1], wca[2], wca[3], wca[4]);

  n = mbstowcs (wcb, "ääå", 5);
  printf ("mbstowcs: %i: %i %i %i %i %i\n", n,
          wcb[0], wcb[1], wcb[2], wcb[3], wcb[4]);

  n = wcstombs (mba, L"ääå", 6);
  printf ("wcstombs: %i: \"%s\"\n", n, mba);

  n = wcstombs (mbb, L"ääå", 9);
  printf ("wcstombs: %i: \"%s\"\n", n, mbb);

  return 0;
}
```

Nell'esempio, la funzione ***mbstowcs()*** viene usata due volte, per convertire una stringa multibyte, composta da tre caratteri, se non si conta quello di terminazione. Nel primo caso, viene specificato che si vogliono convertire esattamente tre caratteri, ma questo significa che nell'array di destinazione rimane il contenuto originale a partire dal quarto elemento. In modo analogo, la funzione ***wcstombs()***

viene usata due volte per convertire una stringa estesa in una stringa multibyte. La stringa estesa si compone di tre caratteri che nella conversione vanno a occupare esattamente sei byte, con l'aggiunta eventuale del carattere nullo di terminazione (che sarebbe il settimo). Si può vedere che quando si chiede una conversione di sei elementi, la stringa ricevente mantiene il contenuto precedente nella parte restante. Ecco cosa si dovrebbe vedere eseguendo il programma:

```
mbstowcs: 3: 228 229 226 4 5
mbstowcs: 3: 228 229 226 0 5
wcstombs: 6: "ääâ*******"
wcstombs: 6: "ääâ"
```

Se al posto della destinazione (il primo argomento) viene posto il puntatore nullo, si ottiene la simulazione dell'operazione, senza memorizzare alcunché e senza tenere conto della quantità massima di elementi che si annota come ultimo argomento. Ciò ha lo scopo di contare quanti elementi servirebbero per produrre una conversione completa. L'esempio seguente modifica quello già visto, sfruttando questa funzionalità:

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
  size_t n;
  size_t max;
  wchar_t wca[] = {1, 2, 3, 4, 5, 6, 7};
  char mba[] = "**************";      // 15 byte total.

  setlocale (LC_ALL, "en_US.UTF-8");

  max = mbstowcs (NULL, "ääâ", 0);
  if (max <= 6)
    {
      n = mbstowcs (wca, "ääâ", max + 1);
      printf ("mbstowcs: %i: %i %i %i %i %i\n", n,
              wca[0], wca[1], wca[2], wca[3], wca[4]);
    }

  max = wcstombs (NULL, L"ääâ", 0);
  if (max <= 14)
    {
      n = wcstombs (mba, L"ääâ", max + 1);
      printf ("wcstombs: %i: \"%s\"\n", n, mba);
    }

  return 0;
}
```

Ecco cosa si dovrebbe vedere eseguendo il programma:

```
mbstowcs: 3: 228 229 226 0 5
wcstombs: 6: "ääâ"
```

Per gli approfondimenti eventuali, si vedano le pagine di manuale *mbstowcs(3)* e *wcstombs(3)*.

69.10 File «inttypes.h»

Il file '`inttypes.h`' della libreria standard serve principalmente a completare le funzionalità di '`stdint.h`', per ciò che riguarda la gestione dei valori numerici interi, il cui rango è controllabile. Infatti, il problema principale nell'uso di interi definiti in modo alternativo allo standard del linguaggio C, privo di librerie, sta nell'uso appropriato degli specificatori di conversione nelle funzioni come ***printf()*** e ***scanf()***. È proprio per risolvere questo problema che nel file '`inttypes.h`' vanno definite, soprattutto, delle macro-variabili da usare in sostituzione degli specificatori di conversione basati sui tipi elementari (si veda eventualmente la realizzazione del file '`inttypes.h`', ma senza le funzioni che lo riguardano, nei sorgenti di os32, listato 95.8).[4]

Gli esempi proposti per descrivere la libreria che fa capo al file '`inttypes.h`' si riferiscono a quanto già definito nella sezione 69.4 a proposito del file '`stdint.h`'.

Inizialmente, il file '`inttypes.h`' deve includere '`stdint.h`', inoltre dichiara il tipo '**`wchar_t`**', già descritto nel file '`stddef.h`':

```
#include <stdint.h>
typedef unsigned int wchar_t;
```

69.10.1 Divisione intera con interi di rango massimo

Nel file '`inttypes.h`' viene definito il tipo '**`imaxdiv_t`**' che va ad affiancarsi ai tipi '**`div_t`**', '**`ldiv_t`**' e '**`lldiv_t`**', definiti nel file '`stdlib.h`'. In pratica si tratta di una struttura il cui scopo è quello di contenere il risultato di una divisione, espresso come quoziente e resto, quando il tipo intero usato è quello massimo:

```
typedef struct {
    intmax_t quot;
    intmax_t rem;
} imaxdiv_t;
```

Il tipo strutturato '**`imaxdiv_t`**' serve alle funzioni ***imaxdiv()*** e ***uimaxdiv()***, le quali sono sostanzialmente equivalenti alle altre funzioni ...***div()*** del file '`stdlib.h`':

```
imaxdiv_t imaxdiv (intmax_t numer, intmax_t denom);
```

```
#include <inttypes.h>
imaxdiv_t
imaxdiv   (intmax_t numer, intmax_t denom)
{
  imaxdiv_t d;
  d.quot = numer / denom;
  d.rem = numer % denom;
  return d;
}
```

69.10.2 Macro-variabili in qualità di specificatori di conversione

Come accennato all'inizio del capitolo, per poter usare le funzioni ...***printf()*** e ...***scanf()***, occorrono degli specificatori di conversione, ma non ne esistono per i tipi interi a rango controllato, pertanto, per questi, servono delle macro-variabili coerenti con il tipo relativo.

Le macro che iniziano per ***PRI***... si usano come parte terminale di specificatori di conversione per la composizione dell'output (...***printf()***), mentre le macro che iniziano per ***SCN***... sono adatte per l'interpretazione dell'input (...***scanf()***).

Le macro '**`PRI`*`x`*`n`**' e '**`SCN`*`x`*`n`**' terminano gli specificatori di conversione '**`%`**...*`x`*', per i tipi interi '**`[u]int`*`n`*`_t`**'; le macro '**`PRI`*`x`*`LEAST`*`n`***' e '**`SCN`*`x`*`LEAST`*`n`***' riguardano i tipi '**`[u]int_least`*`n`*`_t`**'; le macro '**`PRI`*`x`*`FAST`*`n`***' e '**`SCN`*`x`*`FAST`*`n`***' riguardano i tipi '**`[u]int_fast`*`n`*`_t`**'; le macro '**`PRI`*`x`*`MAX`*`n`***' e '**`SCN`*`x`*`MAX`*`n`***' riguardano i tipi '**`[u]intmax_t`**'; le macro '**`PRI`*`x`*`PTR`*`n`***' e '**`SCN`*`x`*`PTR`*`n`***' riguardano i tipi '**`[u]intptr_t`**'.

L'esempio seguente dovrebbe dimostrare il significato di queste macro-variabili, attraverso l'uso di ***printf()***:

```
#include <stdio.h>
#include <inttypes.h>
int
main (int argc, char *argv[])
{
  uint64_t num = INT64_C(1234567890);
  printf ("Il valore della variabile \"num\" "
          "corrisponde a "
          "%020" PRIu64 ".\n", num);
  return 0;
}
```

Il listato seguente mostra come possono essere dichiarate queste macro-variabili:

```
// Composizione dell'output.

#define PRId8           "d"
#define PRId16          "d"
#define PRId32          "d"
#define PRId64          "lld"
```

```
#define PRIdLEAST8      "d"
#define PRIdLEAST16     "d"
#define PRIdLEAST32     "d"
#define PRIdLEAST64     "lld"

#define PRIdFAST8       "d"
#define PRIdFAST16      "d"
#define PRIdFAST32      "d"
#define PRIdFAST64      "lld"

#define PRIdMAX         "lld"
#define PRIdPTR         "d"

#define PRIi8           "i"
#define PRIi16          "i"
#define PRIi32          "i"
#define PRIi64          "lli"

#define PRIiLEAST8      "i"
#define PRIiLEAST16     "i"
#define PRIiLEAST32     "i"
#define PRIiLEAST64     "lli"

#define PRIiFAST8       "i"
#define PRIiFAST16      "i"
#define PRIiFAST32      "i"
#define PRIiFAST64      "lli"

#define PRIiMAX         "lli"
#define PRIiPTR         "i"

#define PRIo8           "o"
#define PRIo16          "o"
#define PRIo32          "o"
#define PRIo64          "llo"

#define PRIoLEAST8      "o"
#define PRIoLEAST16     "o"
#define PRIoLEAST32     "o"
#define PRIoLEAST64     "llo"

#define PRIoFAST8       "o"
#define PRIoFAST16      "o"
#define PRIoFAST32      "o"
#define PRIoFAST64      "llo"

#define PRIoMAX         "llo"
#define PRIoPTR         "o"

#define PRIu8           "u"
#define PRIu16          "u"
#define PRIu32          "u"
#define PRIu64          "llu"

#define PRIuLEAST8      "u"
#define PRIuLEAST16     "u"
#define PRIuLEAST32     "u"
#define PRIuLEAST64     "llu"

#define PRIuFAST8       "u"
#define PRIuFAST16      "u"
#define PRIuFAST32      "u"
#define PRIuFAST64      "llu"

#define PRIuMAX         "llu"
#define PRIuPTR         "u"

#define PRIx8           "x"
#define PRIx16          "x"
#define PRIx32          "x"
#define PRIx64          "llx"

#define PRIxLEAST8      "x"
#define PRIxLEAST16     "x"
#define PRIxLEAST32     "x"
#define PRIxLEAST64     "llx"

#define PRIxFAST8       "x"
#define PRIxFAST16      "x"
#define PRIxFAST32      "x"
#define PRIxFAST64      "llx"
```



```
#define PRIxMAX         "llx"
#define PRIxPTR         "x"

#define PRIX8           "X"
#define PRIX16          "X"
#define PRIX32          "X"
#define PRIX64          "llX"

#define PRIXLEAST8      "X"
#define PRIXLEAST16     "X"
#define PRIXLEAST32     "X"
#define PRIXLEAST64     "llX"

#define PRIXFAST8       "X"
#define PRIXFAST16      "X"
#define PRIXFAST32      "X"
#define PRIXFAST64      "llX"

#define PRIXMAX         "llX"
#define PRIXPTR         "X"

// Interpretazione dell'input.

#define SCNd8           "hhd"
#define SCNd16          "hd"
#define SCNd32          "d"
#define SCNd64          "lld"

#define SCNdLEAST8      "hhd"
#define SCNdLEAST16     "hd"
#define SCNdLEAST32     "d"
#define SCNdLEAST64     "lld"

#define SCNdFAST8       "hhd"
#define SCNdFAST16      "d"
#define SCNdFAST32      "d"
#define SCNdFAST64      "lld"

#define SCNdMAX         "lld"
#define SCNdPTR         "d"

#define SCNi8           "hhi"
#define SCNi16          "hi"
#define SCNi32          "i"
#define SCNi64          "lli"

#define SCNiLEAST8      "hhi"
#define SCNiLEAST16     "hi"
#define SCNiLEAST32     "i"
#define SCNiLEAST64     "lli"

#define SCNiFAST8       "hhi"
#define SCNiFAST16      "i"
#define SCNiFAST32      "i"
#define SCNiFAST64      "lli"

#define SCNiMAX         "lli"
#define SCNiPTR         "i"

#define SCNo8           "hho"
#define SCNo16          "ho"
#define SCNo32          "o"
#define SCNo64          "llo"

#define SCNoLEAST8      "hho"
#define SCNoLEAST16     "ho"
#define SCNoLEAST32     "o"
#define SCNoLEAST64     "llo"

#define SCNoFAST8       "hho"
#define SCNoFAST16      "o"
#define SCNoFAST32      "o"
#define SCNoFAST64      "llo"

#define SCNoMAX         "llo"
#define SCNoPTR         "o"

#define SCNu8           "hhu"
#define SCNu16          "hu"
#define SCNu32          "u"
```

```
#define SCNu64          "llu"

#define SCNuLEAST8      "hhu"
#define SCNuLEAST16     "hu"
#define SCNuLEAST32     "u"
#define SCNuLEAST64     "llu"

#define SCNuFAST8       "hhu"
#define SCNuFAST16      "u"
#define SCNuFAST32      "u"
#define SCNuFAST64      "llu"

#define SCNuMAX         "llu"
#define SCNuPTR         "u"

#define SCNx8           "hhx"
#define SCNx16          "hx"
#define SCNx32          "x"
#define SCNx64          "llx"

#define SCNxLEAST8      "hhx"
#define SCNxLEAST16     "hx"
#define SCNxLEAST32     "x"
#define SCNxLEAST64     "llx"

#define SCNxFAST8       "hhx"
#define SCNxFAST16      "x"
#define SCNxFAST32      "x"
#define SCNxFAST64      "llx"

#define SCNxMAX         "llx"
#define SCNxPTR         "x"
```

69.10.3 Valore assoluto

Nel file '`stdlib.h`' si trovano dichiarate alcune funzioni per il calcolo del valore assoluto: ...***abs()***. Nel file '`inttypes.h`' si aggiunge la funzione ***imaxabs()***, da usare per i valori interi massimi:

```
intmax_t imaxabs (intmax_t j);
```

```
#include <inttypes.h>
intmax_t
imaxabs (intmax_t j)
{
  if (j < 0)
    {
      return -j;
    }
  else
    {
      return j;
    }
}
```

69.10.4 Conversione da stringa a numero intero

Per convertire una stringa contenente un valore numerico intero, quando si vuole fare riferimento all'intero di dimensione massima, si possono usare le funzioni ***strtoimax()***, ***strtouimax()***, ***wcstoimax()*** e ***wcstouimax()***, dichiarate nel file '`inttypes.h`'. Come il nome suggerisce, le prime due funzioni sono destinate alla conversione di stringhe «normali», mentre le altre sono specifiche per le stringhe estese.

Evidentemente si tratta di funzioni che si abbinano alle altre ***strto...()*** del file '`stdlib.h`' e alle funzioni ***wcsto...()*** del file '`wchar.h`'.

```
intmax_t  strtoimax  (const char *restrict nptr,
                      char **restrict endptr, int base);
uintmax_t strtouimax (const char *restrict nptr,
                      char **restrict endptr, int base);
intmax_t  wcstoimax  (const wchar_t *restrict nptr,
                      wchar_t **restrict endptr, int base);
uintmax_t wcstouimax (const wchar_t *restrict nptr,
                      wchar_t **restrict endptr, int base);
```

Come si vede, i parametri delle funzioni sono gli stessi; quello che cambia è il tipo di stringa, che nelle funzioni ***strto...()*** è normale, mentre nelle funzioni ***wcsto...()*** è di tipo esteso. Nel caso di funzioni

...***touimax()*** si ottiene un valore intero senza segno, mentre con le funzioni ...***toimax()*** si ottiene un valore intero con segno.

Il comportamento di queste funzioni è analogo a quello delle altre funzioni ***strto...()*** e ***wcsto...()***, per ciò che riguarda l'interpretazione di valori interi, con la differenza che si fa riferimento al valore intero più grande. Il valore restituito è zero se non si può procedere alla conversione; se invece il valore è al di fuori dell'intervallo rappresentabile, a seconda dei casi si può avere il valore corrispondente a ***INTMAX_MAX***, ***INTMAX_MIN*** o ***UINTMAX_MIN***, con l'aggiornamento della variabile ***errno*** al valore rappresentato da ***ERANGE***.

69.11 File «iso646.h»

Il file '`iso646.h`' della libreria standard definisce alcune macro-variabili da usare in sostituzione di simboli che potrebbero mancare nella propria tastiera, anche se ciò è comunque poco probabile.[5]

Macro	Corrispondenza	Codice
`and`	`&&`	`#define and    &&`
`and_eq`	`&=`	`#define and_eq &=`
`bitand`	`&`	`#define bitand &`
`bitor`	`|`	`#define bitor  |`
`compl`	`~`	`#define compl  ~`
`not`	`!`	`#define not    !`
`not_eq`	`!=`	`#define not_eq !=`
`or`	`||`	`#define or     ||`
`or_eq`	`|=`	`#define or_eq  |=`
`xor`	`^`	`#define xor    ^`
`xor_eq`	`^=`	`#define xor_eq ^=`

69.12 File «stdbool.h»

Il file '`stdbool.h`' della libreria standard definisce alcune macro-variabili da usare per la gestione dei valori logici (*Vero* e *Falso*); in particolare consente di utilizzare il nome ***bool*** al posto di '**`_Bool`**' (si veda eventualmente la realizzazione di questo file nei sorgenti di os32, listato 95.1.11).[6].

```
#define bool    _Bool
#define true    1
#define false   0
#define __bool_true_false_are_defined   1
```

Come si può vedere, la macro-variabile ***__bool_true_false_are_defined*** consente di sapere se le macro-variabili ***bool***, ***true*** e ***false***, sono definite.

69.13 File «stddef.h»

Il file '`stddef.h`' della libreria standard definisce alcuni tipi di dati e delle macro fondamentali (si veda eventualmente la realizzazione del file '`stddef.h`' nei sorgenti di os32, listato 95.1.12).[7]

```
typedef long int          ptrdiff_t;
typedef unsigned long int size_t;
typedef unsigned int      wchar_t;

#define NULL                    0
#define offsetof(TYPE, MEMBER)  \
    ((size_t) &((TYPE *)0)->MEMBER)
```

Di tutte le definizioni merita attenzione la macroistruzione '**`offsetof`**' che serve a misurare lo scostamento di un membro di una struttura, per la quale è il caso di scomporre i suoi componenti:

- l'espressione '**((*tipo_struttura* *)0)**' rappresenta un puntatore nullo trasformato, con un cast, in un puntatore nullo al tipo di struttura alla quale si sta facendo riferimento;
- l'espressione '**((*tipo_struttura* *)0)->*nome_membro***' rappresenta il contenuto del membro indicato, preso a partire dall'indirizzo zero;
- l'espressione '**&((*tipo_struttura* *)0)->*nome_membro***' rappresenta l'indirizzo del membro indicato, preso a partire dall'indirizzo zero.

Pertanto, l'indirizzo del membro, relativo all'indirizzo zero, corrisponde anche al suo scostamento a partire dall'inizio della struttura. Così, tale valore viene convertito con un cast nel tipo '**size_t**'.

69.14 File «string.h»

Il file '`string.h`' della libreria standard definisce il tipo '**size_t**', la macro-variabile ***NULL*** (come dal file '`stddef.h`', descritto nella sezione 69.13) e una serie di funzioni per il trattamento delle stringhe o comunque di sequenze di caratteri (si veda eventualmente la realizzazione del file '`string.h`' e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.20)).

69.14.1 Copia

Seguono i prototipi delle funzioni disponibili per la copia:

```
void *memcpy  (void *restrict dst,
               const void *restrict org, size_t n);
void *memmove (void *dst, const void *org, size_t n);
char *strcpy  (char *restrict dst,
               const char *restrict org);
char *strncpy (char *restrict dst,
               const char *restrict org, size_t n);
```

Lo standard POSIX aggiunge anche i prototipi seguenti:

```
void *memccpy (void *restrict dst, const void *restrict org,
               int c, size_t n);
char *strdup  (const char *org);
```

69.14.1.1 Funzione «memcpy()» (memory copy)

La funzione ***memcpy()*** copia ***n*** caratteri a partire dall'indirizzo indicato da ***org***, per riprodurli a partire dall'indirizzo ***dst***, alla condizione che i due insiemi non risultino sovrapposti. La funzione restituisce l'indirizzo ***dst***.

```
#include <string.h>
void *
memcpy (void *restrict dst, const void *restrict org,
        size_t n)
{
  unsigned char *d = (unsigned char *) dst;
  unsigned char *o = (unsigned char *) org;
  size_t i;
  for (i = 0; n > 0 && i < n; i++)
    {
      d[i] = o[i];
    }
  return dst;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove la variabile ***y*** viene sovrascritta dal contenuto di ***x***, ma questo attraverso la copia dei byte (si intende che gli interi siano da 32 bit).

```
#include <stdio.h>
#include <string.h>

int
main (void)
{
  int  x = 0x12345678;
  int  y = 0xFFFFFFFF;
  printf ("prima: %x\n", y);
  memcpy (&y, &x, sizeof (int));
  printf ("dopo:  %x\n", y);
  return 0;
}
```


Avviando questo programma si deve ottenere un risultato come quello seguente:

```
prima: ffffffff
dopo:  12345678
```

69.14.1.2 Funzione «memccpy()»

La funzione ***memccpy()*** appartiene allo standard POSIX e si distingue rispetto a ***memcpy()*** perché la copia termina al raggiungimento di un certo carattere (il parametro ***c***), restituendo il puntatore alla posizione successiva nella destinazione.

```
#include <string.h>
void *
memccpy (void *restrict dst, const void *restrict org,
         int c, size_t n)
{
  unsigned char *d = (unsigned char *) dst;
  unsigned char *o = (unsigned char *) org;
  size_t i;
  for (i = 0; n > 0 && i < n; i++)
    {
      d[i] = o[i];
      if (o[i] == (unsigned char) c)
        {
          return (dst + i + 1);
        }
    }
  return NULL;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove l'array ***y*** viene sovrascritto dal contenuto di ***x***, fino a quando si raggiunge il carattere '**7**'. Nell'esempio vengono anche visualizzati i valori dei puntatori di ***y*** e della posizione raggiunta all'interno di ***y*** alla fine della copia, sempre in forma di puntatore.

```
#include <stdio.h>
#include <string.h>
#include <inttypes.h>

int
main (void)
{
  char x[11] = { '0', '1', '2', '3', '4', '5',
                 '6', '7', '8', '9', '\0'};
  char y[11] = { 'a', 'b', 'c', 'd', 'e', 'f',
                 'g', 'h', 'i', 'j', '\0'};
  char *z;
  printf ("prima: \"%s\" %" PRIuPTR "\n", y, (uintptr_t) y);
  z = memccpy (y, x, (int) '7', (size_t) 11);
  printf ("dopo:  \"%s\" %" PRIuPTR "\n", y, (uintptr_t) z);
  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
prima: "abcdefghij" 3218609914
dopo:  "01234567ij" 3218609922
```

Come si può vedere dal risultato, l'array ***y*** inizia a partire dal puntatore 3218609914 e, alla fine del trasferimento parziale dall'array ***x***, che si ferma al carattere '**7**', la funzione restituisce il puntatore 3218609922 che individua la posizione successiva al carattere copiato, corrispondente in pratica al carattere '**i**'.

69.14.1.3 Funzione «memmove()» (memory move)

La funzione ***memmove()*** opera in modo simile a ***memcpy()***, con la differenza che le due aree di memoria coinvolte possono sovrapporsi. In pratica la copia avviene prima in un'area temporanea, quindi,

dall'area temporanea viene ricopiata nella destinazione. La funzione restituisce l'indirizzo ***dst***.

```
#include <string.h>
void *
memmove (void *dst, const void *org, size_t n)
{
  char temp[n];
  unsigned char *d = (unsigned char *) dst;
  unsigned char *o = (unsigned char *) org;
  size_t i;
  for (i = 0; i < n; i++)
    {
      temp[i] = o[i];
    }
  for (i = 0; n > 0 && i < n; i++)
    {
      d[i] = temp[i];
    }
  return dst;
}
```

Per osservare il comportamento della funzione si può riutilizzare lo stesso programma usato per ***memcpy()***, con la modifica del nome della funzione chiamata. Il risultato atteso è lo stesso:

```
…
  printf ("prima: %x\n", y);
  memmove (&y, &x, sizeof (int));
  printf ("dopo:  %x\n", y);
…
```

69.14.1.4 Funzione «strcpy()» (string copy)

La funzione ***strcpy()*** copia la stringa ***org*** nell'array a cui punta ***dst***, includendo anche il carattere zero di conclusione delle stringhe, alla condizione che le due stringhe non si sovrappongano. La funzione restituisce ***dst***.

```
#include <string.h>
char *
strcpy (char *restrict dst, const char *restrict org)
{
  size_t i;
  for (i = 0; org[i] != 0; i++)
    {
      dst[i] = org[i];
    }
  dst[i] = 0;
  return dst;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove l'array ***y*** viene sovrascritto dal contenuto di ***x***.

```
#include <stdio.h>
#include <string.h>
int main (void)
{
  char x[] = "abcdefghijklmnopqrstuvwxyz";
  char y[50] = "ciao";
  printf ("prima: %s\n", y);
  strcpy (y, x);
  printf ("dopo:  %s\n", y);
  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
prima: ciao
dopo:  abcdefghijklmnopqrstuvwxyz
```

69.14.1.5 Funzione «strncpy()»

La funzione ***strncpy()*** agisce in modo analogo a quello di '**`strcpy`**', con la differenza che la copia riguarda al massimo i primi ***n*** caratteri, includendo in questo anche il carattere nullo di terminazione delle stringhe. Se però la stringa ***org*** è più breve (in quanto si incontra il carattere di terminazione prima di ***n*** caratteri), i caratteri rimasti vengono copiati con un valore a zero nella destinazione. La funzione restituisce ***dst***.

```
#include <string.h>
char *
strncpy (char *restrict dst, const char *restrict org,
         size_t n)
{
  size_t i;
  for (i = 0; n > 0 && i < n && org[i] != 0; i++)
    {
      dst[i] = org[i];
    }
  for ( ; n > 0 && i < n; i++)
    {
      dst[i] = 0;
    }
  return dst;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove l'array ***y*** viene sovrascritto dal contenuto di ***x***.

```
#include <stdio.h>
#include <string.h>
int main (void)
{
  char x[] = "abcdefghijklmnopqrstuvwxyz";
  char y[50] = "ciaociaociaociaociaociaociaociaociao";
  printf ("prima:   %s\n", y);
  strncpy (y, x, 10);
  printf ("durante: %s\n", y);
  strncpy (y, x, 27);
  printf ("dopo:    %s\n", y);
  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
prima:   ciaociaociaociaociaociaociaociaociao
durante: abcdefghijaociaociaociaociaociaociao
dopo:    abcdefghijklmnopqrstuvwxyz
```

69.14.1.6 Funzione «strdup()»

La funzione ***strdup()***, richiesta dallo standard POSIX, è simile a ***strcpy()***, con la differenza che richiede solo l'indicazione della stringa da duplicare, mentre alloca autonomamente la memoria per produrne una copia. Pertanto, la funzione restituisce il puntatore alla stringa duplicata (e allocata) ed è poi possibile liberare la memoria attraverso la funzione ***free()***.

```
#include <string.h>
#include <stdlib.h>
#include <stdint.h>

char *
strdup (const char *org)
{
  char  *d;
  int    i;
  size_t size;

  for (i = 0; i < (SIZE_MAX - 1); i++)
    {
      if (org[i] == '\0')
        {
          break;
        }
    }

  if (i == (SIZE_MAX - 1))
    {
      return NULL;
    }

  size = i + 1;
```

```
  d = malloc (size);

  if (d == NULL)
    {
      return NULL;
    }

  strcpy (d, org);

  return d;
}
```

Nell'esempio proposto, si riutilizza la funzione ***strcpy()*** e la funzione ***malloc()*** per allocare la memoria necessaria. Viene anche usata la macro-variabile ***SIZE_MAX***, per dare un limite massimo alla scansione, nel caso la stringa di origine non contenga il carattere nullo di terminazione. Per questa ragione, diventa necessario includere i file '`stdint.h`' e '`stdlib.h`'.

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove l'array ***o*** viene copiato altrove, associandogli il puntatore ***d***: se l'operazione ha successo, viene visualizzata la stringa a cui punta ***d***, altrimenti si ottiene un messaggio di errore. Dovendo usare la funzione ***free()*** per liberare la memoria, si include anche il file '`stdlib.h`'.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int
main (void)
{
  char *d;
  char  o[] = "abcdefghijklmnopqrstuvwxyz";

  d = strdup (o);

  if (d == NULL)
    {
      printf ("Non è possibile duplicare la stringa!\n");
    }
  else
    {
      printf ("%s\n", d);
      free (d);
    }
  return 0;
}
```

69.14.2 Concatenamento

Seguono i prototipi delle funzioni per il concatenamento:

```
char *strcat  (char *restrict dst,
               const char *restrict org);
char *strncat (char *restrict dst,
               const char *restrict org, size_t n);
```

69.14.2.1 Funzione «strcat()» (string cat)

La funzione ***strcat()*** copia la stringa ***org*** a partire dalla fine della stringa ***dst*** (sovrascrivendo il carattere nullo preesistente), alla condizione che le due stringhe non siano sovrapposte. La funzione restituisce ***dst***.

```
#include <string.h>
char *
strcat (char *restrict dst, const char *restrict org)
{
  size_t i;
  size_t j;
  for (i = 0; dst[i] != 0; i++)
    {
      ; // Si limita a cercare il carattere nullo.
    }
  for (j = 0; org[j] != 0; i++, j++)
    {
      dst[i] = org[j];
    }
  dst[i] = 0;
  return dst;
}
```


Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove la stringa ***y*** viene estesa con il contenuto di ***x***.

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char x[] = "abcdefghijklmnopqrstuvwxyz";
  char y[50] = "ciao";
  printf ("prima: %s\n", y);
  strcat (y, x);
  printf ("dopo: %s\n", y);
  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
prima: ciao
dopo: ciaoabcdefghijklmnopqrstuvwxyz
```

69.14.2.2 Funzione «strncat()»

La funzione ***strncat()*** si comporta in modo analogo a ***strcat()***, con la differenza che copia al massimo ***n*** caratteri, ammesso che la stringa ***org*** ne contenga abbastanza. In ogni caso, la stringa ***dst*** viene completata con il carattere nullo di terminazione.

```
#include <string.h>
char *
strncat (char *restrict dst, const char *restrict org,
         size_t n)
{
  size_t i;
  size_t j;
  for (i = 0; n > 0 && dst[i] != 0; i++)
    {
      ; // Si limita a cercare il carattere nullo.
    }
  for (j = 0; n > 0 && j < n && org[j] != 0; i++, j++)
    {
      dst[i] = org[j];
    }
  dst[i] = 0;
  return dst;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove la stringa ***y*** viene estesa con il contenuto di ***x***, in due fasi.

```
#include <stdio.h>
#include <string.h>
int main (void)
{
  char x[] = "abcdefghijklmnopqrstuvwxyz";
  char y[50] = "ciao";
  printf ("prima: %s\n", y);
  strncat (y, x, 10);
  printf ("durante: %s\n", y);
  strncat (y, x, 40);
  printf ("dopo: %s\n", y);
  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
prima: ciao
durante: ciaoabcdefghij
dopo: ciaoabcdefghijabcdefghijklmnopqrstuvwxyz
```

69.14.3 Comparazione

Le funzioni di comparazione ***memcmp()***, ***strcmp()*** e ***strncmp()*** confrontano due sequenze di caratteri, determinando se la prima sia maggiore, minore o uguale rispetto alla seconda, scandendo i caratteri progressivamente e arrestando l'analisi appena si incontra una differenza. Pertanto, il carattere che differisce è quello che determina l'ordine tra le due sequenze.

```
int    memcmp  (const void *s1, const void *s2, size_t n);
int    strcmp  (const char *s1, const char *s2);
int    strcoll (const char *s1, const char *s2);
int    strncmp (const char *s1, const char *s2, size_t n);
size_t strxfrm (char *restrict dst,
                const char *restrict org, size_t n);
```

69.14.3.1 Funzione «memcmp()» (memory compare)

La funzione ***memcmp()*** confronta i primi ***n*** caratteri delle aree di memoria a cui puntano ***s1*** e ***s2***, restituendo: un valore pari a zero se le due sequenze si equivalgono; un valore maggiore di zero se la sequenza di ***s1*** è maggiore di ***s2***; un valore minore di zero se la sequenza di ***s1*** è minore di ***s2***.

```
#include <string.h>
int
memcmp (const void *s1, const void *s2, size_t n)
{
  unsigned char *a = (unsigned char *) s1;
  unsigned char *b = (unsigned char *) s2;
  size_t i;
  for (i = 0; n > 0 && i < n; i++)
    {
      if (a[i] > b[i])
        {
          return 1;
        }
      else if (a[i] < b[i])
        {
          return -1;
        }
    }
  return 0;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove le variabili ***x*** e ***y*** sono interi (che si presume siano a 32 bit) rappresentati in memoria invertendo l'ordine dei byte (*little endian*), pertanto il confronto avviene in modo inverso all'apparenza dei simboli.

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  unsigned int  x = 0x123456FF;
  unsigned int  y = 0xEEEEEEEE;
  int  r;
  r = memcmp (&x, &y, sizeof (int));
  printf ("memcmp: %x %i %x\n", x, r, y);
  r = memcmp (&x, &x, sizeof (int));
  printf ("memcmp: %x %i %x\n", x, r, x);
  r = memcmp (&y, &x, sizeof (int));
  printf ("memcmp: %x %i %x\n", y, r, x);
  return 0;
}
```

Avviando questo programma nelle condizioni descritte, si deve ottenere un risultato come quello seguente:

```
memcmp: 123456ff 1 eeeeeeee
memcmp: 123456ff 0 123456ff
memcmp: eeeeeeee -1 123456ff
```


69.14.3.2 Funzione «strcmp()» (string compare)

La funzione ***strcmp()*** confronta due stringhe restituendo: un valore pari a zero se sono uguali; un valore maggiore di zero se la stringa ***s1*** è maggiore di ***s2***; un valore minore di zero se la stringa ***s1*** è minore di ***s2***.

```
#include <string.h>
int
strcmp (const char *s1, const char *s2)
{
  unsigned char *a = (unsigned char *) s1;
  unsigned char *b = (unsigned char *) s2;
  size_t i;
  for (i = 0; ; i++)
    {
      if (a[i] > b[i])
        {
          return 1;
        }
      else if (a[i] < b[i])
        {
          return -1;
        }
      else if (a[i] == 0 && b[i] == 0)
        {
          return 0;
        }
    }
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char x[] = "ciao";
  char y[] = "ciao amore";
  int  r;
  r = strcmp (x, y);
  printf ("strcmp: %s %i %s\n", x, r, y);
  r = strcmp (x, x);
  printf ("strcmp: %s %i %s\n", x, r, x);
  r = strcmp (y, x);
  printf ("strcmp: %s %i %s\n", y, r, x);
  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
strcmp: ciao -1 ciao amore
strcmp: ciao 0 ciao
strcmp: ciao amore 1 ciao
```

69.14.3.3 Funzione «strcoll()» (string collate compare)

La funzione ***strcoll()*** è analoga a ***strcmp()***, con la differenza che la comparazione avviene sulla base della configurazione locale (la categoria '**`LC_COLLATE`**'). Nel caso della configurazione locale '**`C`**' la funzione si comporta esattamente come ***strcmp()***.

69.14.3.4 Funzione «strncmp()»

La funzione ***strncmp()*** si comporta in modo analogo a ***strcmp()***, con la differenza che la comparazione si arresta al massimo dopo ***n*** caratteri.

```
#include <string.h>
int
strncmp (const char *s1, const char *s2, size_t n)
{
  unsigned char *a = (unsigned char *) s1;
  unsigned char *b = (unsigned char *) s2;
  size_t i;
  for (i = 0; i < n ; i++)
    {
      if (a[i] > b[i])
```

```
        {
          return 1;
        }
      else if (a[i] < b[i])
        {
          return -1;
        }
      else if (a[i] == 0 && b[i] == 0)
        {
          return 0;
        }
    }
  return 0;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char x[] = "CIao";
  char y[] = "CIAO";
  int  r;
  r = strncmp (x, y, 4);
  printf ("strcmp: %i %s %i %s\n", 4, x, r, y);
  r = strncmp (x, y, 2);
  printf ("strcmp: %i %s %i %s\n", 2, x, r, x);
  r = strncmp (y, x, 4);
  printf ("strcmp: %i %s %i %s\n", 4, y, r, x);
  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
strncmp: 4 CIao 1 CIAO
strncmp: 2 CIao 0 CIao
strncmp: 4 CIAO -1 CIao
```

69.14.3.5 Funzione «strxfrm()» (string transform)

La funzione ***strxfrm()*** trasforma la stringa ***org*** sovrascrivendo la stringa ***dst*** in modo relativo alla configurazione locale. In pratica, la stringa trasformata che si ottiene può essere comparata con un'altra stringa trasformata nello stesso modo attraverso la funzione ***strcmp()*** ottenendo lo stesso esito che si avrebbe confrontando le stringhe originali con la funzione ***strcoll()***.

Nella stringa di destinazione vengono messi non più di ***n*** caratteri, incluso il carattere nullo di terminazione. Se ***n*** è pari a zero, ***dst*** può essere un puntatore nullo. Le due stringhe non devono sovrapporsi.

La funzione ***strxfrm()*** restituisce la quantità di caratteri necessari a contenere la stringa ***org*** trasformata, senza però contare il carattere nullo di terminazione. Se ***n*** è zero e ***dst*** corrisponde al puntatore nullo, restituisce il valore che sarebbe necessario per trasformare la stringa ***org*** in tutta la sua lunghezza.

L'esempio seguente di tale funzione è valido solo per la configurazione locale '**C**':

```
#include <string.h>
size_t
strxfrm (char *restrict dst, const char *restrict org,
         size_t n)
{
  size_t i;
  if (n == 0 && dst == NULL)
    {
      return strlen (org);
    }
  else
    {
      for (i = 0; i < n ; i++)
        {
          dst[i] = org[i];
          if (org[i] == 0)
```



```
            {
              break;
            }
        }
      return i;
    }
}
```

69.14.4 Ricerca

Seguono i prototipi delle funzioni utili per la ricerca all'interno di sequenze di byte, secondo lo standard C:

```
void  *memchr   (const void *s, int c, size_t n);
char  *strchr   (const char *s, int c);
char  *strrchr  (const char *s, int c);
size_t strspn   (const char *s, const char *accept);
size_t strcspn  (const char *s, const char *reject);
char  *strpbrk  (const char *s, const char *accept);
char  *strstr   (const char *string, const char *substring);
char  *strtok   (char *restrict string,
                 const char *restrict delim);
```

Lo standard POSIX aggiunge anche il prototipo seguente:

```
char  *strtok_r (char *restrict string,
                 const char *restrict delim,
                 char **saveptr);
```

69.14.4.1 Funzione «memchr()» (memory character)

La funzione ***memchr()*** cerca un carattere a partire da una certa posizione in memoria, scandendo al massimo una quantità determinata di caratteri, restituendo il puntatore al carattere trovato. Se nell'ambito specificato non trova il carattere, restituisce il puntatore nullo.

```
#include <string.h>
void *
memchr (const void *s, int c, size_t n)
{
  unsigned char *a = (unsigned char *) s;
  unsigned char x  = (unsigned char) c;
  size_t i;
  for (i = 0; n > 0 && i < n; i++)
    {
      if (a[i] == x)
        {
          return (void *) (s + i);
        }
    }
  return NULL;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, in cui si scandisce il contenuto di una variabile di tipo '**int**', intendendo che questa debba occupare uno spazio di 32 bit:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  int x = 0x13579BDF;
  void *p;
  p = memchr (&x, 0xDF, 4);
  printf ("contenuto della variabile: %x\n", x);
  printf ("indirizzo iniziale della variabile: 0x%x\n",
          (unsigned int) &x);
  printf ("indirizzo di 0x%x all'interno della "
          "variabile: 0x%x\n",
          0xDF,
          (unsigned int) p);
  return 0;
}
```

Avviando questo programma in un'architettura che inverte l'ordine dei byte (*little endian*) si deve ottenere un risultato simile a quello seguente:

```
contenuto della variabile: 13579bdf
```

```
indirizzo iniziale della variabile: 0xbff7f3bc
indirizzo di 0xdf all'interno della variabile: 0xbff7f3bc
```

69.14.4.2 Funzione «strchr()» (string character)

La funzione ***strchr()*** cerca un carattere all'interno di una stringa, restituendo il puntatore al carattere trovato, oppure il puntatore nullo se la ricerca fallisce. Nella scansione viene preso in considerazione anche il carattere nullo di terminazione della stringa.

```
#include <string.h>
char *
strchr (const char *s, int c)
{
  unsigned char *a = (unsigned char *) s;
  unsigned char x  = (unsigned char) c;
  size_t i;
  for (i = 0; ; i++)
    {
      if (a[i] == x)
        {
          return (char *) (s + i);
        }
      else if (a[i] == 0)
        {
          return NULL;
        }
    }
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char *x = "ciao amore mio";
  char *p;

  p = strchr (x, 'a');
  printf ("La stringa \"%s\", collocata a partire ", x);
  printf ("dall'indirizzo %u, contiene il carattere '%c' ",
          (unsigned int) x,  'a');
  printf ("all'indirizzo %u.\n", (unsigned int) p);

  p = strchr (x, 0);
  printf ("La stringa \"%s\", collocata a partire ", x);
  printf ("dall'indirizzo %u, contiene il carattere 0x%x ",
          (unsigned int) x, 0);
  printf ("all'indirizzo %u.\n", (unsigned int) p);

  return 0;
}
```

Avviando questo programma si deve ottenere un risultato simile a quello seguente:

```
La stringa "ciao amore mio", collocata a partire ↩
↪dall'indirizzo 134516936, contiene il carattere ↩
↪'a' all'indirizzo 134516938.
La stringa "ciao amore mio", collocata a partire ↩
↪dall'indirizzo 134516936, contiene il carattere ↩
↪0x0 all'indirizzo 134516950.
```

69.14.4.3 Funzione «strrchr()» (string character)

La funzione ***strrchr()*** cerca un carattere all'interno di una stringa, restituendo il puntatore all'ultimo carattere corrispondente trovato, oppure il puntatore nullo se la ricerca fallisce. Nella scansione viene preso in considerazione anche il carattere nullo di terminazione della stringa.

```
#include <string.h>
char *
strrchr (const char *string, int c)
{
  int i;
  //
  for (i = strlen (string); i >= 0; i--)
    {
      if (string[i] == (char) c)
        {
          break;
        }
    }
  //
  if (i < 0)
    {
      return NULL;
    }
  else
    {
      return (string + i);
    }
}
```


Per verificare sommariamente il comportamento della funzione si può modificare leggermente l'esempio già apparso a proposito della funzione ***strchr()***:

```
…
  p = strrchr (x, 'a');
…
  p = strrchr (x, 0);
…
```

Avviando questo programma si deve ottenere un risultato simile a quello seguente:

```
La stringa "ciao amore mio", collocata a partire ↩
↪dall'indirizzo 134514088, contiene il carattere ↩
↪'a' all'indirizzo 134514093.
La stringa "ciao amore mio", collocata a partire ↩
↪dall'indirizzo 134514088, contiene il carattere ↩
↪0x0 all'indirizzo 134514102.
```

69.14.4.4 Funzione «strspn()» (string span)

La funzione ***strspn()*** calcola la lunghezza massima iniziale della stringa ***s***, composta esclusivamente da caratteri contenuti nella stringa ***accept***, restituendo tale valore.

```
#include <string.h>
size_t
strspn (const char *s, const char *accept)
{
  size_t i;
  size_t j;
  int found;
  for (i = 0; s[i] != 0; i++)
    {
      for (j = 0, found = 0; accept[j] != 0; j++)
        {
          if (s[i] == accept[j])
            {
              found = 1;
              break;
            }
        }
      if (!found)
        {
          break;
        }
    }
  return i;
}
```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char *x = "ciao amore mio";
  char *y = "abcdefghi";
  size_t n;

  n = strspn (x, y);
  printf ("La parte iniziale di \"%s\" ", x);
  printf ("che contiene i caratteri \"%s\" ", y);
```

```
  printf ("si compone di %i caratteri.\n", n);

  n = strspn (x, x);
  printf ("La parte iniziale di \"%s\" ", x);
  printf ("che contiene i caratteri \"%s\" ", x);
  printf ("si compone di %i caratteri.\n", n);

  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
La parte iniziale di "ciao amore mio" che contiene i ↩
↪caratteri "abcdefghi" si compone di 3 caratteri.
La parte iniziale di "ciao amore mio" che contiene i ↩
↪caratteri "ciao amore mio" si compone di 14 caratteri.
```

69.14.4.5 Funzione «strcspn()»

La funzione ***strcspn()*** si comporta in modo analogo a ***strspn()***, con la differenza che l'insieme di caratteri contenuto nella stringa '**reject**' non deve costituire l'insieme iniziale della stringa ***s*** che si va a contare. Pertanto, il valore restituito è la quantità di caratteri iniziali della stringa ***s*** che non si trovano anche nell'insieme ***reject***.

```
#include <string.h>
size_t
strcspn (const char *s, const char *reject)
{
  size_t i;
  size_t j;
  int found;
  for (i = 0; s[i] != 0; i++)
    {
      for (j = 0, found = 0; reject[j] != 0 || found; j++)
        {
          if (s[i] == reject[j])
            {
              found = 1;
              break;
            }
        }
      if (found)
        {
          break;
        }
    }
  return i;
}
```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char *x = "ciao amore mio";
  char *y = "mnopqrstuvwxyz";
  size_t n;

  n = strcspn (x, y);
  printf ("La parte iniziale di \"%s\" ", x);
  printf ("che non contiene i caratteri \"%s\" ", y);
  printf ("si compone di %i caratteri.\n", n);

  n = strcspn (x, x);
  printf ("La parte iniziale di \"%s\" ", x);
  printf ("che non contiene i caratteri \"%s\" ", x);
  printf ("si compone di %i caratteri.\n", n);

  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
La parte iniziale di "ciao amore mio" che non contiene i ↩
↪caratteri "mnopqrstuvwxyz" si compone di 3 caratteri.
```



```
La parte iniziale di "ciao amore mio" che non contiene i ↩
↪caratteri "ciao amore mio" si compone di 0 caratteri.
```

69.14.4.6 Funzione «strpbrk()» (string point break)

La funzione ***strpbrk()*** scandisce la stringa ***s*** alla ricerca del primo carattere che risulti contenuto nella stringa ***accept***, restituendo il puntatore al carattere trovato, oppure, in mancanza di alcuna corrispondenza, il puntatore nullo.

```
#include <string.h>
char *
strpbrk (const char *s, const char *accept)
{
  size_t i;
  size_t j;
  for (i = 0; s[i] != 0; i++)
    {
      for (j = 0; accept[j] != 0; j++)
        {
          if (s[i] == accept[j])
            {
              return (char *) (s + i);
            }
        }
    }
  return NULL;
}
```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char *x = "ciao amore mio";
  char *y = "mnopqrstuvwxyz";
  char *p;

  p = strpbrk (x, y);
  printf ("La stringa \"%s\" che inizia all'indirizzo %u, ",
          x, (unsigned int) x);
  printf ("trova la prima corrispondenza con la "
          "stringa \"%s\" ", y);
  printf ("all'indirizzo %u.\n", (unsigned int) p);

  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
La stringa "ciao amore mio" che inizia all'indirizzo ↩
↪134516840, trova la prima corrispondenza con la ↩
↪stringa "mnopqrstuvwxyz" all'indirizzo 134516843.
```

69.14.4.7 Funzione «strstr()»

La funzione ***strstr()*** cerca la stringa ***substring*** nella stringa ***string*** restituendo il puntatore alla prima corrispondenza trovata (nella stringa ***string***). Se la corrispondenza non c'è, la funzione restituisce il puntatore nullo.

```
#include <string.h>
char *
strstr (const char *string, const char *substring)
{
  size_t i;
  size_t j;
  size_t k;
  int found;
  if (substring[0] == 0)
    {
      return (char *) string;
    }
  for (i = 0, j = 0, found = 0; string[i] != 0; i++)
    {
      if (string[i] == substring[0])
        {
          for (k = i, j = 0;
```

```
               string[k] == substring[j] && string[k] != 0
                && substring[j] != 0;
               j++, k++)
            {
              ;
            }
          if (substring[j] == 0)
            {
              found = 1;
            }
        }
      if (found)
        {
          return (char *) (string + i);
        }
    }
  return NULL;
}
```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char *x = "ciao amore mio";
  char *y = "amore";
  char *p;

  p = strstr (x, y);
  printf ("La stringa \"%s\" che inizia all'indirizzo %u, ",
          x, (unsigned int) x);
  printf ("contiene la stringa \"%s\" ", y);
  printf ("all'indirizzo %u.\n", (unsigned int) p);

  p = strstr (x, "");
  printf ("La stringa \"%s\" che inizia all'indirizzo %u, ",
          x, (unsigned int) x);
  printf ("contiene la stringa \"%s\" ", "");
  printf ("all'indirizzo %u.\n", (unsigned int) p);

  p = strstr (x, "baba");
  printf ("La stringa \"%s\" che inizia all'indirizzo %u, ",
          x, (unsigned int) x);
  printf ("contiene la stringa \"%s\" ", "baba");
  printf ("all'indirizzo %u.\n", (unsigned int) p);

  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
La stringa "ciao amore mio" che inizia all'indirizzo ↩
↪134517000, contiene la stringa "amore" all'indirizzo ↩
↪134517005.
La stringa "ciao amore mio" che inizia all'indirizzo ↩
↪134517000, contiene la stringa "" all'indirizzo 134517000.
La stringa "ciao amore mio" che inizia all'indirizzo ↩
↪134517000, contiene la stringa "baba" all'indirizzo 0.
```

69.14.4.8 Funzione «strtok()» (string token)

La funzione ***strtok()*** serve a suddividere una stringa in unità, definite *token*, specificando un elenco di caratteri da intendere come delimitatori, in una seconda stringa. La funzione va usata in fasi successive, fornendo solo inizialmente la stringa da suddividere che continua poi a essere utilizzata se al suo posto viene fornito il puntatore nullo. La funzione restituisce, di volta in volta, il puntatore alla sottostringa contenente l'unità individuata, oppure il puntatore nullo, se non può trovarla.

La funzione deve tenere memoria di un puntatore in un'area di memoria persistente (quello che nei commenti viene definito «puntatore statico») e deve isolare le unità modificando la stringa originale, inserendo il carattere nullo di terminazione alla fine delle unità individuate.

Quando la funzione viene chiamata indicando al posto della stringa da scandire il puntatore nullo, l'insieme dei delimitatori può essere diverso da quello usato nelle fasi precedenti.

```
#include <string.h>
char *
strtok (char *restrict string, const char *restrict delim)
{
  static char *next = NULL;
  size_t i = 0;
  size_t j;
  int found_token;
  int found_delim;
  //
  // Se la stringa fornita come argomento è un puntatore
  // nullo, occorre avvalersi del puntatore statico. Se
  // però questo è nullo a sua volta, la scansione non può
  // avvenire.
  //
  if (string == NULL)
    {
      if (next == NULL)
        {
          return NULL;
        }
      else
        {
          string = next;
        }
    }
  //
  // Se la stringa fornita come argomento è vuota, la
  // scansione non può avvenire.
  //
  if (string[0] == 0)
    {
      next = NULL;
      return NULL;
    }
  else
    {
      if (delim[0] == 0)
        {
          return string;
        }
    }
  //
  // Trova la prossima unità (token).
  //
  for (i = 0, found_token = 0, j = 0;
       string[i] != 0 && (!found_token);
       i++)
    {
      //
      // Cerca tra i delimitatori.
      //
      for (j = 0, found_delim = 0; delim[j] != 0; j++)
        {
          if (string[i] == delim[j])
            {
              found_delim = 1;
            }
        }
      //
      // Se il carattere attuale della stringa non è
      // un delimitatore, si tratta dell'inizio di una
      // nuova unità (token).
      //
      if (!found_delim)
        {
          found_token = 1;
          break;
        }
    }
  //
  // Se è stata trovata una unità (token) viene aggiustato
  // il puntatore che rappresenta la stringa. Se invece
  // non è stata trovata l'unità, vuol dire che non ce ne
  // possono essere altre.
  //
  if (found_token)
    {
```

```
      string += i;
    }
  else
    {
      next = NULL;
      return NULL;
    }
  //
  // Cerca la fine dell'unità trovata.
  //
  for (i = 0, found_delim = 0; string[i] != 0; i++)
    {
      for (j = 0; delim[j] != 0; j++)
        {
          if (string[i] == delim[j])
            {
              found_delim = 1;
              break;
            }
        }
      if (found_delim)
        {
          break;
        }
    }
  //
  // Se è stato trovato un delimitatore, allora il carattere
  // corrispondente nella stringa deve essere azzerato.
  // Se invece la stringa originale è terminata per conto
  // proprio, allora non è possibile continuare la ricerca
  // in una fase successiva, perché non ci possono essere
  // altre unità.
  //
  if (found_delim)
    {
      string[i] = 0;
      next = &string[i+1];
    }
  else
    {
      next = NULL;
    }
  //
  // A questo punto, la stringa attuale rappresenta
  // l'unità trovata.
  //
  return string;
}
```

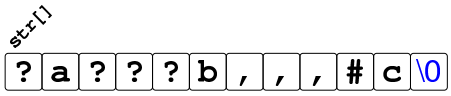
Per comprendere lo scopo della funzione viene utilizzato lo stesso esempio che appare nel documento *ISO/IEC 9899:TC2*, al paragrafo 7.21.5.7, con qualche piccola modifica per poterlo rendere un programma autonomo:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char str[] = "?a???b,,,#c";
  char *t;

  t = strtok (str, "?");          // t punta all'unità "a"
  printf ("strtok: \"%s\"\n", t);
  t = strtok (NULL, ",");         // t punta all'unità "??b"
  printf ("strtok: \"%s\"\n", t);
  t = strtok (NULL, "#,");        // t punta all'unità "c"
  printf ("strtok: \"%s\"\n", t);
  t = strtok (NULL, "?");         // t è un puntatore nullo
  printf ("strtok: \"%s\"\n", t);

  return 0;
}
```

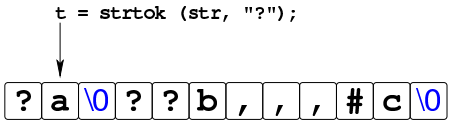
Avviando il programma si ottiene quanto già descritto dai commenti inseriti nel codice:

```
strtok: "a"
strtok: "??b"
strtok: "c"
strtok: "(null)"
```

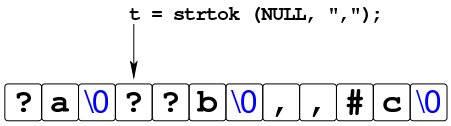

Ciò che avviene nell'esempio può essere schematizzato dalle figure successive. Inizialmente la stringa '**str**' ha in memoria l'aspetto seguente:



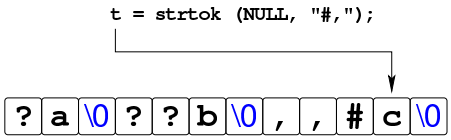
Dopo la prima chiamata della funzione ***strtok()*** la stringa risulta alterata e il puntatore ottenuto raggiunge la lettera '**a**':



Dopo la seconda chiamata della funzione, in cui si usa il puntatore nullo per richiedere una scansione ulteriore della stringa originale, si ottiene un nuovo puntatore che, questa volta, inizia a partire dal quarto carattere, rispetto alla stringa originale, dal momento che il terzo è già stato sovrascritto da un carattere nullo:



La penultima chiamata della funzione ***strtok()*** raggiunge la lettera '**c**' che è anche alla fine della stringa originale:



L'ultimo tentativo di chiamata della funzione non può dare alcun esito, perché la stringa originale si è già conclusa.

Va tenuto in considerazione che la funzione ***strtok()***, dovendo mantenere in memoria la posizione trovata dell'ultima scansione eseguita, da una chiamata a quella successiva, non è «rientrante», pertanto non si presta per i programmi che si suddividono in più thread.

69.14.4.9 Funzione «strtok_r()»

La funzione ***strtok_r()***, richiesta dallo standard POSIX, salva il puntatore interno alla stringa che viene scandita, esternamente, in modo da poter essere usata in un contesto in cui più thread operano simultaneamente. In pratica si aggiunge un terzo parametro, costituito da un puntatore a puntatore a carattere.

Il puntatore a carattere ('**char ***'), il cui puntatore viene fornito come terzo argomento della funzione, deve essere dichiarato prima della chiamata della funzione. La prima volta che viene chiamata la funzione ***strtok_r()*** non conta quale valore abbia effettivamente tale variabile di tipo '**char ***', perché è la funzione stessa che lo inizializza, ma nelle chiamate successive (quando al posto della stringa si dà alla funzione il valore '**NULL**'). Quando termina la scansione della stringa con le chiamate di ***strtok_r()***, la variabile '**char ***' di cui si passa il puntatore, può essere utilizzata per altri scopi, o per altre scansioni.

La soluzione seguente, per la realizzazione della funzione ***strtok_r()***, può essere confrontato con quella relativa alla funzione ***strtok()***, per comprendere il senso e l'utilizzo dell'ultimo parametro.

```
char *
strtok_r (char *restrict string, const char *restrict delim,
```

```
          char **saveptr)
{
  size_t i = 0;
  size_t j;
  int found_token;
  int found_delim;
  //
  // Se la stringa fornita come argomento è un puntatore
  // nullo, occorre avvalersi del puntatore "*saveptr".
  // Se però questo è nullo a sua volta, la scansione non
  // può avvenire.
  //
  if (string == NULL)
    {
      if (*saveptr == NULL)
        {
          return NULL;
        }
      else
        {
          string = *saveptr;
        }
    }
  //
  // Se la stringa fornita come argomento è vuota, la
  // scansione non può avvenire.
  //
  if (string[0] == 0)
    {
      *saveptr = NULL;
      return NULL;
    }
  else
    {
      if (delim[0] == 0)
        {
          return string;
        }
    }
  //
  // Trova la prossima unità (token).
  //
  for (i = 0, found_token = 0, j = 0;
       string[i] != 0 && (!found_token);
       i++)
    {
      //
      // Cerca tra i delimitatori.
      //
      for (j = 0, found_delim = 0; delim[j] != 0; j++)
        {
          if (string[i] == delim[j])
            {
              found_delim = 1;
            }
        }
      //
      // Se il carattere attuale della stringa non è
      // un delimitatore, si tratta dell'inizio di una
      // nuova unità (token).
      //
      if (!found_delim)
        {
          found_token = 1;
          break;
        }
    }
  //
  // Se è stata trovata una unità (token) viene aggiustato
  // il puntatore che rappresenta la stringa. Se invece
  // non è stata trovata l'unità, vuol dire che non ce ne
  // possono essere altre.
  //
  if (found_token)
    {
      string += i;
    }
  else
    {
      *saveptr = NULL;
      return NULL;
    }
  //
  // Cerca la fine dell'unità trovata.
  //
  for (i = 0, found_delim = 0; string[i] != 0; i++)
    {
      for (j = 0; delim[j] != 0; j++)
        {
          if (string[i] == delim[j])
            {
              found_delim = 1;
              break;
            }
        }
      if (found_delim)
        {
          break;
        }
    }
  //
  // Se è stato trovato un delimitatore, allora il carattere
  // corrispondente nella stringa deve essere azzerato.
  // Se invece la stringa originale è terminata per conto
  // proprio, allora non è possibile continuare la ricerca
  // in una fase successiva, perché non ci possono essere
  // altre unità.
  //
  if (found_delim)
    {
      string[i] = 0;
      *saveptr = &string[i+1];
    }
  else
    {
      *saveptr = NULL;
    }
  //
  // A questo punto, la stringa attuale rappresenta
  // l'unità trovata.
  //
  return string;
}
```


Per dimostrare il lavoro della funzione, viene utilizzato lo stesso esempio già usato a proposito di ***strtok()***, con poche piccole modifiche:

```
int
main (void)
{
  char str[] = "?a???b,,,#c";
  char *t;
  char *save;

  t = strtok_r (str, "?", &save);   // t punta all'unità "a"
  printf ("strtok: \"%s\"\n", t);
  t = strtok_r (NULL, ",", &save);  // t punta all'unità "??b"
  printf ("strtok: \"%s\"\n", t);
  t = strtok_r (NULL, "#,", &save); // t punta all'unità "c"
  printf ("strtok: \"%s\"\n", t);
  t = strtok_r (NULL, "?", &save);  // t è un puntatore nullo
  printf ("strtok: \"%s\"\n", t);

  return 0;
}
```

Avviando il programma si ottiene esattamente la stessa cosa dell'esempio già visto:

```
strtok: "a"
strtok: "??b"
strtok: "c"
strtok: "(null)"
```

69.14.5 Funzioni varie

Seguono i prototipi delle funzioni descritte nelle sezioni successive. Questi appartengono allo standard C:

```
void   *memset     (void *s, int c, size_t n);
char   *strerror   (int errnum);
size_t strlen      (const char *s);
```

Il prototipo successivo viene aggiunto dallo standard POSIX:

```
int    strerror_r (int errnum, char *s, size_t n);
```

69.14.5.1 Funzione «memset()» (memory set)

La funzione ***memset()*** consente di inizializzare una certa area di memoria con la ripetizione di un certo carattere. Per la precisione, viene usato il valore del parametro ***c***, tradotto in un carattere senza segno, copiandolo per ***n*** volte a partire dall'indirizzo a cui punta ***s***. La funzione restituisce ***s***.

```
#include <string.h>
void *
memset (void *s, int c, size_t n)
{
  unsigned char *a = (unsigned char *) s;
  unsigned char x  = (unsigned char) c;
  size_t i;
  for (i = 0; n > 0 && i < n; i++)
    {
      a[i] = x;
    }
  return s;
}
```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  int x = 0x12345678;
  printf ("prima: 0x%x\n", x);
  memset (&x, 0xFF, 2);
  printf ("dopo: 0x%x\n", x);

  char X[] = "ciao amore mio";
  printf ("prima: \"%s\"\n", X);
  memset (X, 'Q', 5);
  printf ("dopo: \"%s\"\n", X);

  return 0;
}
```

Avviando questo programma in un elaboratore con architettura a 32 bit e inversione dei byte (*little endian*) si deve ottenere il risultato seguente:

```
prima: 0x12345678
dopo: 0x1234ffff
prima: "ciao amore mio"
dopo: "QQQQQamore mio"
```

69.14.5.2 Funzione «strerror()» (string error)

La funzione ***strerror()*** serve a tradurre il numero fornito come argomento in un puntatore da cui inizia una stringa contenente una spiegazione. In altri termini, serve a trasformare un numero in una descrizione di un tipo di errore. Qui viene mostrata una soluzione priva di utilità, anche se risponde alle richieste delle specifiche:

```
#include <string.h>
char *
strerror (int errnum)
{
  static char answare[] = "Unknown error";
  return answare;
}
```

L'esempio successivo può servire a dimostrare il senso di questa funzione:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  printf ("%s\n", strerror (0));
  printf ("%s\n", strerror (1));
  printf ("%s\n", strerror (2));
  printf ("%s\n", strerror (3));
  printf ("%s\n", strerror (4));
  return 0;
}
```

Utilizzando questo programma compilato con le librerie di un sistema GNU si potrebbero vedere i messaggi seguenti:

```
Success
Operation not permitted
No such file or directory
No such process
Interrupted system call
```

La stringa a cui punta la funzione può essere condivisa da altre chiamate successive della stessa, pertanto, in un programma con thread multipli, è possibile che avvenga la sovrascrittura, a meno di disporre di un elenco separato di tutti i tipi di messaggio di errore.

69.14.5.3 Funzione «strerror_r()» (string error)

La funzione ***strerror_r()*** viene aggiunta dallo standard POSIX e consente di tradurre un errore numerico in stringa, fornendo alla funzione il puntatore iniziale della stringa da produrre e la lunghezza massima che questa può raggiungere, garantendo l'indipendenza tra thread multipli.

Per quanto riguarda l'utilizzo, la differenza fondamentale rispetto a ***strerror()*** sta nel fatto che restituisce un valore intero, pari a zero, se l'operazione ha avuto successo, oppure `-1` in caso di problemi, aggiornando di conseguenza anche la variabile ***errno***. L'esempio successivo può servire a dimostrare il senso di questa funzione:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char msg[100];

  if (!strerror_r (1, msg, 100))
    {
      printf ("%s\n", msg);
    }
  if (!strerror_r (2, msg, 100))
    {
      printf ("%s\n", msg);
    }
  if (!strerror_r (3, msg, 100))
    {
      printf ("%s\n", msg);
    }
  if (!strerror_r (4, msg, 100))
    {
      printf ("%s\n", msg);
    }
  return 0;
}
```

Utilizzando questo programma compilato con le librerie di un sistema GNU si potrebbero vedere i messaggi seguenti:

```
Success
Operation not permitted
No such file or directory
No such process
Interrupted system call
```

69.14.5.4 Funzione «strlen()» (string length)

La funzione ***strlen()*** calcola la lunghezza di una stringa, escludendo dal conteggio il carattere nullo di terminazione:

```
#include <string.h>
size_t
strlen (const char *s)
{
  size_t i;
```

```
  for (i = 0; s[i] != 0 ; i++)
    {
      ; // Esegue solo il conteggio.
    }
  return i;
}
```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  size_t lunghezza;
  char stringa[] = "ciao amore";
  lunghezza = strlen (stringa);
  printf ("la frase \"%s\" si compone di %i caratteri\n",
          stringa,
          lunghezza);
  return 0;
}
```

Avviando il programma si deve vedere il risultato seguente:

```
la frase "ciao amore" si compone di 10 caratteri
```

69.15 File «signal.h»

Il file '`signal.h`' della libreria standard definisce principalmente delle funzioni per la gestione dei segnali che riguardano il programma. Assieme alle funzioni definisce anche delle macro-variabili per classificare i segnali e per fare riferimento a delle funzioni predefinite, destinate astrattamente al trattamento dei segnali (si veda eventualmente la realizzazione del file '`signal.h`' e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.17).

Dal punto di vista del programmatore, l'uso delle funzioni di questo file di intestazione può essere abbastanza semplice, ma la comprensione di come siano organizzate nel file '`signal.h`' diventa invece difficile.

Nello standard POSIX, la questione dei segnali è particolarmente complessa. Nel capitolo viene considerato solo il fatto che i segnali standard sono in numero maggiore, tralasciando sostanzialmente il resto.

Qui vengono proposti due modi alternativi di scrivere il file '`signal.h`' che dovrebbero essere disponibili presso *allegati/c/include/signal.h* e *allegati/c/include/signal-bis.h* .

69.15.1 Dichiarazione contorta

Per la gestione dei segnali ci sono due funzioni che vengono dichiarate nel file '`signal.h`': ***signal()*** e ***raise()***. La funzione ***raise()*** serve ad azionare un segnale, come dire che serve ad attivare manualmente un allarme interno al programma, specificato da un numero particolare che ne definisce il tipo. Il programma contiene sempre una procedura predefinita che stabilisce ciò che deve essere fatto in presenza di un certo allarme, ma il programmatore può ridefinire la procedura attraverso l'uso della funzione ***signal()***, con la quale si associa l'avvio di una funzione particolare in presenza di un certo segnale. Il modello sintattico seguente rappresenta, in modo estremamente semplificato, l'uso della funzione ***signal()***:

```
signal (n_segnale, funzione_da_associare)
```

Logicamente la funzione che si associa a un certo numero di segnale viene indicata negli argomenti della chiamata come puntatore a funzione. La funzione che viene passata come argomento è un gestore di segnale e deve avere una certa forma:

```
void gestore (n_segnale)
```


In pratica, quando viene creata l'associazione tra segnale e funzione che deve gestirlo, la funzione in questione deve avere un parametro tale da poter rappresentare il numero del segnale che la riguarda e non restituisce alcun valore (pertanto è di tipo '**`void`**').

Avendo determinato questo, il modello della funzione ***signal()*** può essere precisato un po' di più:

```
signal (n_segnale, void (*gestore)(int))
```

Ciò significa che il secondo argomento della funzione ***signal()*** è un puntatore a una funzione ('***gestore*`()`**') con un parametro di tipo '**`int`**', la quale non restituisce alcunché ('**`void`**').

Ma non è ancora stato specificato cosa deve restituire la funzione ***signal()***: un puntatore a una funzione che ha un parametro di tipo '**`int`**' e che a sua volta non restituisce alcunché. In pratica, ***signal()*** deve restituire il puntatore a una funzione che ha le stesse caratteristiche di quella del proprio secondo parametro. A questo punto, si arriva al prototipo completo, ma molto difficile da interpretare a prima vista:

```
void (*signal (n_segnale, void (*gestore)(int)))(int);
```

Per ovviare a questo problema di comprensibilità, anche se lo standard non lo prescrive, di norma, nel file '`signal.h`' si dichiara un tipo speciale, in qualità di puntatore a funzione con le caratteristiche del gestore di segnale:

```
…
typedef void (*sighandler_t) (int);
…
```

Così facendo, la funzione ***signal()*** può essere dichiarata in modo più gradevole:

```
sighandler_t signal (n_segnale, sighandler_t gestore);
```

69.15.2 Tipo speciale

A parte il caso di '**`sighandler_t`**' che non fa parte dello standard del linguaggio, il file '`include.h`' definisce il tipo '**`sig_atomic_t`**', il cui uso non viene precisato dai documenti ufficiali. Si chiarisce solo che deve trattarsi di un valore intero, possibilmente di tipo volatile, a cui si possa accedere attraverso una sola istruzione elementare del linguaggio macchina (in modo tale che la lettura o la modifica del suo contenuto non possa essere sospesa a metà da un'interruzione di qualunque genere).

```
typedef int sig_atomic_t;
```

Nell'esempio, il tipo '**`sig_atomic_t`**' viene dichiarato come equivalente al tipo '**`int`**', supponendo che l'accesso alla memoria per un tipo intero normale corrisponda a un'operazione «atomica» nel linguaggio macchina. A ogni modo, il tipo a cui corrisponde '**`sig_atomic_t`**' può dipendere da altri fattori, mentre l'unico vincolo nel rango è quello di poter contenere i valori rappresentati dalle macro-variabili ***SIG…***, le quali individuano mnemonicamente i segnali.

Il programmatore che deve memorizzare un segnale in una variabile, potrebbe usare per questo il tipo '**`sig_atomic_t`**'.

69.15.3 Denominazione dei segnali

Un gruppo di macro-variabili definisce l'elenco dei segnali gestibili. Lo standard del linguaggio ne prescrive solo una quantità minima, mentre il sistema operativo può richiederne degli altri. Teoricamente l'associazione del numero al nome simbolico del segnale è libera, ma in pratica la concordanza con altri standard prescrive il rispetto di un minimo di uniformità.

```
#define SIGINT          2
#define SIGILL          4
#define SIGABRT         6
#define SIGFPE          8
#define SIGSEGV        11
#define SIGTERM        15
```

Tabella 69.208. Denominazione dei segnali indispensabili al linguaggio C.

Denominazione	Significato mnemonico	Descrizione
SIGABRT	*abort*	Deriva da una terminazione anomala che può essere causata espressamente dall'uso della funzione ***abort()***.
SIGFPE	*floating point exception*	Viene provocato da un'operazione aritmetica errata, come la divisione per zero o uno straripamento del risultato.
SIGILL	*illegal*	Istruzione «illegale».
SIGINT	*interrupt*	Deriva dalla ricezione di una richiesta interattiva di attenzione, quale può essere quella di un'interruzione.
SIGSEGV	*segmentation violation*	Deriva da un accesso alla memoria non valido, per esempio oltre i limiti fissati.
SIGTERM	*termination*	Indica la ricezione di una richiesta di terminazione del funzionamento del programma.

69.15.4 Segnali secondo POSIX

Lo standard POSIX prescrive un insieme minimo di segnali più numeroso, attribuendo anche un'azione predefinita a carico del sistema operativo.

Tabella 69.209. Denominazione dei segnali indispensabili allo standard POSIX.

Denominazione	Azione predefinita	Descrizione
SIGABRT	terminazione anomala	Conclusione prematura del processo elaborativo.
SIGALRM	terminazione normale	Segnale di allarme da un orologio programmabile.
SIGBUS	terminazione anomala	Accesso errato alla memoria.
SIGCHLD	ignorato	Conclusione, sospensione o continuazione di un processo figlio.
SIGCONT	continuazione	Ripresa dell'esecuzione di un processo, se risulta sospeso.
SIGFPE	terminazione anomala	Operazione aritmetica errata.
SIGHUP	terminazione normale	Aggancio (interruzione del collegamento con il terminale).
SIGILL	terminazione anomala	Istruzione illegale.
SIGINT	terminazione normale	Segnale di interruzione dal terminale.
SIGKILL	terminazione normale	Eliminazione del processo elaborativo (senza la possibilità che il segnale sia catturato o ignorato).
SIGPIPE	terminazione normale	Scrittura verso un condotto (*pipe*) senza che alcun processo stia leggendo da lì.
SIGQUIT	terminazione anomala	Segnale di abbandono (*quit*) dal terminale.
SIGSEGV	terminazione anomala	Riferimento errato alla memoria.

Denominazione	Azione predefinita	Descrizione
SIGSTOP	sospensione	Sospensione dell'esecuzione (senza la possibilità che il segnale sia catturato o ignorato).
SIGTERM	terminazione normale	Richiesta di conclusione del funzionamento.
SIGSTOP	sospensione	Segnale di stop dal terminale.
SIGTTIN	sospensione	Processo sullo sfondo che attende di poter leggere (dalla tastiera).
SIGTTOU	sospensione	Processo sullo sfondo che attende di poter scrivere (sullo schermo del terminale).
SIGUSR1	terminazione normale	Segnale 1, il cui scopo è definibile dall'utente.
SIGUSR2	terminazione normale	Segnale 2, il cui scopo è definibile dall'utente.
SIGPOLL	terminazione normale	*Pollable event*.
SIGPROF	terminazione normale	*Profiling timer expired*.
SIGSYS	terminazione anomala	Chiamata di sistema errata.
SIGTRAP	terminazione anomala	Trappola scattata per il tracciamento del codice.
SIGURG	ignorato	Informazione urgente disponibile da un socket.
SIGVTALRM	terminazione normale	*Virtual timer expired*.
SIGXCPU	terminazione anomala	Limite del tempo di CPU superato.
SIGXFSZ	terminazione anomala	Limite della dimensione dei file superato.

L'azione predefinita è quella che deve essere svolta dal sistema operativo se il segnale non viene catturato o non viene ignorato dal programma che lo riceve (tenendo conto che per '**SIGKILL**' e '**SIGSTOP**' i programmi non possono intervenire). Una «terminazione normale» implica la conclusione normale del processo elaborativo, a parte il fatto che il valore restituito dal processo dipende dal segnale stesso; una «terminazione anomala» implica di solito qualcosa di più, di solito si tratta dello scarico della memoria del processo in un file (*core dump*), per consentire un'analisi di quanto accaduto; la «sospensione» rappresenta un arresto temporaneo, in attesa di un segnale di «continuazione»; un segnale «ignorato» indica che lo stato del processo non viene cambiato, ma ciò non significa che il segnale in sé sia privo di conseguenze.

Il segnale '**SIGCHLD**' che formalmente viene indicato come privo di effetti, nella tradizione Unix ha un ruolo molto importante e relativamente complesso, per la gestione della dipendenza dei processi. In un sistema Unix i processi hanno una dipendenza gerarchica, trattata secondo un albero genealogico, dove ogni processo ha un genitore. Dato che la conclusione di un processo produce un valore che dovrebbe essere raccolto dal genitore che lo ha avviato (oppure che lo ha adottato, nel caso il genitore vero sia defunto nel frattempo), quando un processo muore (termina di funzionare per qualunque motivo), il genitore riceve un segnale '**SIGCHLD**': se il genitore è in attesa del valore di uscita del processo defunto, lo raccoglie e le tracce residue di tale processo possono essere distrutte definitivamente; altrimenti, se il segnale non viene catturato il processo defunto viene eliminato senza comunicare tale valore al genitore.

69.15.5 Gestori fittizi di segnali

Lo standard prescrive di definire tre macro-variabili che devono espandersi in un puntatore a quel tipo di funzione che deve essere in grado di gestire le azioni da compiere in relazione alla ricezione di un certo segnale. Tuttavia, questo puntatore non deve essere rivolto a una funzione vera, ma averne solo la forma. In pratica, si usano dei valori interi con un valore assoluto molto piccolo e si esegue un cast per trasformarli in puntatori a funzione, come già accennato.

Per ottenere questo risultato, si possono dichiarare le macro-variabili in due modi equivalenti, con la differenza che il secondo è probabilmente più difficile da interpretare:

```
typedef void (*sighandler_t) (int); // Il tipo
                                    // «sighandler_t» è un
                                    // puntatore a funzione
                                    // per la gestione dei
                                    // segnali con parametro
                                    // «int» che
                                    // restituisce «void».
//
// Funzioni non dichiarabili
//
#define SIG_ERR ((sighandler_t) -1) // Trasforma un numero
#define SIG_DFL ((sighandler_t) 0)  // intero  in un tipo
#define SIG_IGN ((sighandler_t) 1)  // «sighandler_t»,
                                    // ovvero un puntatore
                                    // a funzione che però
                                    // non esiste realmente.
```

```
//
// Funzioni non dichiarabili
//
#define SIG_ERR ((void (*) (int)) -1) // Trasforma un numero
#define SIG_DFL ((void (*) (int)) 0)  // intero in un
#define SIG_IGN ((void (*) (int)) 1)  // puntatore a una
                                      // funzione che ha un
                                      // parametro «int» e
                                      // restituisce «void».
```

Lo standard sottolinea il fatto che il numero trasformato in puntatore non deve poter corrispondere all'indirizzo di alcuna funzione reale; pertanto i valori usati possono essere solo molto bassi o molto alti (in termini di valore assoluto), contando sul fatto che a tali indirizzi non ci possano essere funzioni reali. In pratica, non deve succedere che venga dichiarata una funzione per la gestione di un segnale che finisca per avere proprio tali indirizzi, perché se così fosse, non verrebbe avviata, ma al suo posto verrebbe considerata l'azione che una di queste macro-variabili simboleggia.

Tabella 69.212. Macro-variabili per la gestione predefinita dei segnali.

Denominazione	Significato mnemonico	Descrizione
`SIG_DFL`	*default*	Indica simbolicamente che l'azione da compiere alla ricezione del segnale deve essere quella predefinita.
`SIG_IGN`	*ignore*	Indica simbolicamente che alla ricezione del segnale si procede come se nulla fosse accaduto.
`SIG_ERR`	*error*	Rappresenta un risultato errato nell'uso della funzione ***signal()***.

69.15.6 Funzioni

La funzione ***signal()*** viene usata per associare un «gestore di segnale», costituito dal puntatore a una funzione, a un certo segnale; tutto questo allo scopo di attivare automaticamente quella tale funzione al verificarsi di un certo evento che si manifesta tramite un certo segnale.

La funzione ***signal()*** restituisce un puntatore alla funzione che precedentemente si doveva occupare di quel segnale. Se invece l'operazione fallisce, ***signal()*** esprime questo errore restituendo il valore ***SIG_ERR***, spiegando così il motivo per cui questo debba avere l'apparenza di un puntatore a funzione.

Per la stessa ragione per cui esiste ***SIG_ERR***, le macro-variabili ***SIG_DFL*** e ***SIG_IGN*** vanno usate come gestori di segnali, rispettivamente, per ottenere il comportamento predefinito o per far sì che i segnali siano ignorati semplicemente.

In linea di principio si può ritenere che nel proprio programma esista una serie iniziale di dichiarazioni implicite per cui si associano tutti i segnali gestibili a ***SIG_DFL***:

```
…
signal (segnale, SIG_DFL);
…
```

In base al fatto che sia stata dichiarato o meno il tipo '**`sighandler_t`**', la funzione potrebbe avere i prototipi seguenti:

```
sighandler_t signal (int sig, sighandler_t handler);
```

```
void (*signal (int sig, void (*handler) (int))) (int);
```

L'altra funzione da considerare è ***raise()***, con la quale si attiva volontariamente un segnale, dal quale poi dovrebbero o potrebbero sortire delle conseguenze, come stabilito in una fase precedente attraverso ***signal()***. La funzione ***raise()*** è molto semplice:

```
int raise (int sig);
```

La funzione richiede come argomento il numero del segnale da attivare e restituisce un valore pari a zero in caso di successo, altrimenti restituisce un valore diverso da zero. Naturalmente, a seconda dell'azione che viene intrapresa all'interno del programma, a seguito della ricezione del segnale, può darsi che dopo questa funzione non venga eseguito altro, pertanto non è detto che possa essere letto il valore che la funzione potrebbe restituire.

69.15.7 Esempio

Viene proposto un esempio che serve a dimostrare il meccanismo di provocazione e intercettazione dei segnali:

```
#include <stdio.h>
#include <signal.h>

void sig_generic_handler (int sig)
{
  printf ("Ho intercettato il segnale n. %i.\n", sig);
}

void sigfpe_handler (int sig)
{
  printf ("Attenzione: ho intercettato il segnale "
          "SIGFPE (%i)\n"
          "            e devo concludere il "
          "funzionamento!\n", sig);
  exit (sig);
}

void sigterm_handler (int sig)
{
  printf ("Attenzione: ho intercettato il segnale "
          "SIGTERM (%i),\n"
          "            però non intendo rispettarlo.\n",
          sig);
}

void sigint_handler (int sig)
{
  printf ("Attenzione: ho intercettato il segnale "
          "SIGINT (%i),\n"
          "            però non intendo rispettarlo.\n",
          sig);
}

int main (void)
{
  signal (SIGFPE,  sigfpe_handler);
  signal (SIGTERM, sigterm_handler);
  signal (SIGINT,  sigint_handler);
```

```
  signal (SIGILL,  sig_generic_handler);
  signal (SIGSEGV, sig_generic_handler);

  int c;
  int x;

  printf ("[0][Invio] divisione per zero\n");
  printf ("[c][Invio] provoca un segnale SIGINT\n");
  printf ("[t][Invio] provoca un segnale SIGTERM\n");
  printf ("[q][Invio] conclude il funzionamento\n");
  while (1)
    {
      c = getchar();
      if (c == '0')
        {
          printf ("Sto per eseguire una divisione per "
                  "zero:\n");
          x = x / 0;
        }
      else if (c == 'c')
        {
          raise (SIGINT);
        }
      else if (c == 't')
        {
          raise (SIGTERM);
        }
      else if (c == 'q')
        {
          return 0;
        }
    }
  return 0;
}
```

All'inizio del programma vengono definite delle funzioni per il trattamento delle situazioni che hanno provocato un certo segnale. Nella funzione ***main()***, prima di ogni altra cosa, si associano tali funzioni ai segnali principali, quindi si passa a un ciclo senza fine, nel quale possono essere provocati dei segnali premendo un certo tasto, come suggerito da un breve menù. Per esempio è possibile provocare la condizione che si verifica tentando di dividere un numero per zero:

```
[0][Invio] divisione per zero
[c][Invio] provoca un segnale SIGINT
[t][Invio] provoca un segnale SIGTERM
[q][Invio] conclude il funzionamento
```

0 [*Invio*]

```
Sto per eseguire una divisione per zero:
Attenzione: ho intercettato il segnale SIGFPE (8)
            e devo concludere il funzionamento!
```

La divisione per zero fa scattare il segnale '**SIGFPE**' che viene intercettato dalla funzione ***sigfpe_handler()***, la quale però non può far molto e così conclude anche il funzionamento del programma.

Attraverso il menù è possibile provocare anche un segnale '**SIGINT**' e un segnale '**SIGTERM**', ma per questo è più interessante provare con i mezzi che dovrebbe offrire il sistema operativo:

```
[0][Invio] divisione per zero
[c][Invio] provoca un segnale SIGINT
[t][Invio] provoca un segnale SIGTERM
[q][Invio] conclude il funzionamento
```

[*Ctrl c*] [*Invio*]

```
Attenzione: ho intercettato il segnale SIGINT (2),
            però non intendo rispettarlo.
```

Utilizzando un sistema operativo Unix o simile, da un altro terminale, o da un'altra console, è possibile inviare un segnale specifico al programma:

$ **kill** ***n_processo*** [*Invio*]

```
Attenzione: ho intercettato il segnale SIGTERM (15),
            però non intendo rispettarlo.
```

$ **kill -s 4** ***n_processo*** [*Invio*]


```
Ho intercettato il segnale n. 4.
```

$ **kill -s 11** ***n_processo*** [*Invio*]

```
Ho intercettato il segnale n. 11.
```

Secondo l'esempio, i segnali 4 e 11 sono, rispettivamente, '**SIGILL**' e '**SIGSEGV**'.

[*q*] [*Invio*]

69.16 File «time.h»

Il file '`time.h`' della libreria standard definisce principalmente delle funzioni per il trattamento delle informazioni data-orario. Non è stabilito in che modo venga rappresentato il tempo internamente alle funzioni, anche se di norma si tratta di un valore intero che esprime una quantità di secondi o di frazioni di secondo (si veda eventualmente la realizzazione del file '`time.h`' e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.29).

69.16.1 Il tempo di CPU

La funzione ***clock()*** consente di ottenere il tempo di utilizzo del microprocessore (CPU), espresso virtualmente in cicli di CPU. In pratica, viene definita la macro-variabile ***CLOCKS_PER_SEC***, contenente il valore che esprime convenzionalmente la quantità di cicli di CPU per secondo; quindi, il valore restituito dalla funzione ***clock()*** si traduce in secondi dividendolo per ***CLOCKS_PER_SEC***. Il valore restituito dalla funzione ***clock()*** e l'espressione in cui si traduce la macro-variabile ***CLOCKS_PER_SEC*** sono di tipo '**clock_t**':

```
typedef long int clock_t;   // Unità di tempo convenzionale
                            // che rappresenta un ciclo
                            // virtuale di CPU.

#define CLOCKS_PER_SEC  1000000L  // Valore convenzionale di
                                  // 1 s, in termini di
                                  // cicli virtuali di CPU.

clock_t clock (void);             // Tempo di utilizzo della
                                  // CPU.
```

La funzione ***clock()*** restituisce il tempo di CPU espresso in unità '**clock_t**', utilizzato dal processo elaborativo a partire dall'avvio del programma. Se la funzione non è in grado di dare questa indicazione, allora restituisce il valore `-1`, o più precisamente '**(clock_t) (-1)**'.

Per valutare l'intervallo di tempo di utilizzo della CPU, da una certa posizione del programma, a un'altra, occorre memorizzare i valori ottenuti dalla funzione e poi procedere a una sottrazione.

Per comprendere il significato della funzione ***clock()***, del tipo '**clock_t**' e della macro-variabile ***CLOCKS_PER_SEC***, viene proposto un esempio molto semplice, ma completo, dove si intende che il tipo '**clock_t**' sia intero e sia contenibile in una variabile di tipo '**long int**':

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

int
main (int argc, char *argv[])
{
  clock_t t0;
  clock_t t1;
  long int i;
  long int x;

  t0 = clock ();
  printf ("Tempo iniziale: %li/%li\n",
          (long int) t0, (long int) CLOCKS_PER_SEC);

  for (i = 0; i < 10000000; i++)
    {
      x = i * 123;
    }

  t1 = clock ();
```

```
  printf ("Tempo finale: %li/%li\n",
          (long int) t1, (long int) CLOCKS_PER_SEC);

  return 0;
}
```

Avviando questo programma si potrebbe leggere un risultato simile al testo seguente, dove si vede un valore di '**CLOCKS_PER_SEC**' pari a 1000000:

```
Tempo iniziale: 0/1000000
Tempo finale: 20000/1000000
```

69.16.2 Rappresentazione interna del tempo

Generalmente, nei sistemi Unix si tratta il tempo come una quantità di secondi trascorsi a partire da un'epoca di riferimento, che tradizionalmente coincide con l'ora zero del giorno 1 gennaio 1970. Da questo concetto deriva il tipo '**time_t**' della libreria, che, secondo lo standard, rappresenta la quantità di unità di tempo trascorsa a partire da un'epoca di riferimento.

```
typedef long int time_t; // Unità di tempo convenzionale
                         // per le informazioni data-orario.
```

Ammesso che si tratti di un numero intero, così come viene ipotizzato dall'esempio proposto, il rango costituisce il limite alle date rappresentabili. Pertanto, se il tipo '**time_t**' viene dichiarato come numero intero con segno, a 32 bit, per rappresentare una quantità di secondi (come nella tradizione Unix), significa che si possono rappresentare al massimo 24855 giorni, pari a circa 68 anni.[8] Se l'epoca di riferimento è il 1970, si può arrivare al massimo al 2038.

69.16.3 Rappresentazione strutturata del tempo

La libreria standard prescrive che sia definito il tipo '**struct tm**', con il quale è possibile rappresentare tutte le informazioni relative a un certo tempo, secondo le convenzioni umane. Lo standard prescrive con precisione i membri minimi della struttura e l'intervallo di valori che possono contenere:

```
struct tm {
    int tm_sec;     // Secondi:                 da 0 a 60.
    int tm_min;     // Minuti:                  da 0 a 59.
    int tm_hour;    // Ora:                     da 0 a 23.
    int tm_mday;    // Giorno del mese:         da 1 a 31.
    int tm_mon;     // Mese dell'anno:          da 0 a 11.
    int tm_year;    // Anno dal 1900.
    int tm_wday;    // Giorno della settimana: da 0 a 6
                    // con lo zero corrispondente alla
                    // domenica.
    int tm_yday;    // Giorno dell'anno:        da 0 a 365.
    int tm_isdst;   // Ora estiva. Contiene un valore
                    // positivo se è in vigore l'ora estiva;
                    // zero se l'ora è quella «normale»
                    // ovvero quella invernale;
                    // un valore negativo se l'informazione
                    // non è disponibile.
};
```

Si può osservare che il mese viene rappresentato con valori che vanno da 0 a 11, pertanto gennaio si indica con lo zero e dicembre con il numero 11; inoltre, l'intervallo ammesso per i secondi consente di rappresentare un secondo in più, dato che l'intervallo corretto sarebbe da 0 a 59; infine, il fatto che i giorni dell'anno vadano da 0 (il primo) a 365 (l'ultimo), significa che negli anni normali i valori vanno da 0 a 364, mentre negli anni bisestili si arriva a contare fino a 365.

69.16.4 Funzioni per l'elaborazione di valori legati al tempo

Un gruppo di funzioni dichiarate nel file '`time.h`' ha lo scopo di elaborare in qualche modo le informazioni legate al tempo ed eventualmente di convertirle in formati diversi. Queste funzioni trattano il tempo in forma di variabili di tipo '**time_t**' o di tipo '**struct tm**'.[9]

La variabile di tipo '**time_t**' che viene usata in queste funzioni potrebbe esprimere un valore riferito al tempo universale (UT), mentre le funzioni che la utilizzano dovrebbero tenere conto del fuso orario, in base alle informazioni che può offrire il sistema operativo.

69.16.4.1 Funzione «time()»

La funzione ***time()*** determina il tempo attuale secondo il calendario del sistema operativo, restituendolo nella forma del tipo '**time_t**'. La funzione richiede un parametro, costituito da un puntatore di tipo '**time_t ***': se questo puntatore è valido, la stessa informazione che viene restituita viene anche memorizzata nell'indirizzo indicato da tale puntatore.

```
time_t  time (time_t *timer);
```

In pratica, se è possibile, l'informazione data-orario raccolta dalla funzione, viene anche memorizzata in ***timer**.

Se la funzione non può fornire l'informazione richiesta, allora restituisce il valore `-1`, o più precisamente: '**(time_t) (-1)**'.

69.16.4.2 Funzione «difftime()»

La funzione ***difftime()*** calcola la differenza tra due date, espresse in forma '**time_t**' e restituisce l'intervallo in secondi, in una variabile in virgola mobile, di tipo '**double**':

```
double  difftime (time_t time1, time_t time0);
```

Per la precisione, viene eseguito ***time1*** − ***time0*** e di conseguenza va il segno del risultato.

69.16.4.3 Funzione «mktime()»

La funzione ***mktime()*** riceve come argomento il puntatore a una variabile strutturata di tipo '**struct tm**', contenente le informazioni sull'ora locale, e determina il valore di quella data secondo la rappresentazione interna, di tipo '**time_t**':

```
time_t  mktime (struct tm *timeptr);
```

La funzione tiene in considerazione solo alcuni membri della struttura; per la precisione, non considera il giorno della settimana e il giorno dell'anno; inoltre, ammette anche valori al di fuori degli intervalli stabiliti per i vari membri della struttura; infine, considera un valore negativo per il membro ***timeptr->tm_isdst*** come la richiesta di determinare se sia o meno in vigore l'ora estiva per la data indicata.

Se la funzione non è in grado di restituire un valore rappresentabile nel tipo '**time_t**', o comunque se non può eseguire il suo compito, restituisce il valore `-1`, o più precisamente '**(time_t) (-1)**'. Se invece tutto procede regolarmente, la funzione provvede anche a correggere i valori dei vari membri della struttura e a ricalcolare il giorno della settimana e dell'anno.

L'esempio successivo mostra la dichiarazione di una variabile strutturata di tipo '**struct tm**', assegnando ai suoi membri dei valori non corretti. Con l'aiuto della funzione ***mktime()*** si ricostruisce la data secondo le convenzioni comuni:

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

int
main (int argc, char *argv[])
{
  struct tm t;
  time_t tx;

  t.tm_year = 107;    // 2007 - 1900
  t.tm_mon  = 5;
  t.tm_mday = 33;
  t.tm_hour = 0;
  t.tm_min  = 0;
  t.tm_sec  = 60;
  t.tm_isdst = -1;

  printf ("%i/%i/%i %i:%i:%i\n",
```

```
          t.tm_year + 1900, t.tm_mon + 1, t.tm_mday,
          t.tm_hour, t.tm_min, t.tm_sec);

  tx = mktime (&t);

  if (tx == (time_t) (-1))
    {
      printf ("Errore! %li\n", (long int) tx);
    }
  else
    {
      printf ("%i/%i/%i %i:%i:%i\n",
               t.tm_year + 1900, t.tm_mon + 1, t.tm_mday,
               t.tm_hour, t.tm_min, t.tm_sec);

      printf ("giorno della settimana: %i\n", t.tm_wday);
      printf ("giorno dell'anno: %i\n", t.tm_yday + 1);
      printf ("ora estiva: %i\n", t.tm_isdst);
    }
  return 0;
}
```

Eseguendo questo programma di esempio si dovrebbe ottenere il testo seguente:

```
2007/6/33 0:0:60
2007/7/3 0:1:0
giorno della settimana: 2
giorno dell'anno: 184
ora estiva: 1
```

69.16.4.4 Funzioni «gmtime()» e «localtime()»

Le funzioni ***gmtime()*** e ***localtime()*** hanno in comune il fatto di ricevere come argomento il puntatore di tipo '**`time_t *`**', a un'informazione data-orario, per restituire il puntatore a una variabile strutturata di tipo '**`struct tm *`**'. In altri termini, le due funzioni convertono una data espressa nella forma del tipo '**`time_t`**', in una data suddivisa nella struttura '**`struct tm`**':

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

Nell'ambito di queste funzioni, è ragionevole supporre che l'informazione di tipo '**`time_t`**' a cui fanno riferimento, sia espressa in termini di tempo universale e che le funzioni stesse abbiano la possibilità di stabilire il fuso orario e la modalità di regolazione dell'ora estiva.

In ogni caso, la differenza tra le due funzioni sta nel fatto che ***gmtime()*** traduce il tempo a cui punta il suo argomento in una struttura contenente la data tradotta secondo il tempo coordinato universale, mentre ***localtime()*** la traduce secondo l'ora locale.

Va osservato che queste funzioni restituiscono un puntatore a un'area di memoria che può essere sovrascritta da altre chiamate alle stessi funzioni o a funzioni simili.

69.16.5 Conversione in stringa

Un piccolo gruppo di funzioni del file '`time.h`' è destinato alla conversione dei valori data-orario in stringhe, per l'interpretazione umana.

69.16.5.1 Funzione «asctime()»

La funzione ***asctime()*** converte un'informazione data-orario, espressa nella forma di una struttura '**`struct tm`**', in una stringa che esprime l'ora locale, usando però una rappresentazione fissa in lingua inglese:

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

In pratica, dal momento che la data e l'orario vanno espressi secondo le convenzioni della lingua inglese, lo standard stesso descrive completamente questa funzione e il listato seguente è tratto letteralmente da tale definizione:


```
#include <time.h>
char *asctime(const struct tm *timeptr)
{
  static const char wday_name[7][3] = {
      "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
  };
  static const char mon_name[12][3] = {
      "Jan", "Feb", "Mar", "Apr", "May", "Jun",
      "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
  };
  static char result[26];
  sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %i\n",
      wday_name[timeptr->tm_wday],
      mon_name[timeptr->tm_mon],
      timeptr->tm_mday, timeptr->tm_hour,
      timeptr->tm_min, timeptr->tm_sec,
      1900 + timeptr->tm_year);
  return result;
}
```

69.16.5.2 Funzione «ctime()»

La funzione ***ctime()*** converte un'informazione data-orario, espressa nella forma del tipo '**`time_t`**' in una stringa che esprime l'ora locale, usando però una rappresentazione fissa in lingua inglese:

```
char *ctime (const time_t *timer);
```

Il comportamento di questa funzione è tale da generare una stringa analoga a quella della funzione ***asctime()***, tanto che la si potrebbe esprimere così:

```
char *ctime (const time_t *timer)
{
  return asctime (localtime (timer));
}
```

Oppure, come macroistruzione, così:

```
#define ctime(t) (asctime (localtime (t)));
```

69.16.5.3 Funzione «strftime()»

La funzione ***strftime()*** si occupa di interpretare il contenuto di una struttura di tipo '**`struct tm`**' e di tradurlo in un testo, secondo una stringa di composizione libera. In altri termini, questa funzione si comporta in modo simile a ***printf()***, dove l'input è costituito dalla struttura contenente le informazioni data-orario.

```
size_t strftime (char * restrict s,
                 size_t maxsize,
                 const char * restrict format,
                 const struct tm * restrict timeptr);
```

Dal modello del prototipo della funzione, si vede che questa restituisce un valore numerico di tipo '**`size_t`**'. Questo valore rappresenta la quantità di elementi[10] che sono stati scritti nella stringa di destinazione, rappresentata dal primo parametro. Dal computo di questi elementi è escluso il carattere nullo di terminazione, benché venga comunque aggiunto dalla funzione.

La funzione richiede, nell'ordine: un array di caratteri da utilizzare per comporre il testo; la dimensione massima di questo array; la stringa di composizione, contenente del testo costante e degli specificatori di conversione; il puntatore alla struttura contenente le informazioni data-orario da usare nella conversione.

La funzione termina il proprio lavoro con successo solo se può scrivere nell'array di destinazione il testo composto secondo le indicazioni della stringa di composizione, includendo anche il carattere nullo di terminazione. Se ciò non avviene, il valore restituito dalla funzione è zero e il contenuto dell'array di destinazione è imprecisato.

Il listato successivo mostra un programma completo che dimostra il funzionamento di ***strftime()***. Va osservato che la conversione eseguita da tale funzione è sensibile alla configurazione locale; precisamente dipende dalla categoria '**`LC_TIME`**':

```
#include <stdio.h>
#include <locale.h>
#include <time.h>
```

```
int
main (int argc, char *argv[])
{
  char s[100];
  time_t t      = time (NULL);
  struct tm *tp = localtime (&t);
  int dim;
  setlocale (LC_ALL, "it_IT.UTF-8");
  dim = strftime (s, 100, "Ciao amore: sono "
                          "le %H:%M del %d %B %Y.", tp);
  printf ("%d: %s\n", dim, s);
  return 0;
}
```

Ecco cosa si potrebbe ottenere eseguendo questo programma:

```
45: Ciao amore: sono le 09:32 del 27 giugno 2012.
```

Nella tabella successiva vengono elencati gli specificatori di conversione principali. Sono ammissibili delle varianti, con l'aggiunta di modificatori, che però non vengono descritte. Per esempio è ammissibile l'uso degli specificatori '**%Ec**' e '**%Od**', per indicare rispettivamente una variante di '**%c**' e '**%d**'.

Tabella 69.244. Specificatori di conversione usati dalla funzione ***strftime()***.

Specificatore	Corrispondenza
%C	*century* Il secolo, ottenuto dividendo l'anno per 100 e ignorando i decimali.
%y %Y	*year* L'anno: nel primo caso si mostrano solo le ultime due cifre, mentre nel secondo si mostrano tutte.
%b | %h %B	Rispettivamente, il nome abbreviato e il nome per esteso del mese.
%m	*month* Il numero del mese, da 01 a 12.
%d %e	*day* Il giorno del mese, in forma numerica, da 1 a 31, utilizzando sempre due cifre: nel primo caso si aggiunge eventualmente uno zero; nel secondo si aggiunge eventualmente uno spazio.
%a %A	Rispettivamente, il nome abbreviato e il nome per esteso del giorno della settimana.
%H %L	*hour* L'ora, espressa rispettivamente a 24 ore e a 12 ore.
%p	La sigla da usare, secondo la configurazione locale, per specificare che si tratta di un'ora antimeridiana o pomeridiana. Nella convenzione inglese si ottengono, per esempio, le sigle «AM» e «PM».
%r	L'ora espressa a 12 ore, completa dell'indicazione se trattasi di ora antimeridiana o pomeridiana, secondo le convenzioni locali.
%R	L'ora e i minuti, equivalente a '**%H:%M**'.
%M	*minute* I minuti, da 00 a 59.
%S	*second* I secondi, espresso con valori da 00 a 60.
%T	*time* L'ora, i minuti e i secondi, equivalente a '**%H:%M:%S**'.
%z %Z	*time zone* La rappresentazione del fuso orario, nel primo caso come distanza dal tempo coordinato universale (UTC), mentre nel secondo si usa una rappresentazione conforme alla configurazione locale.
%j	*julian* Il giorno dell'anno, usando sempre tre cifre numeriche: da 001 a 366.

Specificatore	Corrispondenza
%g %G	L'anno a cui appartiene la settimana secondo lo standard ISO 8601: nel primo caso si mostrano solo le ultime due cifre, mentre nel secondo si ha l'anno per esteso. Secondo lo standard ISO 8601 la settimana inizia con lunedì e la prima settimana dell'anno è quella che include il 4 gennaio.
%V	Il numero della settimana secondo lo standard ISO 8601. I valori vanno da 01 a 53. Secondo lo standard ISO 8601 la settimana inizia con lunedì e la prima settimana dell'anno è quella che include il 4 gennaio.
%u %w	Il giorno della settimana, espresso in forma numerica, dove, rispettivamente, si conta da 1 a 7, oppure da 0 a 6. Zero e sette rappresentano la domenica; uno è il lunedì.
%U %W	Il numero della settimana, contando, rispettivamente, dalla prima domenica o dal primo lunedì di gennaio. Si ottengono cifre da 00 a 53.
%x	La data, rappresentata secondo le convenzioni locali.
%X	L'ora, rappresentata secondo le convenzioni locali.
%c	La data e l'ora, rappresentate secondo le convenzioni locali.
%D	*date* La data, rappresentata come '**%m/%d/%Y**'.
%F	La data, rappresentata come '**%Y-%m-%d**'.
%n	Viene rimpiazzato dal codice di interruzione di riga.
%t	Viene rimpiazzato da una tabulazione orizzontale.
%%	Viene rimpiazzato da un carattere di percentuale.

69.17 File «stdio.h»

Il file '`stdio.h`' della libreria standard è quello che fornisce le funzioni più importanti e in generale è il più complesso da realizzare, in quanto dipende strettamente dal meccanismo di gestione dei file del sistema operativo (si veda eventualmente la realizzazione del file '`stdio.h`' e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.18). L'elemento più delicato che viene definito qui è il tipo di dati '**FILE**', da cui dipende quasi tutto il resto.

Alle complicazioni che esistevano già alla nascita del linguaggio, nei primi sistemi Unix, si aggiungono attualmente quelle relative alla distinzione tra file di testo e file binari, oltre che quelle relative alla gestione dei caratteri multibyte, per cui la lettura o la scrittura attraverso un flusso di dati deve tenere conto dello stato di completamento di tali informazioni.

Il file '`stdio.h`' definisce le funzioni principali per l'accesso ai file e una serie di funzioni per la lettura e scrittura di dati formattati (si vedano ***print()***, ***scanf()*** e altre analoghe), ma altre funzioni realizzate espressamente per caratteri e stringhe estese (formate da elementi '**wchar_t**') si trovano nel file '`wchar.h`'.

I file proposti che si basano sugli esempi del capitolo sono incompleti, in quanto manca la dichiarazione del tipo '**FILE**' e del tipo '**fpos_t**'.

69.17.1 Tipi

Il file '`stdio.h`', oltre a '**size_t**' che fa già parte del file '`stddef.h`', e di '**va_list**' che fa già parte del file '`stdarg.h`', dichiara due tipi di dati a uso specifico per la gestione dei file: '**FILE**' e '**fpos_t**', realizzati normalmente attraverso delle strutture.

Il tipo '**fpos_t**' serve a rappresentare tutte le informazioni necessarie a specificare univocamente le posizioni interne a un file, per gli scopi delle funzioni ***fgetpos()*** e ***fsetpos()***. Il tipo '**FILE**' deve poter esprimere tutte le informazioni necessarie a controllare un flusso di file (ovvero le operazioni su un file aperto), in particola-

re le posizioni correnti, il puntatore alla memoria tampone (*buffer*), l'indicatore di errore e di fine file.

```
typedef struct { /* omissis */ } fpos_t;

typedef struct { /* omissis */ } FILE;
```

L'organizzazione effettiva delle strutture che costituiscono i tipi '**fpos_t**' e '**FILE**' dipende strettamente dal sistema operativo (nel contesto particolare della propria architettura); pertanto, per poterne approfondire le caratteristiche, occorre prima uno studio dettagliato delle funzionalità del sistema operativo stesso.

Alcune funzioni aggiunte dallo standard POSIX utilizzano anche il tipo '**off_t**', che è descritto nel file di intestazione '`sys/types.h`'.

69.17.2 Macro-variabili varie

Il file '`stdio.h`' dichiara la macro-variabile ***NULL***, come già avviene nel file '`stddef.h`', assieme ad altre macro-variabili a uso delle funzioni dichiarate al proprio interno. Quelle più semplici sono descritte nella tabella 69.248. L'esempio proposto della dichiarazione di tali macro-variabili è molto approssimativo:

```
#define _IOFBF          0 // Input-output fully buffered.
#define _IOLBF          1 // Input-output line buffered.
#define _IONBF          2 // Input-output with no buffering.

#define EOF             (-1)

#define FOPEN_MAX       10
#define FILENAME_MAX    254
#define L_tmpnam        FILENAME_MAX

#define SEEK_SET        0       // Dall'inizio.
#define SEEK_CUR        1       // Dalla posizione corrente.
#define SEEK_END        2       // Dalla fine del file.

#define TMP_MAX         100000  // Si ipotizza di usare nomi
                                // da «TMP00000.tmp» a
                                // «TMP99999.tmp».
```

La porzione successiva riguarda le estensioni POSIX:

```
#define L_ctermid       14
#define P_tmpdir        "/tmp"
```

Tabella 69.248. Macro-variabili comuni per le funzioni di '`stdio.h`'.

Denominazione	Significato mnemonico	Descrizione
`_IOFBF`	*input output fully buffered*	Indica simbolicamente la richiesta di utilizzo di una memoria tampone a blocchi.
`_IOLBF`	*input output line buffered*	Indica simbolicamente la richiesta di utilizzo di una memoria tampone gestita a righe di testo.
`_IONBF`	*input output with no buffering*	Indica simbolicamente la richiesta di non utilizzare alcuna memoria tampone.
`BUFSIZE`	*buffer size*	Rappresenta la dimensione predefinita della memoria tampone.
`EOF`	*end of file*	È un numero intero di tipo '**int**', negativo, che rappresenta il raggiungimento della fine del file. È in pratica ciò che si ottiene leggendo oltre la fine del file.
`FOPEN_MAX`	*file open max*	Il numero di file che un processo elaborativo può aprire simultaneamente, in base alle limitazioni poste dal sistema operativo.

Denominazione	Significato mnemonico	Descrizione
`FILENAME_MAX`		La dimensione di un array di elementi '**char**', tale da essere abbastanza grande da contenere il nome del file più lungo (incluse le eventuali sequenze multibyte) che il sistema consenta di gestire.
`L_tmpnam`	*temporary name*	La dimensione di un array di elementi '**char**', tale da essere abbastanza grande da contenere il nome di un file temporaneo generato dalla funzione ***tmpnam()***.
`SEEK_CUR`	*seek current*	Indica di eseguire un posizionamento a partire dalla posizione corrente del file.
`SEEK_END`		Indica di eseguire un posizionamento a partire dalla fine di un file.
`SEEK_SET`		Indica di eseguire un posizionamento a partire dall'inizio di un file.
`TMP_MAX`		Rappresenta la quantità massima di nomi di file differenti che possono essere generati dalla funzione ***tmpnam()***.

Tabella 69.249. Macro-variabili aggiunte dalle estensioni POSIX.

Denominazione	Significato mnemonico	Descrizione
`L_ctermid`	*character terminal identity*	La dimensione di un array di elementi '**char**', tale da essere abbastanza grande da contenere il nome del file di dispositivo restituito dalla funzione ***ctermid()***.
`P_tmpdir`	*temporary directory*	Definisce il percorso di una directory temporanea, da scegliere quando ciò che viene specificato con la funzione ***tempnam()*** non è appropriato.

69.17.3 Ipotesi di gestione del tipo «FILE»

Pur non essendo necessario che sia così, si può ipotizzare che per ogni file che possa essere aperto simultaneamente, sia disponibile un elemento di tipo '**FILE**' organizzato in un array. In tal caso, potrebbe essere dichiarato come nell'esempio seguente, già nel file '`stdio.h`', anche se il nome usato per l'array è puramente indicativo:

```
…
FILE _stream[FOPEN_MAX];
…
```

L'uso della macro-variabile ***FOPEN_MAX*** garantisce che siano predisposti esattamente tutti gli elementi necessari alla gestione simultanea del limite di file previsti.

69.17.4 Flussi standard

Lo standard del linguaggio C prescrive che i nomi dei flussi standard previsti siano delle macro-variabili, tali da espandersi in espressioni che rappresentino puntatori di tipo '**FILE ***', diretti ai flussi standard rispettivi. Nel caso del compilatore GNU C i puntatori sono già definiti con lo stesso nome dei flussi e, nel file '`stdio.h`' vi si fa riferimento in qualità di variabili esterne (in quanto dichiarate nella libreria precompilata):

```
extern FILE *stdin;     // Si ipotizza che la libreria C
                        // definisca già i puntatori ai
extern FILE *stdout;    // flussi standard, usando
extern FILE *stderr;    // i nomi predefiniti.
                        //
#define stdin  stdin    // In questo caso, è facile definire
```

```
#define stdout stdout   // le macro che fanno riferimento
#define stderr stderr   // ai flussi standard.
```

Diversamente, nell'ipotesi in cui si gestisca un array di elementi '**FILE**', si potrebbe supporre che i primi tre elementi siano usati per i flussi standard e in tal caso le dichiarazioni delle macro-variabili potrebbero essere fatte così:

```
…
#define stdin  (&_stream[0])
#define stdout (&_stream[1])
#define stderr (&_stream[2])
…
```

69.17.5 Funzioni per la rimozione e la ridenominazione dei file

Le funzioni ***remove()*** e ***rename()*** consentono, rispettivamente di eliminare o di rinominare un file. Il file in questione viene individuato da una stringa, il cui contenuto deve conformarsi alle caratteristiche del sistema operativo. Le due funzioni hanno in comune il fatto di restituire un valore intero (di tipo '**int**'), dove il valore zero rappresenta il completamento con successo dell'operazione, mentre un valore differente indica un fallimento.

```
int remove (const char *filename);
int rename (const char *old, const char *new);
```

La sintassi per l'uso della funzione ***remove()*** è evidente dal suo prototipo, in quanto si attende un solo argomento che è costituito dal nome del file da eliminare; nel caso della funzione ***rename()***, invece, il primo argomento è il nome del file preesistente e il secondo è quello che si vuole attribuirgli.

È importante ribadire che il comportamento delle due funzioni dipende dal sistema operativo. Per esempio, la ridenominazione può provocare la cancellazione di un file preesistente con lo stesso nome che si vorrebbe attribuire a un altro, oppure potrebbe limitarsi a fallire. In un sistema Unix o simile, molto dipende dalla configurazione dei permessi.

69.17.6 Funzioni per la gestione dei file temporanei

Le funzioni ***tmpfile()*** e ***tmpnam()*** servono per facilitare la creazione di file temporanei. La prima crea automaticamente un file di cui non si conosce il nome e la collocazione, aprendolo in aggiornamento (modalità '**wb+**'); la seconda si limita a generare un nome che potrebbe essere usato per creare un file temporaneo:

```
FILE *tmpfile (void);
char *tmpnam  (char *s);
```

L'uso della funzione ***tmpfile()*** è evidente, in quanto non richiede argomenti e restituisce il puntatore al file creato; la seconda richiede l'indicazione di un array di caratteri da poter modificare, restituendo comunque il puntatore all'inizio dello stesso array. In ogni caso va chiarito che il file creato con la funzione '**tmpfile**', una volta chiuso, viene rimosso automaticamente.

Le due funzioni devono essere in grado di poter generare un numero di nomi differente pari almeno al valore rappresentato da '**TMP_MAX**', rimanendo il fatto che non possano essere aperti più di '**FOPEN_MAX**' file e che non possono essere generati file con nomi già esistenti.

Se si utilizza la funzione ***tmpnam()***, l'array di caratteri che costituisce il primo argomento (***s***), viene usato dalla funzione per scriverci il nome del file temporaneo, restituendone poi il puntatore; tale array deve avere una dimensione di almeno '**L_tmpnam**' elementi, come si vede nell'esempio seguente:

```
#include <stdio.h>
int main (void)
{
  char t[L_tmpnam];
  char *p;
  p = tmpnam (t);
  printf ("%s %s\n", t, p);
  return 0;
}
```


Se la funzione ***tmpnam()*** riceve come argomento il puntatore nullo, il nome del file temporaneo viene scritto in un'area di memoria statica che viene sovrascritta a ogni chiamata successiva della funzione stessa.

Entrambe le funzioni, se non possono eseguire il loro compito, restituiscono un puntatore nullo.

Le estensioni POSIX aggiungono anche la funzione ***tempnam()***, la quale ha un comportamento simile a quello di ***tmpnam()***, in quanto non crea il file, ma restituisce il percorso del file che potrebbe essere creato:

```
char *tempnam (const char *dir, const char *prefix);
```

La funzione ***tempnam()*** richiede l'indicazione di una stringa contenente il percorso di una directory, in cui si vuole sia creato un file temporaneo. Se la stringa indica una directory inadatta (perché non esiste o non è accessibile o non gli si può scrivere) oppure si indica il puntatore nullo, si fa riferimento a quanto descritto dalla macro-variabile ***P_tmpdir***. Se anche la directory a cui si riferisce la macro-variabile ***P_tmpdir*** dà dei problemi, è possibile che la funzione decida in qualche modo dove sia possibile collocare un file temporaneo.

Il secondo argomento della funzione è una stringa che rappresenta un prefisso da usare per il nome del file temporaneo. Tale prefisso può essere lungo al massimo cinque caratteri. Se si vuole omettere tale prefisso, basta indicare il puntatore nullo.

Se tutto va bene, la funzione alloca dello spazio in memoria per la stringa che deve contenere il percorso di un file temporaneo che si potrebbe creare (ma senza crearlo) e ne restituisce il puntatore. Quando tale informazione non serve più, la memoria allocata può essere liberata con la funzione ***free()***. Se invece la funzione fallisce nel suo compito, restituisce il puntatore nullo e aggiorna la variabile ***errno***.

69.17.7 Funzioni per l'apertura e la chiusura dei flussi di file

Le funzioni ***fopen()***, ***freopen()*** e ***fclose()***, consentono di aprire e chiudere i file, gestendoli attraverso un puntatore al ***flusso di file*** loro associato (*stream*). Il puntatore in questione è di tipo '**FILE ***'.

```
FILE *fopen   (const char *restrict filename,
               const char *restrict io_mode);

FILE *freopen (const char *restrict filename,
               const char *restrict io_mode,
               FILE *restrict stream);

int   fclose  (FILE *stream);
```

Quando viene aperto un file, gli si associa una variabile strutturata di tipo '**FILE**', contenente tutte le informazioni che servono a gestirne l'accesso. Questa variabile deve rimanere univoca e vi si accede normalmente attraverso un puntatore ('**FILE ***'). Dal momento che per il linguaggio C un file aperto è un flusso, la variabile strutturata che contiene le informazioni necessarie a gestirne l'accesso viene identificata come il flusso stesso, pertanto nei prototipi la variabile che contiene il puntatore di tipo '**FILE ***' viene denominata generalmente ***stream***.

Dal momento che non è compito del programmatore dichiarare la variabile di tipo '**FILE**', in pratica ci si riferisce al flusso di file sempre solo attraverso un puntatore a quella variabile. Pertanto, è più propriamente il puntatore a tale variabile che rappresenta il flusso di file.

L'apertura di un file, oltre che l'indicazione del nome del file, richiede di specificare la modalità, ovvero il tipo di accesso che si intende gestire. Sono previste le modalità elencate nella tabella 69.258.

Tabella 69.258. Modalità di accesso ai file.

Sigla	Mnemonico	Descrizione
`r`	*read*	Accesso in sola lettura di un file di testo.
`w`	*write*	Accesso a un file di testo in scrittura, che implica la creazione del file all'apertura, ovvero il suo troncamento a zero, se esiste già.
`a`	*append*	Accesso a un file di testo in aggiunta, che implica la creazione del file all'apertura, ovvero la sua estensione se esiste già.

`rb`
`wb`
`ab`
	binary	Accesso in lettura, scrittura o aggiunta, ma di tipo binario.

`r+`
`w+`
`a+`
	update	Accesso a un file di testo in lettura, scrittura o aggiunta, assieme alla modalità di aggiornamento. In pratica, con la lettura è consentita anche la scrittura; con la scrittura e l'aggiunta è consentita anche la rilettura.

`rb+`|`r+b`
`wb+`|`w+b`
`ab+`|`a+b`
		Accesso a un file binario in lettura, scrittura o aggiunta, assieme alla modalità di aggiornamento. In pratica, con la lettura è consentita anche la scrittura; con la scrittura e l'aggiunta è consentita anche la rilettura. Si può osservare che il segno '**+**' può essere messo indifferentemente in mezzo o alla fine.

La funzione ***fopen()*** apre il file indicato come primo argomento (una stringa), con la modalità specificata nel secondo (un'altra stringa), restituendo il puntatore al flusso che consente di accedervi (se l'operazione fallisce, la funzione restituisce il puntatore nullo). La modalità di accesso viene espressa attraverso le sigle elencate nella tabella 69.258.

La funzione ***freopen()*** consente di associare un file differente a un flusso già esistente, cambiando anche la modalità di accesso, cosa che viene fatta normalmente per ridirigere i flussi standard. I primi due argomenti della funzione sono gli stessi di ***fopen()***, con l'aggiunta alla fine del puntatore al flusso che si vuole ridirigere. La funzione restituisce il puntatore al flusso ridiretto se l'operazione ha successo, altrimenti produce soltanto il puntatore nullo. Se nel primo argomento, al posto di indicare il nome del file, si mette un puntatore nullo, la chiamata della funzione serve solo per modificare la modalità di accesso a un file già aperto, senza ridirigerne il flusso. Va osservato che il cambiamento della modalità di accesso, in ogni caso, dipende dal sistema operativo e non è detto che si possano applicare tutte le combinazioni.

La funzione ***fclose()*** permette di chiudere il flusso indicato come argomento, restituendo un valore numerico pari a zero se l'operazione ha successo, oppure il valore corrispondente alla macro-variabile ***EOF*** in caso contrario. La chiusura di un flusso implica la scrittura di dati rimasti in sospeso (in una memoria tampone). Un flusso già chiuso non deve essere chiuso nuovamente.

Dal momento che lo standard POSIX introduce il concetto di descrittore di file, per poter associare un flusso di file a un file già aperto come descrittore, si usa la funzione ***fdopen()***, mentre per fare il contrario, si usa la funzione ***fileno()***:

```
FILE *fdopen  (int fdn, const char *io_mode);
int   fileno  (FILE *stream);
```


La funzione ***fdopen()*** richiede l'indicazione del numero del descrittore e della modalità di accesso, la quale deve essere compatibile con quanto già definito a proposito del descrittore stesso. L'associazione tra flusso di file e descrittore comporta inizialmente l'azzeramento dell'indicatore di errore e di quello di fine file; inoltre, se viene chiuso il flusso di file, si ottiene automaticamente la chiusura del descrittore relativo.

La funzione ***fileno()*** restituisce il numero di descrittore associato a un flusso di file già aperto. Se però l'operazione fallisce, restituisce il valore `-1` e aggiorna la variabile ***errno***.

69.17.8 Funzioni per la gestione della memoria tampone

Le funzioni ***setvbuf()*** e ***setbuf()*** consentono di attribuire una memoria tampone (*buffer*) a un certo flusso di dati (un file già aperto), mentre ***fflush()*** consente di richiedere espressamente lo scarico della memoria in modo che le operazioni sospese di scrittura siano portate a termine completamente.

```
int  setvbuf (FILE *restrict stream, char *restrict buffer,
              int buf_mode, size_t size);

void setbuf  (FILE *restrict stream, char *restrict buffer);

int  fflush  (FILE *stream);
```

La funzione ***setvbuf()*** permette di attribuire una memoria tampone a un file che è appena stato aperto e per il quale non è ancora stato eseguito alcun accesso. Il primo argomento della funzione è il puntatore al flusso relativo e il secondo è il puntatore all'inizio dell'array di caratteri da usare come memoria tampone. Se al posto del riferimento alla memoria tampone si indica un puntatore nullo, si intende che la funzione debba allocare automaticamente lo spazio necessario; se invece l'array viene fornito, è evidente che deve rimanere disponibile per tutto il tempo in cui il flusso rimane aperto.

Il terzo argomento atteso dalla funzione ***setvbuf()*** è un numero che esprime la modalità di funzionamento della memoria tampone. Questo numero viene fornito attraverso l'indicazione di una tra le macro-variabili ***_IOFBF***, ***_IOLBF*** e ***_IONBF***. Il quarto argomento indica la dimensione dell'array da usare come memoria tampone: se l'array viene fornito effettivamente, si tratta della dimensione che può essere utilizzata; altrimenti è la dimensione richiesta per l'allocazione automatica.

La funzione ***setvbuf()*** restituisce zero se l'operazione richiesta è eseguita con successo; in caso contrario restituisce un valore differente.

La funzione ***setbuf()*** è una semplificazione di ***setvbuf()*** che non restituisce alcun valore, dove al posto di indicare la modalità di gestione della memoria tampone, si intende implicitamente quella corrispondente alla macro-variabile ***_IOFBF*** (pertanto si tratta di una gestione completa della memoria tampone), mentre al posto di indicare la dimensione dell'array che costituisce la memoria tampone si intende il valore corrispondente alla macro-variabile ***BUFSIZ***. In pratica, è come utilizzare la funzione ***setvbuf()*** così:

```
(void) setvbuf (stream, buffer, _IOFBF, BUFSIZ);
```

La funzione '**`fflush`**' si usa per i file aperti in scrittura, allo scopo di aggiornare i file se ci sono dati sospesi nella memoria tampone che devono ancora essere trasferiti effettivamente. La funzione si attende come argomento il puntatore al flusso per il quale eseguire questo aggiornamento, ma se si fornisce il puntatore nullo (la macro-variabile ***NULL***), si ottiene l'aggiornamento di tutti i file aperti in scrittura. A parte questo, la funzione non altera lo stato del flusso.

La funzione ***fflush()*** restituisce zero se riesce a completare con successo il proprio compito, altrimenti restituisce il valore corrispondente a ***EOF*** e aggiorna la variabile individuata dall'espressione ***errno*** in modo da poter risalire al tipo di errore che si è presentato.

La funzione ***fflush()*** interviene solo nella memoria tampone gestita internamente al programma, ma bisogna tenere presente che il sistema operativo potrebbe gestire un'altra memoria del genere, per il cui scarico occorre eventualmente intervenire con funzioni specifiche del sistema stesso.

69.17.9 Funzioni per la composizione dell'output

Alcune funzioni del file '`stdio.h`' sono realizzate con lo scopo principale di comporre una stringa attraverso l'inserzione di componenti di vario genere, convertendo i dati in modo da poterli rappresentare in forma «tipografica», nel senso di sequenza di caratteri che hanno una rappresentazione grafica.

Queste funzioni hanno in comune una stringa contenente degli ***specificatori di conversione***, caratterizzati dal fatto che iniziano con il simbolo di percentuale ('`%`') e dalla presenza di un elenco indefinito di argomenti, il cui valore viene utilizzato in sostituzione degli specificatori di conversione. Il modo in cui si esprime uno specificatore di conversione può essere complesso, pertanto viene mostrato un modello sintattico che descrive la sua struttura:

`%`[*simbolo*][*n_ampiezza*][`.`*n_precisione*][`hh`|`h`|`l`|`ll`|`j`|`z`|`t`|`L`]*tipo*

La prima cosa da individuare in uno specificatore di conversione è il tipo di argomento che viene interpretato e, di conseguenza, il genere di rappresentazione che se ne vuole produrre. Il tipo viene espresso da una lettera alfabetica, alla fine dello specificatore di conversione. La tabella successiva riepiloga i tipi principali.

Tabella 69.261. Tipi di conversione principali.

Simbolo	Tipo di argomento	Conversione applicata
`%`…`d` `%`…`i`	`int`	Numero intero con segno da rappresentare in base dieci.
`%`…`u`	`unsigned int`	Numero intero senza segno da rappresentare in base dieci.
`%`…`o`	`unsigned int`	Numero intero senza segno da rappresentare in ottale (senza lo zero iniziale che viene usato spesso per caratterizzare un tale tipo di rappresentazione).
`%`…`x` `%`…`X`	`unsigned int`	Numero intero senza segno da rappresentare in esadecimale (senza il prefisso '**`0x`**' o '**`0X`**' che viene usato spesso per caratterizzare un tale tipo di rappresentazione).
`%`…`c`	`int`	Un carattere singolo, dopo la conversione in '**`unsigned char`**'.
`%`…`s`	`char *`	Una stringa.
`%`…`f`	`double`	Un numero a virgola mobile, da rappresentare in notazione decimale fissa: [`-`]*iii*`.`*dddddd*
`%`…`e` `%`…`E`	`double`	Un numero a virgola mobile, da rappresentare in notazione esponenziale: [`-`]*i*`.`*ddddd*`e`±*xx* [`-`]*i*`.`*ddddd*`E`±*xx*
`%`…`g` `%`…`G`	`double`	Un numero a virgola mobile, rappresentato in notazione decimale fissa o in notazione esponenziale, a seconda di quale si presti meglio in base ai vincoli posti da altri componenti dello specificatore di conversione.
`%p`	`void *`	Un puntatore generico rappresentato in qualche modo in forma grafica.

Simbolo	Tipo di argomento	Conversione applicata
`%n`	`int *`	Questo specificatore non esegue alcuna conversione e si limita a memorizzare un valore intero (di tipo '**`int`**') nella variabile a cui punta l'argomento. Per la precisione, viene memorizzata la quantità di caratteri generati fino a quel punto dalla conversione.
`%%`		Questo specificatore si limita a produrre un carattere di percentuale ('`%`') che altrimenti non sarebbe rappresentabile.

Nel modello sintattico che descrive lo specificatore di conversione, si vede che subito dopo il segno di percentuale può apparire un simbolo (*flag*). I simboli principali che possono essere utilizzati sono descritti nella tabella successiva.

Tabella 69.262. Alcuni simboli, o *flag*.

Simbolo	Corrispondenza
`%+`… `%#+`… `%+0`***ampiezza***… `%#+0`***ampiezza***…	Il segno «+» fa sì che i numeri con segno lo mostrino anche se è positivo. Può combinarsi con lo zero e il cancelletto.
`%0`***ampiezza***… `%+0`***ampiezza***… `%#0`***ampiezza***… `%#+0`***ampiezza***…	Lo zero fa sì che siano inseriti degli zeri a sinistra per allineare a destra il valore, nell'ambito dell'ampiezza specificata. Può combinarsi con il segno «+» e il cancelletto.
`%`***ampiezza***… `% `***ampiezza***…	In mancanza di uno zero iniziale, in presenza dell'indicazione dell'ampiezza, il valore viene allineato a destra usando degli spazi. È possibile esprimere esplicitamente l'intenzione di usare gli spazi mettendo proprio uno spazio, ma in generale non è richiesto. Se si mette lo spazio letteralmente, questo non è poi compatibile con lo zero, mentre le combinazioni con gli altri simboli sono ammissibili.
`%-`***ampiezza***… `%-+`***ampiezza***… `%#-`***ampiezza***… `%#-+`***ampiezza***…	Il segno meno, usato quando la conversione prevede l'uso di una quantità fissa di caratteri con un valore che appare di norma allineato a destra, fa sì che il risultato sia allineato a sinistra. Il segno meno si può combinare il segno «+» e il cancelletto.
`%#`…	Il cancelletto richiede una modalità di rappresentazione alternativa, ammesso che questa sia prevista per il tipo di conversione specificato. È compatibili con gli altri simboli, ammesso che il suo utilizzo serva effettivamente per ottenere una rappresentazione alternativa.

Subito prima della lettera che definisce il tipo di conversione, possono apparire una o due lettere che modificano la lunghezza del valore da interpretare (per lunghezza si intende qui la quantità di byte usati per rappresentarlo). Per esempio, '`%`…**`Lf`**' indica che la conversione riguarda un valore di tipo '**`long double`**'. Tra questi specificatori della lunghezza del dato in ingresso ce ne sono alcuni che indicano un rango inferiore a quello di '**`int`**', come per esempio '`%`…**`hhd`**' che si riferisce a un numero intero della dimensione di un '**`signed char`**'; in questi casi occorre comunque considerare che nella trasmissione degli argomenti alle funzioni interviene sempre la promozione a intero, pertanto viene letto il dato della dimensione specificata, ma viene «consumato» il risultato ottenuto dalla pro-

mozione. La tabella successiva riepiloga i modificatori di lunghezza principali.

Tabella 69.263. Alcuni modificatori della lunghezza del dato in ingresso.

Simbolo	Tipo	Simbolo	Tipo
%…hhd %…hhi	signed char	%…hhu %…hho %…hhx | %…hhX	unsigned char
%…hd %…hi	short int	%…hu %…ho %…hx | %…hX	unsigned short int
%…ld %…li	long int	%…lu %…lo %…lx | %…lX	unsigned long int
%…lc	wint_t	%…ls	wchar_t *
%…lld %…lli	long long int	%…llu %…llo %…llx | %…llX	unsigned long long int
%…jd %…ji	intmax_t	%…ju %…jo %…jx | %…jX	uintmax_t
%…zd %…zi	size_t	%…zu %…zo %…zx | %…zX	size_t
%…td %…ti	ptrdiff_t	%…tu %…to %…tx | %…tX	ptrdiff_t
%…Le | %…LE %…Lf | %…LF %…Lg | %…LG	long double		

I modificatori di lunghezza si possono utilizzare anche con il tipo '**%…n**'. In tal caso, si intende che il puntatore sia del tipo specificato dalla lunghezza. Per esempio, '**%tn**' richiede di memorizzare la quantità di byte composta fino a quel punto in una variabile di tipo '**ptrdiff_t**', a cui si accede tramite il puntatore fornito.

Tra il simbolo (*flag*) e il modificatore di lunghezza può apparire un numero che rappresenta l'ampiezza da usare nella trasformazione ed eventualmente la precisione: '***ampiezza*[.*precisione*]**'. Il concetto parte dalla rappresentazione dei valori in virgola mobile, dove l'ampiezza indica la quantità complessiva di caratteri da usare e la precisione indica quanti di quei caratteri usare per il punto decimale e le cifre successive, ma si applica anche alle stringhe.

In generale, per quanto riguarda la rappresentazione di valori numerici, la parte intera viene sempre espressa in modo completo, anche se l'ampiezza indicata è inferiore; ai numeri interi la precisione non si applica; per i numeri in virgola mobile con rappresentazione esponenziale, la precisione riguarda le cifre decimali che precedono l'esponente; per le stringhe la precisione specifica la quantità di caratteri da considerare, troncando il resto.

In un altro capitolo, la tabella 67.26 riporta un elenco di esempi di utilizzo della funzione ***printf()*** dove si può valutare l'effetto dell'indicazione dell'ampiezza e della precisione.

L'ampiezza, o la precisione, o entrambe, potrebbero essere indicate da un asterisco, come per esempio '**%*.*f**'. L'asterisco usato in questo modo indica che il valore corrispondente (ampiezza, precisione o entrambe) viene tratto dagli argomenti come intero ('**int**'). Pertanto, per tornare all'esempio composto come '**%*.*f**', dagli argomenti viene prelevato un intero che rappresenta l'ampiezza, un altro intero che rappresenta la precisione, quindi si preleva un valore '**double**' che è quanto va rappresentato secondo l'ampiezza e la precisione richieste.

69.17.9.1 Funzioni che ricevono gli argomenti direttamente

Un gruppo di funzioni per la composizione dell'output riceve direttamente gli argomenti variabili che servono agli specificatori di conversione:

```
int sprintf  (char *restrict s, const char *restrict format,
              ...);
int snprintf (char *restrict s, size_t n,
              const char *restrict format, ...);
int fprintf  (FILE *restrict stream,
              const char *restrict format, ...);
int printf   (const char *restrict format, ...);
```

Tutte le funzioni di questo gruppo hanno in comune la stringa di composizione, costituita dal parametro ***format***, e gli argomenti successivi che sono in quantità e qualità indeterminata, in quanto per la loro interpretazione contano gli specificatori di conversione inseriti nella stringa di composizione. Inoltre, tutte queste funzioni restituiscono la quantità di caratteri prodotti dall'elaborazione della stringa di composizione. Va osservato che il conteggio riguarda solo i caratteri e non include, eventualmente, il carattere nullo di terminazione di stringa che viene usato per le funzioni ***sprintf()*** e ***snprintf()***. Se durante il procedimento di composizione si verifica un errore, queste funzioni possono restituire un valore negativo.

La funzione ***sprintf()*** produce il risultato della composizione memorizzandolo a partire dal puntatore indicato come primo parametro (***s***) e aggiungendo il carattere nullo di terminazione. La funzione ***snprintf()***, invece, produce al massimo ***n***−1 caratteri, aggiungendo sempre il carattere nullo di terminazione.

La funzione ***fprintf()*** scrive il risultato della composizione attraverso il flusso di file ***stream***, mentre ***printf()*** lo scrive attraverso lo standard output.

69.17.9.2 Funzioni che ricevono gli argomenti da un'altra funzione

A fianco delle funzioni descritte nella sezione precedente, un gruppo analogo svolge le stesse operazioni, ma ricevendo gli argomenti variabili per riferimento. In pratica si tratta di ciò che serve quando gli argomenti variabili sono stati ottenuti da un'altra funzione e non da una chiamata diretta.

```
int vsprintf  (char *restrict s,
               const char *restrict format, va_list arg);
int vsnprintf (char *restrict s, size_t n,
               const char *restrict format, va_list arg);
int vfprintf  (FILE *restrict stream,
               const char *restrict format, va_list arg);
int vprintf   (const char *restrict format, va_list arg);
```

Il funzionamento è conforme a quello delle funzioni che non hanno la lettera '**v**' iniziale; per esempio, ***vsprintf()*** si comporta conformemente a ***sprintf()***. Per comprendere la differenza si potrebbe dimostrare la realizzazione ipotetica della funzione ***printf()*** avvalendosi di ***vprintf()***:

```
int printf   (const char *restrict format, ...)
{
  va_list arg;
  va_start (arg, format);
  int count;
  count = vprintf (format, arg);
  va_end (arg);
  return count;
}
```

69.17.10 Funzioni per l'interpretazione dell'input

«

Un piccolo gruppo di funzioni del file '`stdio.h`' è specializzato nell'interpretazione di una stringa, dalla quale si vanno a estrapolare dei componenti da collocare in variabili di tipo opportuno. In altri termini, da una stringa che rappresenta un valore espresso attraverso caratteri grafici, si vuole estrarre il valore e assegnare a una certa variabile.

Il meccanismo è opposto a quello usato dalle funzioni del tipo '…**`printf()`**' e anche in questo caso si parte da una stringa contenente principalmente degli specificatori di conversione, seguita da un numero indefinito di argomenti. Gli specificatori delle funzioni che interpretano l'input sono simili a quelli usati per la composizione dell'output, ma non possono essere equivalenti in tutto. Sinteticamente si possono descrivere così:

`%`**`[`**`*`**`]`**`[`***n_ampiezza***`]`**`[``hh`**`|`**`h`**`|`**`l`**`|`**`ll`**`|`**`j`**`|`**`z`**`|`**`t`**`|`**`L`**`]`***tipo***

Come si può vedere, all'inizio può apparire un asterisco, il cui scopo è quello di annullare l'assegnamento del valore a una variabile. In pratica, con l'asterisco il dato corrispondente allo specificatore viene interpretato, ma poi non viene salvato.

Successivamente può apparire un numero che rappresenta l'ampiezza del dato da interpretare, in byte, il cui scopo è quello di limitare la lettura fino a un certo carattere (inteso come '**`char`**', pertanto le sequenze multibyte contano per più di una unità singola).

Dopo può apparire una sigla, composta da una o più lettere, il cui scopo è quello di modificare la dimensione predefinita della variabile di destinazione. In altri termini, senza questo modificatore si intende che la variabile ricevente debba essere di una certa grandezza, ma con l'aggiunta del «modificatore di lunghezza» si precisa invece qualcosa di diverso. In pratica, il modificatore di lunghezza usato da queste funzioni è equivalente a quello delle funzioni di composizione dell'output.

Al termine dello specificatore di conversione appare una lettera che dichiara come deve essere interpretato il dato in ingresso e, in mancanza del modificatore di lunghezza, indica anche la dimensione della variabile ricevente.

Tabella 69.267. Tipi di conversione principali.

Simbolo	Tipo di argomento	Conversione applicata
`%…d`	`int *`	Numero intero con segno rappresentato in base dieci.
`%…i`	`int *`	Numero intero con segno rappresentare in base dieci o in base otto, avendo come prefisso uno zero, oppure in base sedici, avendo come prefisso '**`0x`**' o '**`0X`**'.
`%…u`	`unsigned int *`	Numero intero senza segno rappresentato in base dieci.
`%…o`	`unsigned int *`	Numero intero senza segno rappresentato in ottale (con o senza lo zero iniziale).
`%…x`	`unsigned int *`	Numero intero senza segno rappresentato in esadecimale (con o senza il prefisso '**`0x`**' o '**`0X`**').

Simbolo	Tipo di argomento	Conversione applicata
`%…c`	`char *`	Interpreta un solo carattere, o più caratteri se si specifica l'ampiezza. Nella lettura contano anche gli spazi o qualunque altro carattere e non viene aggiunto il carattere nullo di terminazione.
`%…s`	`char *`	Interpreta una sequenza di caratteri che non siano spazi, aggiungendo alla fine il carattere nullo di terminazione.

`%…a`
`%…e`
`%…f`
`%…g`
	`double *`	Un numero a virgola mobile rappresentato in notazione decimale fissa o in notazione esponenziale:
		`[``-`**`]`***iii . dddddd***
		`[``-`**`]`***i . dddddd*`e`$\pm$***xx***
		`[``-`**`]`***i . dddddd*`E`$\pm$***xx***

`%p`	`void *`	Interpreta il valore di un puntatore che sia rappresentato nello stesso modo in cui farebbe la funzione '**`printf("%p",`** ***puntatore*****`)`**'.
`%n`	`int *`	Questo specificatore non esegue alcuna conversione e si limita a memorizzare la quantità di caratteri ('**`char`**') letti fino a quel punto.
`%…[…]`	`char *`	Interpreta una stringa non vuota contenente solo i caratteri elencati tra parentesi quadre, aggiungendo alla fine il carattere nullo di terminazione. Se tra i caratteri si cerca anche la parentesi quadra chiusa, questa va messa all'inizio dell'elenco: '`%…`**`[]`**`…]`'.
`%…[^…]`	`char *`	Interpreta una stringa non vuota contenente solo caratteri diversi da quelli elencati tra parentesi quadre, aggiungendo alla fine il carattere nullo di terminazione. Se tra i caratteri da escludere si vuole indicare anche la parentesi quadra chiusa, questa va messa all'inizio dell'elenco: '`%…`**`[^]`**`…]`'.
`%%`		Interpreta un carattere di percentuale tra i dati in ingresso, ma senza memorizzare alcunché.

Tabella 69.268. Alcuni modificatori della lunghezza del dato in uscita.

Simbolo	Tipo	Simbolo	Tipo

`%…hhd`
`%…hhi`
	`signed char *`	`%…hhu`
		`%…hho`
		`%…hhx`
		`%…hhn`
			`unsigned char *`

`%…hd`
`%…hi`
	`short int *`	`%…hu`
		`%…ho`
		`%…hx`
		`%…hn`
			`unsigned short int *`

`%…ld`
`%…li`
	`long int *`	`%…lu`
		`%…lo`
		`%…lx`
		`%…ln`
			`unsigned long int *`

Simbolo	Tipo	Simbolo	Tipo
		%…lc %…ls %…lc %…l[…]	wchar_t *
%…lld %…lli	long long int *	%…llu %…llo %…llx %…lln	unsigned long long int *
%…jd %…ji	intmax_t *	%…ju %…jo %…jx %…jn	uintmax_t *
%…zd %…zi	size_t *	%…zu %…zo %…zx %…zn	size_t *
%…td %…ti	ptrdiff_t *	%…tu %…to %…tx %…tn	ptrdiff_t *
%…Le %…Lf %…Lg	long double *		

A proposito dell'interpretazione di caratteri e di stringhe, va precisato cosa accade quando si usa il modificatore '**l**' (elle). Se nello specificatore di conversione appare un valore numerico che esprime un'ampiezza, questa indica una quantità di caratteri, in ingresso, da intendersi come byte. Utilizzando gli specificatori '**%…lc**' e '**%…ls**', la quantità di questi caratteri continua a riferirsi a byte, ma si interpretano le sequenze multibyte in ingresso per generare caratteri di tipo '**wchar_t**'.

Il documento che descrive lo standard del linguaggio afferma che la stringa di conversione è composta da direttive, ognuna delle quali è formata da: uno o più spazi (spazi veri e propri o caratteri di tabulazione orizzontale); un carattere multibyte diverso da '**%**' e diverso dai caratteri che rappresentano spazi, oppure uno specificatore di conversione.

[*spazi*] *carattere_multibyte* | %…

Dalla sequenza multibyte che costituisce i dati in ingresso da interpretare, vengono eliminati automaticamente gli spazi iniziali e finali (tutto ciò che si può considerare spazio, anche il codice di interruzione di riga), quando all'inizio o alla fine non ci sono corrispondenze con specificatori di conversione che possono interpretarli.

Quando la direttiva di interpretazione inizia con uno o più spazi orizzontali, significa che si vogliono ignorare gli spazi a partire dalla posizione corrente nella lettura dei dati in ingresso; inoltre, la presenza di un carattere che non fa parte di uno specificatore di conversione indica che quello stesso carattere deve essere incontrato nell'interpretazione dei dati in ingresso, altrimenti il procedimento

di lettura e valutazione si deve interrompere. Se due specificatori di conversione appaiono adiacenti, i dati in ingresso corrispondenti possono essere separati da spazi orizzontali o da spazi verticali (il codice di interruzione di riga).

Purtroppo, la sintassi per la scrittura delle stringhe di conversione non è molto soddisfacente e diventa difficile spiegarne il comportamento, a meno di rimanere fermi su esempi molto semplici.

69.17.10.1 Funzioni che ricevono gli argomenti direttamente

Un gruppo di funzioni per l'interpretazione dell'input riceve direttamente gli argomenti variabili che servono agli specificatori di conversione:

```
int fscanf (FILE *restrict stream,
            const char *restrict format, ...);
int sscanf (const char *restrict s,
            const char *restrict format, ...);
int scanf  (const char *restrict format, ...);
```

Tutte le funzioni di questo gruppo hanno in comune la stringa di conversione, costituita dal parametro ***format***, e gli argomenti successivi che sono puntatori di tipo indeterminato, in quanto per la loro interpretazione contano gli specificatori di conversione inseriti nella stringa. Inoltre, tutte queste funzioni restituiscono la quantità di valori assegnati alle variabili rispettive, oppure il valore corrispondente alla macro-variabile ***EOF*** nel caso si verifichi un errore prima di qualunque conversione.

La funzione ***sscanf()*** scandisce il contenuto della stringa indicata come primo parametro (**s**); la funzione ***fscanf()*** scandisce l'input proveniente dal flusso di file indicato come primo argomento (***stream***), mentre la funzione ***scanf()*** scandisce direttamente lo standard input.

69.17.10.2 Funzioni che ricevono gli argomenti da un'altra funzione

A fianco delle funzioni descritte nella sezione precedente, un gruppo analogo svolge le stesse operazioni, ma ricevendo gli argomenti variabili per riferimento. In pratica si tratta di ciò che serve quando gli argomenti variabili sono stati ottenuti da un'altra funzione e non da una chiamata diretta.

```
int vfscanf (FILE *restrict stream,
             const char *restrict format,
             va_list arg);
int vsscanf (const char *restrict s,
             const char *restrict format,
             va_list arg);
int vscanf  (const char *restrict format,
             va_list arg);
```

Il funzionamento è conforme a quello delle funzioni che non hanno la lettera '**v**' iniziale; per esempio, ***vscanf()*** si comporta conformemente a ***scanf()***. Per comprendere la differenza si potrebbe dimostrare la realizzazione ipotetica della funzione ***scanf()*** avvalendosi di ***vscanf()***:

```
int scanf (const char *restrict format, ...);
{
  va_list arg;
  va_start (arg, format);
  int count;
  count = vscanf (format, arg);
  va_end (arg);
  return count;
}
```

69.17.11 Funzioni per la lettura e la scrittura di un carattere alla volta

Le funzioni ***fgetc()*** e ***getc()*** leggono un carattere (*char*) attraverso il flusso di file indicato come argomento:

```
int fgetc (FILE *stream);

#define getc(STREAM) (fgetc (STREAM))
```

Lo standard prescrive che la funzione ***getc()*** sia in realtà una macroistruzione, così come si ipotizza nella dichiarazione appena mostrata. A questo proposito occorre tenere presente che, se si usa ***getc()***, l'espressione usata per individuare il flusso di file potrebbe essere valutata più di una volta.

Il carattere letto da ***fgetc()*** viene interpretato senza segno e trasformato in un intero (pertanto deve risultare essere di segno positivo). Se viene tentata la lettura oltre la fine del file, la funzione restituisce il valore rappresentato da ***EOF*** e memorizza questa condizione nella variabile strutturata che rappresenta il flusso di file. Se invece si verifica un errore di lettura, viene impostato il contenuto dell'indicatore di errore relativo al flusso di file e la funzione restituisce sempre il valore ***EOF***.

Secondo lo standard, la funzione ***getchar()*** è equivalente a '`getc (stdin)`', senza specificare altro. Ciò può significare ragionevolmente che se ***getc()*** è una macroistruzione, anche ***getchar()*** dovrebbe esserlo, altrimenti potrebbe trattarsi di una funzione vera e propria:

```
#define getchar (getc (stdin))
```

La funzione ***ungetc()*** ha lo scopo di annullare l'effetto della lettura dell'ultimo carattere, ma il modo in cui viene gestita la cosa rende la questione molto delicata:

```
int ungetc (int c, FILE *stream);
```

Semplificando il problema, la funzione ***ungetc()*** rimanda indietro il carattere ***c*** nel flusso di file ***stream*** dal quale è appena stata eseguita una lettura. Tuttavia, non è garantito che il carattere in questione sia effettivamente quello che è stato letto per ultimo, ma la fase successiva di lettura deve fornire per primo tale carattere.

Si comprende intuitivamente che, se si eseguono operazioni di spostamento della posizione corrente relativa al flusso di file in questione, il carattere rimandato indietro con la funzione ***ungetc()*** debba essere dimenticato, soprattutto se questo non corrispondeva a quello che effettivamente era stato letto per ultimo in quel momento.

L'uso della funzione ***ungetc()*** implica un aggiornamento della posizione corrente relativa al flusso di file, ma questa modifica, in presenza di file di testo che non siano realizzati secondo lo standard tradizionale dei sistemi Unix, implica che l'entità di questa modifica non possa essere predeterminabile.[11]

La funzione ***ungetc()*** può fallire nel suo intento e lo standard prescrive che sia «garantita» la possibilità di rimandare indietro almeno un carattere. Se la funzione riesce a eseguire l'operazione, restituisce il valore positivo corrispondente al carattere rinviato; altrimenti restituisce il valore della macro-variabile ***EOF***.

Le funzioni ***fputc()***, ***putc()*** e ***putchar()*** eseguono l'operazione inversa, rispettivamente, di ***fgetc()***, ***getc()*** e ***getchar()***; anche in questo caso vale il fatto che ***putc()*** possa essere realizzata come macroistruzione:

```
int fputc (int c, FILE *stream);

#define putc(CHAR, STREAM) (fputc ((CHAR), (STREAM)))

#define putchar(CHAR) (putc ((CHAR), stdout))
```

La funzione ***fputc()*** scrive un carattere (fornito come numero intero positivo) attraverso il flusso di file indicato; ***putc()*** fa lo stesso, ma potrebbe essere una macroistruzione; ***putchar()*** scrive attraverso lo standard output.

Se la scrittura fallisce, le funzioni (o le macroistruzioni) restituiscono il valore ***EOF***; diversamente restituiscono il valore positivo corrispondente al carattere scritto.

Per come sono state proposte queste funzioni, non c'è differenza nell'uso di ***getc()*** al posto di ***fgetc()***, così come tra ***putc()*** e ***fputc()***. Evidentemente, se la propria libreria può esprimere le macroistruzioni ***getc()*** e ***putc()*** richiamando funzioni del sistema operativo (funzioni che dovrebbero essere richiamate anche da ***fgetc()*** e ***fputc()***), si può risparmiare un livello di chiamate per accelerare leggermente l'esecuzione del programma.

69.17.12 Funzioni per l'input e l'output di file di testo

Le funzioni ***fgets()*** e ***fputs()*** sono utili per l'accesso a file di testo, quando si vuole indicare il flusso di file:

```
char *fgets (char *restrict s, int n,
             FILE *restrict stream);

int   fputs (const char *restrict s, FILE *restrict stream);
```

La funzione ***fgets()*** legge al massimo $n-1$ caratteri (nel senso di elementi '`char`') attraverso il flusso di file ***stream***, copiandoli in memoria a partire dall'indirizzo ***s*** e aggiungendo alla fine il carattere nullo di terminazione delle stringhe. La lettura si esaurisce prima di $n-1$ caratteri se viene incontrato il codice di interruzione di riga, il quale viene rappresentato nella stringa a cui punta ***s***, ovvero se si raggiunge la fine del file. In ogni caso, la stringa ***s*** viene terminata correttamente con il carattere nullo.

La funzione ***fgets()*** restituisce la stringa ***s*** se la lettura avviene con successo, ovvero se ha prodotto almeno un carattere; altrimenti, il contenuto dell'array a cui punta ***s*** non viene modificato e la funzione restituisce il puntatore nullo. Se si creano errori imprevisti, la funzione potrebbe restituire il puntatore nullo, ma senza garantire che l'array ***s*** sia rimasto intatto.

La funzione ***fputs()*** serve a copiare la stringa a cui punta ***s*** nel file rappresentato dal flusso di file ***stream***. La copia della stringa avviene escludendo però il carattere nullo di terminazione. Va osservato che questa funzione, pur essendo contrapposta evidentemente a ***fgets()***, **non conclude la riga** del file, ovvero, non aggiunge il codice di interruzione di riga. Per ottenere la conclusione della riga di un file di testo, occorre inserire nella stringa, espressamente, il carattere '**\n**'.

La funzione ***fputs()*** restituisce il valore rappresentato da ***EOF*** se l'operazione di scrittura produce un errore; altrimenti restituisce un valore positivo qualunque.

Le funzioni ***gets()*** e ***puts()*** sono utili per l'accesso a file di testo, quando si vogliono utilizzare i flussi standard. In linea di massima, assomigliano a ***fgets()*** e ***fputs()***, ma il funzionamento non è perfettamente conforme a quelle:

```
char *gets (char *s);

int   puts (const char *s);
```

Il funzionamento di ***gets()*** è perfettamente conforme a quello di ***fgets()***, con la sola differenza che il flusso di file da cui si leggono i caratteri è lo standard input. Nel caso di ***puts()***, a parte il fatto che si usa lo standard output per la scrittura, occorre sottolineare che alla fine della stringa **viene accodata la scrittura del codice di interruzione di riga**.

69.17.13 Funzioni per l'input e output diretto

Le funzioni ***fread()*** e ***fwrite()*** consentono di leggere e scrivere attraverso un flusso di file aperto, il quale deve essere specificato espressamente tra gli argomenti. Lo standard prescrive che queste funzioni si avvalgano rispettivamente di ***fgetc()*** e di ***fputc()***.

```
size_t fread  (void *restrict ptr,
               size_t size,
               size_t nmemb,
               FILE *restrict stream);

size_t fwrite (const void *restrict ptr,
               size_t size,
               size_t nmemb,
               FILE *restrict stream);
```

Le due funzioni (***fread()*** e ***fwrite()***) hanno praticamente gli stessi argomenti, usati in modo analogo. La lettura e la scrittura avviene a blocchi da ***size*** byte, ripetuta per ***nmemb*** volte, attraverso il flusso di file specificato come ***stream***. La lettura implica la memorizzazione dei caratteri in forma di elementi '**`unsigned char`**', a partire dall'indirizzo indicato dal puntatore ***ptr***; la scrittura copia nello stesso modo i caratteri a partire dal puntatore ***ptr***, verso il flusso di file.

L'aggiornamento della posizione corrente interna al file a cui si riferisce il flusso avviene esattamente come per le funzioni ***fgetc()*** e ***fputc()***.

Il valore restituito dalle funzioni ***fread()*** e ***fwrite()*** rappresenta la quantità di blocchi, ovvero la quantità di elementi ***nmemb*** che sono stati copiati con successo. Pertanto, se si ottiene un valore inferiore a ***nmemb***, significa che l'operazione è stata interrotta a causa di un errore.

69.17.14 Funzioni per il posizionamento

Sono previste diverse funzioni per modificare la posizione corrente dei flussi di file. Le funzioni più semplici per iniziare sono ***fseek()***, ***ftell()*** e ***rewind()***:

```
int      fseek  (FILE *stream, long int offset, int whence);
long int ftell  (FILE *stream);
void     rewind (FILE *stream);
```

Lo standard POSIX prevede anche le funzioni ***fseeko()*** e ***ftello()***, equivalenti alle funzioni ***fseek()*** e ***ftell()*** dello standard C, con la differenza che lo scostamento, fornito come argomento o restituito dalla funzione, è di tipo '**`off_t`**', al posto di essere di tipo '**`long int`**':

```
int      fseeko (FILE *stream, off_t offset, int whence);
off_t    ftello (FILE *stream);
```

In generale, le funzioni ***fseek()*** e ***fseeko()*** spostano la posizione corrente relativa al flusso di file ***stream***, nella nuova posizione determinata dai parametri ***whence*** e ***offset***. Il parametro ***whence*** viene fornito attraverso una macro-variabile che può essere ***SEEK_SET***, ***SEEK_CUR*** o ***SEEK_END***, indicando rispettivamente l'inizio del file, la posizione corrente o la fine del file. Dalla posizione indicata dal parametro ***whence*** viene aggiunta, algebricamente, la quantità di byte indicata dal parametro ***offset***.

Quanto descritto a proposito del posizionamento con le funzioni ***fseek()*** e ***fseeko()*** riguarda i file che vengono gestiti in modo binario, perché con i file di testo è opportuno avere maggiore accortezza: il valore del parametro ***offset*** deve essere zero, oppure quanto restituito in precedenza dalle funzioni ***ftell()*** o ***ftello()*** per lo stesso flusso di file, ma in tal caso, ovviamente, il parametro ***whence*** deve corrispondere a ***SEEK_SET***.

Le funzioni ***fseek()*** e ***fseeko()*** restituiscono zero se possono eseguire l'operazione, altrimenti danno un risultato diverso; nel caso di ***fseeko()***, quando si presenta un errore, il risultato restituito è precisamente `-1`, e in più viene aggiornata anche la variabile ***errno***.

Le funzioni ***ftell()*** e ***ftello()*** restituiscono la posizione corrente del flusso di file indicato come argomento. Questo valore può essere usato con ***fseek()*** o ***fseeko()*** rispettivamente, al posto dello scostamento (il parametro ***offset***), indicando come posizione di riferimento l'inizio del file, ovvero ***SEEK_SET***. Se le funzioni ***ftell()*** e ***ftello()*** non riescono a fornire la posizione, restituiscono il valore `-1` (tradotto rispettivamente in '**`long int`**' e '**`off_t`**') e annotano il fatto nella variabile ***errno***.

La funzione ***rewind()*** si limita a riposizionare il flusso di file all'inizio. In pratica è come utilizzare la funzione ***fseek()*** specificando uno scostamento pari a zero a partire da ***SEEK_SET***, ignorando il valore restituito:

```
(void) fseek (stream, 0L, SEEK_SET)
```

Va osservato che il riposizionamento di un flusso di file implica l'azzeramento dell'indicatore di fine file, se questo risulta impostato, e la cancellazione dei caratteri che eventualmente fossero stati rimandati indietro con la funzione ***ungetc()***.

Le funzioni ***fseek()*** e ***ftell()*** sono utili particolarmente per i file binari ed eventualmente per i file di testo con una rappresentazione dei caratteri tradizionale. Ma quando il file di testo contiene anche caratteri espressi attraverso sequenze multibyte, il posizionamento al suo interno dovrebbe tenere anche conto del progresso nell'interpretazione di queste sequenze. Pertanto, esistono altre due funzioni per leggere la posizione e ripristinarla in un secondo momento:

```
int fgetpos (FILE *restrict stream, fpos_t *restrict pos);
int fsetpos (FILE *stream, const fpos_t *pos);
```

Entrambe le funzioni che appaiono nei due prototipi restituiscono zero se l'operazione è stata compiuta con successo, altrimenti restituiscono un valore differente. Nel caso particolare di ***fsetpos()***, se si verifica un errore, questo viene annotato nella variabile ***errno***.

Le due funzioni richiedono come primo argomento il flusso di file a cui ci si riferisce; come secondo argomento richiedono il puntatore a una variabile di tipo '**`fpos_t`**'. La funzione ***fgetpos()*** memorizza nella variabile a cui punta il parametro ***pos*** le informazioni sulla posizione corrente del file, assieme allo stato di interpretazione relativo alle sequenze multibyte; la funzione ***fsetpos()***, per converso, utilizza la variabile a cui punta ***pos*** per ripristinare la posizione memorizzata, assieme allo stato di avanzamento dell'interpretazione di una sequenza multibyte.

69.17.15 Accesso esclusivo ai flussi di file

Lo standard POSIX, introducendo la gestione dei thread multipli, inserisce nel file '`stdio.h`' alcune funzioni per controllare l'accesso esclusivo ai flussi di file. Va tenuto a mente che si tratta di un controllo che riguarda esclusivamente il processo elaborativo in corso, e non l'accesso ai file da parte di processi differenti.

```
void flockfile   (FILE *stream);
int  trylockfile (FILE *stream);
void funlockfile (FILE *stream);
```

La funzione ***flockfile()*** cerca di ottenere un accesso esclusivo a un file, individuato da un puntatore che rappresenta il flusso di file relativo. Se il file risulta già impegnato, rimane in attesa, fino a quando si libera. La funzione ***trylockfile()***, invece, non rimane in attesa e restituisce l'esito della sua azione: zero se è riuscita a ottenere l'accesso esclusivo, un numero diverso se invece non c'è riuscita. Quando poi un thread che aveva ottenuto l'accesso esclusivo a un flusso di file, non ne ha più bisogno, lo libera con la funzione ***funlockfile()***.

Le funzioni ***getc()***, ***putc()***, ***getchar()*** e ***putchar()***, quando sono presenti le estensioni per la gestione dei thread multipli, e di conseguenza anche per l'accesso esclusivo ai flussi di file, si comportano rispettando tali vincoli. Eventualmente, se per qualche ragione si vogliono usare queste funzionalità, ignorando espressamente tali vincoli, sono disponibili funzioni equivalenti, il cui nome termina per '**`_unlocked`**':

```
int getc_unlocked (FILE *stream);
int putc_unlocked (int c, FILE *stream);
int getchar_unlocked (void);
int putchar_unlocked (int c);
```

Lo standard POSIX prescrive comunque che queste siano usate solo da thread che hanno già ottenuto un accesso esclusivo al flusso relativo, in modo da non compromettere la gestione controllata di tali accessi. Pertanto, in tal modo queste funzioni consentono sem-

plicemente un'esecuzione più rapida, dal momento che non vengono eseguiti tutti i controlli necessari.

69.17.16 Condotti

Lo standard POSIX introduce il concetto di «condotto», ovvero di *pipe*, attraverso il quale è possibile inviare lo standard output di un processo elaborativo, verso lo standard input di un altro. Per attuare questo meccanismo, è necessario che un processo sia in grado di avviare un altro processo, attraverso un comando da dare alla shell, e da questo processo viene poi letto lo standard output o scritto lo standard input.

Dal punto di vista del processo che crea il condotto, il processo secondario avviato è trattato come se fosse un file, dove però si può solo leggere o scrivere, ma non si possono fare entrambe le cose.

```
FILE *popen  (const char *command, const char *type);
int   pclose (FILE *stream);
```

Il condotto viene aperto con la funzione ***popen()***, la quale assomiglia a ***fopen()***, ma invece dell'indicazione del percorso del file da aprire, richiede il comando da eseguire attraverso la shell '`/bin/sh`' (nota come la shell POSIX). Per quanto riguarda il tipo di accesso, va osservato che può trattarsi soltanto di lettura o scrittura, pertanto si può scegliere solo tra la stringa '`"r"`' o '`"w"`'.

La lettura o la scrittura in un flusso di file associato a un condotto avviene nel modo consueto, ma la chiusura richiede l'uso della funzione ***pclose()***.

69.17.17 Informazioni sul terminale

Lo standard POSIX prevede che il sistema operativo abbia una gestione dei file di dispositivo per rappresentare i vari componenti fisici dell'elaboratore. Nel file '`stdio.h`' inserisce la funzione ***ctermid()***, con lo scopo di conoscere il percorso del file di dispositivo del terminale associato al processo elaborativo.

```
char *ctermid (char *s);
```

La funzione si aspetta di ricevere come argomento il puntatore a una stringa modificabile, in cui scrivere il percorso del terminale. Se però viene fornito un puntatore nullo, l'informazione viene annotata in un'area di memoria che può essere statica (e quindi riutilizzata a ogni chiamata della funzione). Il percorso del terminale, annotato dalla funzione, può utilizzare al massimo la quantità di caratteri definita dalla macro-variabile ***L_ctermid***; pertanto, se si definisce un array di caratteri da usare per tale annotazione, deve essere almeno quella dimensione.

La funzione restituisce sempre il puntatore a una stringa, che può essere nulla se non è possibile determinare il terminale. Pertanto la funzione non prevede l'indicazione di errori.

69.17.18 Gestione degli errori

Un gruppo di funzioni di '`stdio.h`' consente di verificare ed eventualmente azzerare lo stato degli indicatori di errore riferiti a un certo flusso di file:

```
void clearerr (FILE *stream);
int  feof     (FILE *stream);
int  ferror   (FILE *stream);
void perror   (const char *s);
```

La funzione ***clearerr()*** azzera gli indicatori di errore e di fine file per il flusso di file indicato come argomento, senza restituire alcunché.

La funzione ***feof()*** controlla lo stato dell'indicatore di fine file per il flusso di file indicato. Se questo non è attivo restituisce zero, altrimenti restituisce un valore diverso da zero.

La funzione ***ferror()*** controlla lo stato dell'indicatore di errore per il flusso di file indicato. Se questo non è attivo restituisce zero, altrimenti restituisce un valore diverso da zero.

La funzione ***perror()*** prende in considerazione la variabile ***errno*** e cerca di tradurla in un messaggio testuale da emettere attraverso lo

standard error (con tanto di terminazione della riga, in modo da riposizionare a capo il cursore). Se il parametro ***s*** corrisponde a una stringa non vuota, il testo di questa viene posto anteriormente al messaggio, separandolo con due punti e uno spazio ('`: `'). Il contenuto del messaggio è lo stesso che si otterrebbe con la funzione ***strerror()***, fornendo come argomento la variabile ***errno***.

69.17.19 Realizzazione di «vsnprintf()» e altre collegate

Le funzioni per la composizione dell'output che possono essere realizzate senza avere definito la gestione dei file, sono quelle che si limitano a produrre una stringa. La funzione che va realizzata per prima è ***vsnprintf()***, in quanto ***snprintf()*** si può limitare a richiamarla. Naturalmente, anche ***vsprintf()*** e ***sprintf()*** possono avvalersi della stessa ***vsnprintf()***, ponendo un limite massimo abbastanza grande alla stringa da generare. Nella sezione 95.18.42 è disponibile un esempio di realizzazione parziale di ***vsnprintf()***. L'esempio seguente mostra come si ottiene ***snprintf()***, una volta che è disponibile ***vsnprintf()***:

```
#include <stdio.h>
int
snprintf (char *restrict string, size_t n,
          const char *restrict format, ...)
{
  va_list ap;
  va_start (ap, format);
  return vsnprintf (string, n, format, ap);
}
```

Eventualmente, per realizzare le funzioni ***vsprintf()*** e ***sprintf()***, secondo le limitazioni già descritte, sono sufficienti due macroistruzioni:

```
#define vsprintf(s, format, arg) \
                (vsnprintf (s, SIZE_MAX, format, arg))
#define sprintf(s, ...)          \
                (snprintf (s, SIZE_MAX, __VA_ARGS__))
```

69.18 Riferimenti

- Wikipedia, *C standard library*, *http://en.wikipedia.org/wiki/C_standard_library*
- *ISO/IEC 9899:TC2*, *http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf*
- Wikipedia, *assert.h*, *limits.h*, *stdint.h*, *errno.h*, *locale.h*, *ctype.h*, *stdarg.h*, *stdlib.h*, *inttypes.h*, *iso646.h*, *stdbool.h*, *stddef.h*, *string.h*, *signal.h*, *time.h*, *stdio.h*

  http://en.wikipedia.org/wiki/Assert.h , *http://en.wikipedia.org/wiki/Limits.h* , *http://en.wikipedia.org/wiki/Stdint.h* , *http://en.wikipedia.org/wiki/Errno.h* , *http://en.wikipedia.org/wiki/Locale.h* , *http://en.wikipedia.org/wiki/Ctype.h* , *http://en.wikipedia.org/wiki/Stdarg.h* , *http://en.wikipedia.org/wiki/Stdlib.h* , *http://en.wikipedia.org/wiki/Inttypes.h* , *http://en.wikipedia.org/wiki/Iso646.h* , *http://en.wikipedia.org/wiki/Stdbool.h* , *http://en.wikipedia.org/wiki/Stddef.h* , *http://en.wikipedia.org/wiki/String.h* , *http://en.wikipedia.org/wiki/Signal.h* , *http://en.wikipedia.org/wiki/Time.h* , *http://en.wikipedia.org/wiki/Stdio.h*
- The Open Group, *The Single UNIX® Specification, Version 2*, *assert.h*, *limits.h*, *stdint.h*, *errno.h*, *locale.h*, *ctype.h*, *stdarg.h*, *stdlib.h*, *inttypes.h*, *iso646.h*, *stdbool.h*, *stddef.h*, *string.h*, *signal.h*, *time.h*, *stdio.h*

  http://pubs.opengroup.org/onlinepubs/000095399/basedefs/assert.h.html ,
  http://pubs.opengroup.org/onlinepubs/000095399/basedefs/limits.h.html ,
  http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stdint.h.html ,
  http://pubs.opengroup.org/onlinepubs/000095399/basedefs/errno.h.html ,

http://pubs.opengroup.org/onlinepubs/000095399/basedefs/locale.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/ctype.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stdarg.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stdlib.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/inttypes.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/iso646.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stdbool.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stddef.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/string.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/signal.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/time.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stdio.h.html

- Steven Pemberton, *Enquire: Everything you wanted to know about your C Compiler and Machine, but didn't know who to ask*, *http://homepages.cwi.nl/~steven/enquire.html*

[1] Il carattere viene convertito da '**`unsigned char`**' a '**`int`**'.

[2] Si tratta di un puntatore di puntatore, solo perché si deve poter alterare ciò a cui punta, ma questo tipo di valore è, a sua volta, un puntatore.

[3] Il tipo '**`size_t`**', restituito dalle funzioni, è un intero senza segno; pertanto, in condizioni normali, ovvero con una rappresentazione dei valori negativi con il complemento a due, la conversione di `-1` si traduce nel valore massimo rappresentabile.

[4] Lo standard POSIX non estende il contenuto del file '`inttypes.h`'.

[5] Lo standard POSIX non estende il file '`iso646.h`'.

[6] Lo standard POSIX non estende il file '`stdbool.h`'

[7] Lo standard POSIX non estende il file '`stddef.h`'.

[8] Il valore positivo massimo è $(2^{31})-1$, il quale, diviso per la quantità di secondi di un giorno (86400) dà 24855 che, diviso 365, dà circa 68 anni.

[9] Il caso della funzione ***clock()*** e del tipo '**`clock_t`**' è stato considerato a parte.

[10] Si tratta di byte: se il testo copiato è costituito da sequenze multi-byte, i byte sono in quantità maggiore rispetto ai caratteri tipografici che si ottengono.

[11] L'arretramento di un carattere nella posizione corrente di un file di testo non è detto corrisponda alla sottrazione di una unità, perché bisogna tenere in considerazione il modo in cui un file di testo è strutturato nel proprio sistema operativo.