Last update: August 17, 2009

MyLib-C

A Small Library of Basic Utilities in ANSI C

Pierre L'Ecuyer and Richard Simard [1]

Département d'Informatique et de Recherche opérationnelle
Université de Montréal

This document describes a set of basic utilities, implemented in ANSI C, used in the software developed in the author's *simulation laboratory*. Most of these tools were originally implemented in the Modula-2 language. Some of them have been reimplemented in C in order to facilitate the code translation of other software from Modula-2 to C.

[1] Francis Picard and Jean-Sébastien Sénécal have also participated in the development of MyLib.

Contents

gdef

Platform-dependent options are defined here. These options are used by other modules to decide when platform-dependent functions must be commented out or not. Most of these options are set to their true values by the program *configure* in the installation process. The user may choose to set some of them manually. This module also contains a function that prints the current host name.

Global macros

```
#define FALSE 0
#define TRUE 1

#ifndef HAVE_LEBOOL
#define HAVE_LEBOOL
   typedef int  lebool;
#endif
```

Defines the boolean type `lebool`, whose only possible values are `TRUE` and `FALSE`.

```
#ifdef HAVE_STDINT_H
#include <stdint.h>
#endif

#ifndef HAVE_UINT32_T
#if UINT_MAX >= 4294967295UL
   typedef unsigned int  uint32_t;
#else
   typedef unsigned long  uint32_t;
#endif
#endif

#ifndef HAVE_UINT8_T
   typedef unsigned char  uint8_t;
#endif
```

The 8-bit and 32-bit unsigned integers.

```
#define USE_LONGLONG
```

Define this macro if 64-bit integers are available and ensure that they are defined correctly in the following `typedef`. Otherwise, undefine this macro.

```
#ifdef USE_LONGLONG
   typedef long long  longlong;
   typedef unsigned long long  ulonglong;
#define PRIdLEAST64  "lld"
#define PRIuLEAST64  "llu"
#endif
```

The 64-bit integer types. Note that the 64-bit integers types `long long` and `unsigned long`

`long` may exist and be called by different names. The macros `PRIdLEAST64` and `PRIuLEAST64` are defined in the ISO C99 standard in order to print *signed* and *unsigned* 64-bit integers, respectively. Define them correctly if they are not already defined, otherwise comment them out.

`#undef USE_ANSI_CLOCK`

On a MS-Windows platform, the MS-Windows function `GetProcessTimes` will be used to measure the CPU time used by programs (in module `chrono`).

On Linux/Unix platforms, if the macro `USE_ANSI_CLOCK` is defined, the timers will call the ANSI C `clock` function. However, on systems where the type `clock_t` is a 32-bit `long`, the time returned will wrap around to negative values after about 36 minutes. When the macro `USE_ANSI_CLOCK` is undefined, the module `chrono` gets the CPU time used by a program via an alternate non-ANSI C timer based on the POSIX (The Portable Operating System Interface) function `times`, assuming this function is available. The POSIX standard is described in the IEEE Std 1003.1-2001 document (see The Open Group web site at `http://www.opengroup.org/onlinepubs/007904975/toc.htm`).

`#define DIR_SEPARATOR "/"`

Used to separate directories in the pathname of a file. It is `"/"` on Unix-Linux and most other platforms. It may have to be set to `"\\"` on some platforms.

`#undef USE_GMP`

Define this macro if the GNU multi-precision package GMP is available. GMP is a portable library written in C for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. See the Free Software Foundation web site at `http://www.gnu.org/software/gmp/manual`. A few random number generators in library TestU01 use arbitrary large integers, and they have been implemented with GMP functions. If one wants to use GMP, the GMP header file (`gmp.h`) must be in the search path of the C compiler for included files, and the GMP library must be linked to create executable programs.

`#undef HAVE_MATHEMATICA`

Define this macro if the *Mathematica* software [4] and the *MathLink* program that allows a C program to call functions from *Mathematica* are available and you want to use them. This is used only in module `usoft` of library TestU01, where the random number generators from *Mathematica* can be called from a C program for testing with TestU01.

When a C program uses *Mathematica*, it must be compiled with the options `-I$MATHINC -L$MATHLIB -lML`, where `$MATHINC` is the path to the header file `mathlink.h` and `$MATHLIB` is the path to the *MathLink* library `libML.a`. For example, in the environment of our lab, both `$MATHINC` and `$MATHLIB` must be set to

`<dir>/mathematica/5.0/linux/AddOns/MathLink/DeveloperKit/Linux/CompilerAdditions`.

To run a main program named `tulip` on a Unix/Linux platform that calls *Mathematica* functions, one may use

`tulip -linkname 'math -mathlink' -linklaunch`.

Host machine

`void gdef_GetHostName (char machine[], int n);`

Returns in `machine` the host name. Will copy at most n characters, so the array `machine[]` should have a size $\geq n$. This is useful, for example, to get the name of the machine on which a program is running.

`void gdef_WriteHostName (void);`

Prints the name of the machine on which a program is running. This should work on any Unix or Linux machine.

util

Safe functions to open and close files, to allocate dynamic memory, to read/write booleans, and to print error messages. Some of the "functions" are actually implemented as macros, in the interest of speed.

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

Macros

`util_Error (S);`

Prints the string `S`, then stops the program.

`util_Assert (Assertion, S);`

If `lebool Assertion` is `FALSE` (= 0), then prints the string `S` and stops the program.

`util_Warning (Condition, S);`

If `lebool Condition` is `TRUE` ($\neq 0$), then prints the string `S`.

`util_Max (x, y);`

Returns the largest of the two numbers `x`, `y`.

`util_Min (x, y);`

Returns the smallest of the two numbers `x`, `y`.

Prototypes

`FILE * util_Fopen (const char *name, const char *mode);`

Calls `fopen` (from `stdio.h`) with same arguments, but checks for errors. Opens or creates file with name `name` in mode `mode`. Returns a pointer to FILE that is associated with the stream. If `name` cannot be accessed, the program stops.

`int util_Fclose (FILE *stream);`

Calls `fclose` (from `stdio.h`) with same arguments, but checks for errors. Closes the file associated with `stream`. If the file is successfully closed, `0` is returned. If an error occurs or the file was already closed, `EOF` is returned.

`int util_GetLine (FILE *file, char *Line, char c);`

Reads a line of data from `file`. Blank lines and comments are ignored. A comment is any line whose first non-whitespace character is `c`. If the character `c` appears anywhere on a line that is not a comment, then `c` and the rest of the line are ignored too. The function returns -1 if end-of-file or an error is encountered, otherwise it returns 0.

`void util_ReadBool (char S[], lebool *x);`

Reads a `lebool` value from string `S` and returns it in x. The possible values are `TRUE` and `FALSE`.

`void util_WriteBool (lebool x, int d);`

Writes the value of x in a field of width d. If $d < 0$, x is left-justified, otherwise right-justified.

`void * util_Malloc (size_t size);`

Calls `malloc` (from `stdlib.h`) with same arguments, but checks for errors. Allocates memory large enough to hold an object of size `size`. A successful call returns the base address of the allocated space, otherwise the programs stops. The standard type `size_t` is defined in `stdio.h`.

`void * util_Calloc (size_t dim, size_t size);`

Calls `calloc` (from `stdlib.h`) with same arguments, but checks for errors. Allocates memory large enough to hold an array of `dim` objects each of size `size`. A successful call returns the base address of the allocated space, otherwise the programs stops. The standard type `size_t` is defined in `stdio.h`.

`void * util_Realloc (void *ptr, size_t size);`

Calls `realloc` (from `stdlib.h`) with same arguments, but checks for errors. Takes a pointer to a memory region previously allocated and referenced by `ptr`, then changes its size to `size` while preserving its content. A successful call returns the base address of the resized (or new) space, otherwise the programs stops. The standard type `size_t` is defined in `stdio.h`.

`void * util_Free (void *p);`

Calls `free (p)` (from `stdlib.h`) to free memory allocated by `util_Malloc`, `util_Calloc` or `util_Realloc`. Always returns the `NULL` pointer.

bitset

This module defines sets of bits and useful operations for such sets. Some of these operations are implemented as macros.

Constants

`extern unsigned long bitset_maskUL[];`

`bitset_maskUL[j]` has bit j set to 1 and all other bits set to 0. Bit 0 is the least significant bit.

`extern unsigned long bitset_MASK[];`

`bitset_MASK[j]` has all the first j bits set to 1 and all other bits set to 0. Bit 0 is the least significant bit.

Types

`typedef unsigned long bitset_BitSet;`

Set of bits. Bits are numbered starting from 0 for the least significant bit. If bit s is 1, then element s is a member of the set, otherwise not.

Macros

`bitset_SetBit (S, b);`

Sets bit `b` in set `S` to 1.

`bitset_ClearBit (S, b);`

Sets bit `b` in set `S` to 0.

`bitset_FlipBit (S, b);`

Flips bit `b` in set `S`; thus, $0 \rightarrow 1$ and $1 \rightarrow 0$.

`bitset_TestBit (S, b);`

Returns the value of bit `b` in set `S`.

`bitset_RotateLeft (S, t, r);`

Rotates the `t` bits of set `S` by `r` bits to the left. `S` is considered as a `t`-bit number kept in the least significant bits of the equivalent number `S`.

`bitset_RotateRight (S, t, r);`

Rotates the `t` bits of set `S` by `r` bits to the right. `S` is considered as a `t`-bit number kept in the least significant bits of the equivalent number `S`.

Prototypes

`bitset_BitSet bitset_Reverse (bitset_BitSet Z, int s);`

Reverses the `s` least significant bits of `Z` considered as a number. Thus, if `s` = 4 and `Z` = 0011, the returned value is 1100.

`void bitset_WriteSet (char *desc, bitset_BitSet Z, int s);`

Prints the string `desc` (which may be empty), then writes the `s` least significant bits of `Z` considered as an unsigned binary number. This corresponds to the `s` first elements of `Z`.

chrono

This module acts as an interface to the system clock to compute the CPU time used by parts of a program. Even though the ANSI/ISO macro `CLOCKS_PER_SEC = 1000000` is the number of clock ticks per second for the value returned by the `clock` function (so this function returns the number of microseconds), on some systems where the 32-bit type `long` is used to measure time, the value returned by `clock` wraps around to negative values after about 36 minutes. On some other systems where time is measured using the 32-bit type `unsigned long`, the clock may wrap around to 0 after about 72 minutes. When the macro `USE_ANSI_CLOCK` in module `gdef` is undefined, a non-ANSI-C clock is used. On Linux-Unix systems, it calls the POSIX function `times` to get the CPU time used by a program. On a Windows platform (when the macro `HAVE_WINDOWS_H` is defined), the Windows function `GetProcessTimes` will be used to measure the CPU time used by programs.

Every variable of type `chrono_Chrono` acts as an independent *stopwatch*. Several such stopwatchs can run at any given time. An object of type `chrono_Chrono` must be declared for each of them. The function `chrono_Init` resets the stopwatch to zero, `chrono_Val` returns its current reading, and `chrono_Write` writes this reading to the current output. The returned value includes part of the execution time of the functions from module `chrono`. The `chrono_TimeFormat` allows one to choose the kind of time units that are used.

Below is an example of how the functions may be used. A stopwatch named `mytimer` is declared and created. After 2.1 seconds of CPU time have been consumed, the stopwatch is read and reset. Then, after an additional 330 seconds (or 5.5 minutes) of CPU time the stopwatch is read again, printed to the output and deleted.

```
double t;
chrono_Chrono *mytimer = chrono_Create ();
     ⋮     (suppose 2.1 CPU seconds are used here.)
t = chrono_Val (mytimer, chrono_sec);     /* Here, t = 2.1 */
chrono_Init (mytimer);
     ⋮     (suppose 330 CPU seconds are used here.)
t = chrono_Val (mytimer, chrono_min);     /* Here, t = 5.5 */
chrono_Write (mytimer, chrono_hms);     /* Prints:  00:05:30.00 */
chrono_Delete (mytimer);
```

Types

```
typedef struct {
   unsigned long microsec;
   unsigned long second;
   } chrono_Chrono;
```

For every stopwatch needed, the user must declare a variable of this type and initialize it by calling `chrono_Create`.

```
typedef enum {
   chrono_sec,
   chrono_min,
   chrono_hours,
   chrono_days,
   chrono_hms
   } chrono_TimeFormat;
```

Types of units in which the time on a `chrono_Chrono` can be read or printed: in seconds (`sec`)), minutes (`min`), hours (`hour`), days (`days`), or in the `HH:MM:SS.xx` format, with hours, minutes, seconds and hundreths of a second (`hms`).

Timing functions

```
chrono_Chrono * chrono_Create (void);
```

Creates and returns a stopwatch, after initializing it to zero. This function must be called for each new `chrono_Chrono` used. One may reinitializes it later by calling `chrono_Init`.

```
void chrono_Delete (chrono_Chrono * C);
```

Deletes the stopwatch `C`.

```
void chrono_Init (chrono_Chrono * C);
```

Initializes the stopwatch `C` to zero.

```
double chrono_Val (chrono_Chrono * C, chrono_TimeFormat Unit);
```

Returns the time used by the program since the last call to `chrono_Init(C)`. The parameter `Unit` specifies the time unit. Restriction: `Unit = chrono_hms` is not allowed here; it will cause an error.

```
void chrono_Write (chrono_Chrono * C, chrono_TimeFormat Unit);
```

Prints the CPU time used by the program since its last call to `chrono_Init(C)`. The parameter `Unit` specifies the time unit.

num

This module offers some useful constants and basic tools to manipulate numbers represented in different forms.

```
#include "gdef.h"
```

Constants

```
#define num_Pi      3.14159265358979323846
```

The number π.

```
#define num_ebase   2.7182818284590452354
```

The number e.

```
#define num_Rac2    1.41421356237309504880
```

$\sqrt{2}$, the square root of 2.

```
#define num_1Rac2   0.70710678118654752440
```

$1/\sqrt{2}$.

```
#define num_Ln2     0.69314718055994530941
```

$\ln(2)$, the natural logarithm of 2.

```
#define num_1Ln2    1.44269504088896340737
```

$1/\ln(2)$.

```
#define num_MaxIntDouble    9007199254740992.0
```

Largest integer $n_0 = 2^{53}$ such that all integers $n \leq n_0$ are represented exactly as a `double`.

Precomputed powers

```
#define num_MaxTwoExp   64
```

Powers of 2 up to `num_MaxTwoExp` are stored exactly in the array `num_TwoExp`.

```
extern double num_TwoExp[];
```

Contains precomputed powers of 2. One has `num_TwoExp[i]` $= 2^i$ for $0 \leq i \leq$ `num_MaxTwoExp`.

```
#define num_MAXTENNEGPOW   16
```

Negative powers of 10 up to `num_MAXTENNEGPOW` are stored in the array `num_TENNEGPOW`.

`extern double num_TENNEGPOW[];`

Contains the precomputed negative powers of 10. One has `TENNEGPOW[j]`$= 10^{-j}$, for $j = 0, \ldots,$ `num_MAXTENNEGPOW`.

Prototypes

`#define num_Log2(x) (num_1Ln2 * log(x))`

Gives the logarithm of x in base 2.

`long num_RoundL (double x);`

Rounds x to the nearest (`long`) integer and returns it.

`double num_RoundD (double x);`

Rounds x to the nearest (`double`) integer and returns it.

`int num_IsNumber (char S[]);`

Returns `1` if the string `S` begins with a number (with the possibility of spaces and a $+/-$ sign before the number). For example, " + 2" and "4hello" return `1`, while "− + 2" and "hello" return `0`.

`void num_IntToStrBase (long k, long b, char S[]);`

Returns in `S` the string representation of `k` in base `b`.

`void num_Uint2Uchar (unsigned char output[], unsigned int input[], int L);`

Transforms the L 32-bit integers contained in `input` into $4L$ characters and puts them into `output`. The order is such that the 8 most significant bits of `input[0]` will be in `output[0]`, the 8 least significant bits of `input[0]` will be in `output[3]`, and the 8 least significant bits of `input[L-1]` will be in `output[4L-1]`. Array `output` must have at least $4L$ elements.

`void num_WriteD (double x, int i, int j, int k);`

Writes `x` to current output. Uses a total of at least `i` positions (including the sign and point when they appear), `j` digits after the decimal point and at least `k` significant digits. The number is rounded if necessary. If there is not enough space to print the number in decimal notation with at least `k` significant digits (`j` or `i` is too small), it will be printed in scientific notation with at least `k` significant digits. In that case, `i` is increased if necessary. Restriction: `j` and `k` must be strictly smaller than `i`.

`void num_WriteBits (unsigned long x, int k);`

Writes `x` in base 2 in a field of at least $\max\{b, |k|\}$ positions, where b is the number of bits in an `unsigned long`. If $k > 0$, the number will be right-justified, otherwise left-justified.

`long num_MultModL (long a, long s, long c, long m);`

Returns $(as + c) \bmod m$. Uses the decomposition technique of [3] to avoid overflow. Supposes that $s < m$.

`double num_MultModD (double a, double s, double c, double m);`

Returns $(as + c) \bmod m$, assuming that a, s, c, and m are all *integers* less than 2^{35} (represented exactly). Works under the assumption that all positive integers less than 2^{53} are represented exactly in floating-point (in `double`).

`long num_InvEuclid (long m, long z);`

This function computes the inverse $z^{-1} \bmod m$ by the modified Euclid algorithm (see [2, p. 325]) and returns the result. If the inverse does not exist, returns 0.

`unsigned long num_InvExpon (int E, unsigned long z);`

This function computes the inverse $z^{-1} \bmod 2^E$ by exponentiation and returns the result. If the inverse does not exist, returns 0. Restriction: E not larger than the number of bits in an `unsigned long`.

num2

This module provides procedures to compute a few numerical quantities such as factorials, combinations, Stirling numbers, Bessel functions, gamma functions, and so on. These functions are more esoteric than those provided by `num`.

```
#include "gdef.h"
#include <math.h>
```

Prototypes

```
double num2_Factorial (int n);
```

The factorial function. Returns the value of $n!$

```
double num2_LnFactorial (int n);
```

Returns the value of $\ln(n!)$, the natural logarithm of the factorial of n. Gives at least 16 decimal digits of precision (relative error $< 0.5 \times 10^{-15}$)

```
double num2_Combination (int n, int s);
```

Returns the value of $\binom{n}{s}$, the number of different combinations of s objects amongst n.

```
#ifdef HAVE_LGAMMA
#define num2_LnGamma lgamma
#else
   double num2_LnGamma (double x);
#endif
```

Calculates the natural logarithm of the gamma function $\Gamma(x)$ at `x`. Our `num2_LnGamma` gives 16 decimal digits of precision, but is implemented only for $x > 0$. The function `lgamma` is from the `ISO C99` standard math library.

```
double num2_Digamma (double x);
```

Returns the value of the logarithmic derivative of the Gamma function $\psi(x) = \Gamma'(x)/\Gamma(x)$.

```
#ifdef HAVE_LOG1P
#define num2_log1p log1p
#else
   double num2_log1p (double x);
#endif
```

Returns a value equivalent to $\log(1 + x)$ accurate also for small x. The function `log1p` is from the `ISO C99` standard math library.

`void num2_CalcMatStirling (double *** M, int m, int n);`

Calculates the Stirling numbers of the second kind,

$$M[i,j] = \left\{ \begin{array}{c} j \\ i \end{array} \right\} \quad \text{for } 0 \le i \le m \text{ and } 0 \le i \le j \le n. \tag{1}$$

See D. E. Knuth, *The Art of Computer Programming*, vol. 1, second ed., 1973, Section 1.2.6. The matrix M is the transpose of Knuth's (1973). This procedure allocates memory for the 2-dimensionnal matrix M, and fills it with the values of Stirling numbers; the memory should be freed later with the function `num2_FreeMatStirling`.

`void num2_FreeMatStirling (double *** M, int m);`

Frees the memory space used by the Stirling matrix created by calling `num2_CalcMatStirling`. The parameter `m` must be the same as the `m` in `num2_CalcMatStirling`.

`double num2_VolumeSphere (double p, int t);`

Calculates the volume V of a sphere of radius 1 in t dimensions using the norm L_p, according to the formula

$$V = \frac{[2\Gamma(1+1/p)]^t}{\Gamma\left(1+t/p\right)}, \qquad p > 0,$$

where Γ is the well-known gamma function. The case of the sup norm L_∞ is obtained by choosing $p = 0$. Restrictions: $p \ge 0$ and $t \ge 1$.

`double num2_EvalCheby (const double A[], int N, double x);`

Evaluates a series of Chebyshev polynomials T_j, at point $x \in [-1,\ 1]$, using the method of Clenshaw [1], i.e. calculates and returns

$$y = \frac{A_0}{2} + \sum_{j=1}^{N} A_j T_j(x).$$

`double num2_BesselK025 (double x);`

Returns the value of $K_{1/4}(x)$, where K_ν is the modified Bessel's function of the second kind. The relative error on the returned value is less than 0.5×10^{-6} for $x > 10^{-300}$.

tables

This module provides an implementation of variable-sized arrays (matrices), and procedures to manipulate them. The advantage is that the size of the array needs not be known at compile time; it can be specified only during the program execution. There are also procedures to sort arrays, to print arrays in different formats, and a few tools for hashing tables. The functions `tables_CreateMatrix...` and `tables_DeleteMatrix...` manage memory allocation for these dynamic matrices.

As an illustration, the following piece of code declares and creates a 100×500 table of floating point numbers, assigns a value to one table entry, and eventually deletes the table:

```
double ** T;
T = tables_CreateMatrixD (100, 500);
T[3][7] = 1.234;
...
tables_DeleteMatrixD (&T);
```

```
#include "gdef.h"
```

Printing styles

```
typedef enum {
   tables_Plain,
   tables_Mathematica,
   tables_Matlab
   } tables_StyleType;
```

Printing styles for matrices.

Functions to create, delete, sort, and print tables

```
long ** tables_CreateMatrixL  (int M, int N);
unsigned long ** tables_CreateMatrixUL (int M, int N);
double ** tables_CreateMatrixD  (int M, int N);
```

Allocates contiguous memory for a dynamic matrix of `M` rows and `N` columns. Returns the base address of the allocated space.

```
void tables_DeleteMatrixL  (long *** T);
void tables_DeleteMatrixUL (unsigned long *** T);
void tables_DeleteMatrixD  (double *** T);
```

Releases the memory used by the matrix `T` (see `tables_CreateMatrix`) passed by reference, that is, using the `&` symbol. `T` is set to `NULL`.

```
void tables_CopyTabL (long T1[], long T2[], int n1, int n2);
void tables_CopyTabD (double T1[], double T2[], int n1, int n2);
```

Copies `T1[n1..n2]` in `T2[n1..n2]`.

```
void tables_QuickSortL (long T[], int n1, int n2);
void tables_QuickSortD (double T[], int n1, int n2);

#ifdef USE_LONGLONG
   void tables_QuickSortLL (longlong T[], int n1, int n2);
   void tables_QuickSortULL (ulonglong T[], int n1, int n2);
#endif
```

Sort the tables `T[n1..n2]` in increasing order.

```
void tables_WriteTabL (long V[], int n1, int n2, int k, int p, char Desc[]);

#ifdef USE_LONGLONG
   void tables_WriteTabLL (longlong V[], int n1, int n2, int k, int p,
                           char Desc[]);
   void tables_WriteTabULL (ulonglong V[], int n1, int n2, int k, int p,
                            char Desc[]);
#endif
```

Write the elements `n1` to `n2` of table `V`, `k` per line, `p` positions per element. If `k` = 1, the index will also be printed. `Desc` contains a description of the table.

```
void tables_WriteTabD (double V[], int n1, int n2, int k, int p1, int p2,
                       int p3, char Desc[]);
```

Writes the elements `n1` to `n2` of table `V`, `k` per line, with at least `p1` positions per element, `p2` digits after the decimal point, and at least `p3` significant digits. If `k` = 1, the index will also be printed. `Desc` contains a description of the table.

```
void tables_WriteMatrixD (double** Mat, int i1, int i2, int j1, int j2,
                          int w, int p, tables_StyleType style,
                          char Name[]);
```

Writes the submatrix with lines `i1` $\leq i \leq$ `i2` and columns `j1` $\leq j \leq$ `j2` of the matrix `Mat` with format `style`. The elements are printed in `w` positions with a precision of `p` digits. `Name` is an identifier for the submatrix.

For `Matlab`, the file containing the matrix must have the extension `.m`. For example, if it is named `poil.m`, it will be accessed by the simple call `poil` in `Matlab`. For `Mathematica`, if the file is named `poil`, it will be read using `<< poil;`.

```
void tables_WriteMatrixL (long** Mat, int i1, int i2, int j1, int j2, int w,
                          tables_StyleType style, char Name[]);
```

Similar to `tables_WriteMatrixD`.

`long tables_HashPrime (long n, double load);`

Returns a prime number M to be used as the size (the number of elements) of a hashing table. M will be such that the load factor n/M do not exceed `load`. If `load` is small, an important part of the table will be unused; that will accelerate searches and insertions. This function uses a small sequence of prime numbers; the real load factor may be significatively smaller than `load` because only a limited number of prime numbers are in the table. In case of failure, returns -1.

mystr

This module offers some tools for the manipulation of character strings.

```
void mystr_Delete (char S[], unsigned int index, unsigned int len);
```

Deletes `len` characters from S, starting at position `index`.

```
void mystr_Insert (char Res[], char Source[], unsigned int Pos);
```

Inserts the string `Source` into `Res`, starting at position `Pos`.

```
void mystr_ItemS (char R[], char S[], const char T[], unsigned int N);
```

Returns in R the N-th substring of S (counting from 0). Substrings are delimited by any character from the set T.

```
int mystr_Match (char Source[], char Pattern[]);
```

Returns `1` if the string `Source` matches the string `Pattern`, and `0` otherwise. The characters "?" and "*" are recognized as wild characters in the string `Pattern`.

```
void mystr_Slice (char R[], char S[], unsigned int P, unsigned int L);
```

Returns in `R` the substring in `S` beginning at position `P` and of length `L`.

```
void mystr_Subst (char Source[], char OldPattern[], char NewPattern[]);
```

Searches for the string `OldPattern` in the string `Source`, and replaces its first occurence with `NewPattern`.

```
void mystr_Position (char Substring[], char Source[], unsigned int at,
                     unsigned int * pos, int * found);
```

Searches for the string `Substring` in the string `Source`, starting at position `at`, and returns the position of its first occurence in `pos`.

addstr

The functions described here are convenient tools for constructing character strings that contain a series of numeric parameters, with their values. For example, suppose one wishes to put "`LCG with m = 101, a = 12, s = 1`" in the string `str`, where the actual numbers 101, 12, and 1 must be taken as the values of `long` integer variables `m`, `a`, and `s`. This can be achieved by the instructions:

```
strcpy (str, "LCG with ");
addstr_Long (str, " m = ", m);
addstr_Long (str, ", a = ", m);
addstr_Long (str, ", s = ", s);
```

Each function `addstr_... (char *to, const char *add, ...)` first appends the string `add` to the string `to`, then appends to it a character string representation of the number (or array of numbers) specified by its last parameter. In the case of an array of numbers (e.g., `addstr_ArrayLong`), the parameter `high` specifies the size of the array, and the elements `[0..high-1]` are added to `str`. The `...LONG` versions are for 64-bit integers. In all cases, the string `to` should be large enough to accomodate what is appended to it.

```
#include "gdef.h"
```

Prototypes

```
void  addstr_Int (char *to, const char *add, int n);

void  addstr_Uint (char *to, const char *add, unsigned int n);

void  addstr_Long (char *to, const char *add, long n);

void  addstr_Ulong (char *to, const char *add, unsigned long n);

void  addstr_Double (char *to, const char *add, double x);

void  addstr_Char (char *to, const char *add, char c);

void  addstr_Bool (char *to, const char *add, int b);

#ifdef USE_LONGLONG
void  addstr_LONG (char *to, const char *add, longlong n);

void  addstr_ULONG (char *to, const char *add, ulonglong n);
#endif
```

```
void  addstr_ArrayInt (char *to, const char *add, int high, int []);

void  addstr_ArrayUint (char *to, const char *add, int high,
                        unsigned int []);

void  addstr_ArrayLong (char *to, const char *add, int high, long []);

void  addstr_ArrayUlong (char *to, const char *add, int high,
                         unsigned long []);

void  addstr_ArrayDouble (char *to, const char *add, int high, double []);
```

tcode

Program `tcode` makes compilable code from a TEX or LATEX document. It creates a file `FOut` for a compiler like cc (or any other), starting from a file `FIn`. The names of these two files must be given by the user, with appropriate extension, when calling the program. The two file names (with the extension) must be different.

Only the text included between the `\code` and `\endcode` delimiters will appear in the second file. Only the following LATEX commands can appear between `\code` and `\endcode`:

`\hide`, `\endhide`, `\iffalse`, `\fi`, `\smallcode`, `\smallc`.

Everything else between `\code` and `\endcode` must be legal code in the output file, apart from two exceptions: the TEX command `\def\code`, defining `\code` will not start a region of valid code, nor will `\code` appearing on a line after a TEX comment character `%`.

If one wants code to appear in the compilable file, but be invisible in the `dvi` file obtained from processing the `tex` file with LATEX, one should put this code between the delimiters `\hide` and `\endhide`, or between the delimiters `\iffalse` and `\fi`.

The program is called by:

`tcode` ⟨`FIn`⟩ ⟨`FOut`⟩

Examples: If one wants to extract the *C* code from the LATEX file `chrono.tex`, and place it in the header file `chrono.h`, the following command should be used:

```
tcode  chrono.tex  chrono.h
```

To extract *Java* code from the LATEX file `Event.tex`, and place it in the file `Event.java`, one must use:

```
tcode  Event.tex  Event.java
```

References

[1] C. W. Clenshaw. Chebychev series for mathematical functions. National Physical Laboratory Mathematical Tables 5, Her Majesty's Stationery Office, London, 1962.

[2] D. E. Knuth. *The Art of Computer Programming, Vol. 2.* Addison-Wesley, Reading, Mass., second edition, 1981.

[3] P. L'Ecuyer and S. Côté. Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software*, 17(1):98–111, 1991.

[4] S. Wolfram. *The Mathematica Book.* Wolfram Media/Cambridge University Press, Champaign, USA, third edition, 1996.