The STURM Library Manual

A C++ library for symbolic computation.
Edition 1.0 for Version 1.0 of the STURM library.

Hoon Hong, Andreas Neubacher, Volker Stahl

RISC-Linz Report Series No. 94-30

1 Introduction

This manual describes the functionality of the STURM library, a C++ library for sequential and parallel symbolic computation.

The STURM library currently provides the features listed below. (The library is still under development so its functionality is constantly being extended.)

- Memory management based on the notion of an infinite heap simulated by a garbage collection scheme (see Section 3.2 [Memory Management], page 7).
- Arithmetic over integers (see Chapter 4 [Integers], page 18) and rationals (see Chapter 5 [Rational Numbers], page 25) of arbitrary size.
- Arithmetic over double precision floating point numbers (see Chapter 6 [Double Precision Floating Point Numbers], page 30) and intervals (see Chapter 7 [Double Precision Intervals], page 34).
- A multi-processor version which supports parallel execution (see section "Overview" in *The C++ Interface to the STURM Multi-Processor Kernel*).

The example code below should give an impression of what programs using the STURM library look like. This simple program reads two integers from the standard input, adds them, and writes the result to the standard output.

```
#include <iostream.h>
#include "Sturm/Kernel.h"
#include "Sturm/Integer.h"

int
STURM_Main(int argc, char *argv[])
{
  Type_Integer A, B, C;

  cout << " Insert first integer: "; cin >> A;
  cout << "Insert second integer: "; cin >> B;

  C = A + B;
  cout << A << " + " << B << " =\n" << C << "\n";
}
```

Assume that the name of the file containing this program is '`Simple.cc`', that the environment variable `Sturm` contains the path of the root directory for the STURM library, and that the directory '`$Sturm/Bin`' is on the shell's search path. Then this example program would be compiled, linked and run by the following sequence of commands:

```
$ sturmc++ Simple.cc -o Simple
$ ./Simple
 Insert first integer: 123456789012345678901234567890
Insert second integer: 987654321098765432109876543210000
123456789012345678901234567890 + 987654321098765432109876543210000 =
988888888898888888888988888888890
```

Please note that instead of using the systems C++ compiler programs should be compiled and linked with the command `sturmc++` which can be found in the directory '`$Sturm/Bin`'. This script takes care of setting include paths correctly and of linking the required libraries.

2 Writing Programs

This section lists some points which should be kept in mind when writing programs using the STURM library.

2.1 Header Files

The STURM header files are located in a separate directory which must be given in addition to the filename. This means that if you need to include a STURM header file '*Foo*.h' you must use the statement

```
#include "Sturm/Foo.h"
```

In general there is one separate header file for every package. This must be included if any function or type of a given package is used. The documentation for each package gives the name of that header file.

Every header file only exports those types and function declarations which are documented. You should never assume that any header file is included by default. For example, some of the functions of the rational package (which are declared in '`Rational.h`') take objects of type `Type_Integer` as arguments. This does not imply that it is guaranteed that the header file '`Integer.h`' is included by '`Rational.h`'. If you use functions from both the rational and the integer package you should explicitly include both header files.

Possible deviations from this general scheme are documented in the package descriptions.

2.2 The Main Program

Instead of using the standard C++ `main` function as the entry point to your program, the STURM library requires a function called `STURM_Main`. This has the consequence that before any of your code is executed, the following operations take place:

1. The command line arguments are parsed. Arguments with special meaning for the STURM library (see Section 2.3 [Command Line Arguments], page 4) are used for changing various default values.
2. Various initialization functions are called taking either default values or values set by command line arguments.
3. The function `STURM_Main(int argc, char *argv[])` is called with `argc` and `argv` modified such that any arguments reserved for the STURM library are removed.

Warning: The necessity of having initialization functions called before *any* of the STURM library functions are executed means that you must not define any global variables of a type provided by the STURM library. Such global variables might trigger constructor calls whose order is compiler dependent and thus may cause undefined behaviour of the system.

2.3 Command Line Arguments

The STURM library reserves the following command line arguments:

`--help` Displays a usage message describing the STURM command line arguments and terminates the program immediately.

`--heap` *heapsize*
Sets the size of the heap to at least *heapsize* bytes.

Note that these arguments are not passed to `STURM_Main` (see Section 2.2 [The Main Program], page 3).

2.4 Compiling and Linking

Although the setup of the STURM library files may differ among systems, in general it should look as follows:

There is a top level directory below which all files are located. We will refer to this directory through the shell environment variable `Sturm`.

The directory '`$Sturm/Bin/`' contains shell scripts which simplify working with the library. For these scripts to work the variable `Sturm` must be set correctly and the '`Bin`' directory must be on your shell's search path.

If your environment is set up correctly then you can execute the following shell commands for compiling and linking a program called *foo*:

```
$ sturmc++ foo.cc -c -o foo.o
$ sturmc++ foo.o -o foo
```

This will produce an executable called *foo*. The script `sturmc++` accepts all options of the C++ compiler used. For documentation of the the implementation on your system execute the script as

```
$ sturmc++ -h
```

If you have problems with your setup please contact your local STURM maintainer.

2.5 Implementation Details

All source files containing functions relevant to the general initialization of the STURM library are located in the directory

'`$Sturm/Src/Kernel/Initialization/`'

This is where the actual `main` function is defined. It carries out the following operations:

1. Command line options are parsed by calls to the following functions:

   `GetOpts` A local function which parses system dependent arguments. (Currently in the sequential version there are only arguments for memory management).

   `STURM_GetOpts`
   A global function which parses system independent arguments. (Currently there are no such arguments.)

2. Various packages are initialized by calls to the following functions:

   `Init` A local function which initializes various core packages. (Currently this is only the memory management package.)

   `STURM_Init`
   A global function which initializes system independent packages. (Currently these are the error handler, integer, and the floating point unit packages.)

3. The function `STURM_Main` (i.e. the application code) is called.
4. Cleanup for various packages is done by calls to the following functions:

   `Cleanup` A local function which cleans up system dependent packages. (Currently there are no such packages.)

   `STURM_Cleanup`
   A global function which cleans up system independent packages. (Currently there is no such package.)

How to Add Initialization Functions

If a package that requires initialization is added to the library, the following steps have to be taken:

- If the package requires initialization parameters, add entries for these parameters to the structure `Type_DefaultOpts` which is defined in the file '`$Sturm/Src/Kernel/Initialization.h`'.
- Add code for setting default values and parsing command line arguments for these parameters to the function `STURM_GetOpts`.
- Add code for calling the package's initialization function to the function `STURM_Init`.
- If necessary, add code for cleaning up the package's data structures to the function `STURM_Cleanup`.
- Recompile the library.

3 Kernel

The kernel of the STURM library provides an interface to system dependent functionalities and defines an abstract machine upon which the other packages of the library are based.

The functions and types of the kernel are defined in the header file '`Sturm/Kernel.h`'.

3.1 Error Handling

If an error in a STURM application occurs, an error handler is called via `ERR_CallHandler`. Dedicated error handlers can be defined by `ERR_DefineHandler`.

The functions and types of the error handler package are defined in the header file '`Sturm/Kernel.h`'.

3.1.1 Calling an Error Handler

The function `ERR_CallHandler (`*fn*`,`*msg*`,`*argc*`,`*argv*`)` calls the error handler for function *fn*. If no dedicated error handler for this function is defined, then a default error handler is called. The arguments of `ERR_CallHandler` are as follows

fn
: The name of the function for which an error handler will be called

msg
: An error message.

argc, *argv*
: Optional arguments passed to the error handler.

3.1.2 Defining a Dedicated Error Handler

A specific error handler for some function can be defined by `ERR_DefineHandler (`*fn*`, `*handler*`)`. Here, *fn* is a character string which describes the function for which the handler is defined and *handler* is a pointer to an error handling function. If *fn* is the empty string, then the default error handler is redefined as *handler*.

An error handler is a void function with the arguments `(char *`*fn*`, char *`*msg*`, char *`*file*`, int` *line*`, int` *argc*`, char **`*argv*`)`. When a handler is called via `ERR_CallHander(`*fn*`,`*msg*`,`*argc*`,`*argv*`)` the arguments *fn*,*msg*,*argc*,*argv* are passed to the handler and *file*, *line* are the file name and the line number where the handler was called.

3.1.3 Example Programs

The following example shows how a dedicated error handler is defined and called.

```
#include <iostream.h>
#include <stdlib.h>
#include "Sturm/Kernel.h"

// The dedicated error handler for function STURM_Main
void MyHandler(char *fn, char *msg, char *file, int line, int argc, char *argv[])
{
  cout << "This is the error handler for function " << fn << ".\n";
  cout << "Error message: " << msg << ".\n";
  cout << "File: " << file << "\n";
  cout << "Line: " << line << "\n";
  if(argc > 0)
  {
    cout << "Arguments:\n";
    for(int i=0; i<argc; i++)
      cout << argv[i] << "\n";
  }
  exit(-1);
}


// Main Program
int STURM_Main(int argc, char *argv[])
{
  ERR_DefineHandler("STURM_Main",MyHandler);
  ERR_CallHandler("STURM_Main","No error, just a test",0,NULL);
}
```

The output of this program is

```
This is the error handler for function STURM_Main.
Error message: No error, just a test.
File: t.cc
Line: 24
```

3.1.4 Implementation Details

The list of pointers to error handlers and corresponding function names are stored in a global data structure which is pointed to by `Global_TrapVectorList`. This list is initialized during `main` by the function `ERR_Init`.

3.2 Memory Management

The Memory Management facilities simulate an infinite heap from which contiguous sections of memory (e.g. for arrays and structures) of nearly arbitrary size can be allocated. This is realized by a garbage collection mechanism which is activated when an allocation function cannot find sufficient free memory on the heap. As long as dynamic memory allocation is only done by the facilities described in this document the programmer need not be concerned with memory management considerations such as avoiding memory leaks.

3.2.1 Allocation

Memory is allocated by calling the function `MEM_New`. Code using this function may look as follows:

```
Type_Integer *A = MEM_New(Type_Integer, 100);
...
Type_Integer I = "-9876543210";
for (i=0; i<100; i++)
  A[i] = I*i;
```

It is not necessary to explicitly free memory allocated on the heap. The garbage collector will free all blocks which are not referenced any more. For speeding up garbage collection you may use the function `MEM_NewInt` for allocating memory which is *not* used for storing STURM objects or pointers to memory blocks allocated on the heap. For example, in the code fragment

```
int *len = MEM_NewInt(int, 256);
char **str = MEM_New(char *, 256);
...
for (i=0; i<256; i++) {
  len[i] = 50;
  str[i] = MEM_NewInt(char, 50);
}
```

it is safe to use `MEM_NewInt` for allocating the arrays of `int` and `char`, but it must not be used for allocating the array of pointers to `char`. Similarly, in the previous example the array of `Type_Integer` is an array of STURM objects which may contain pointers and thus must *not* be allocated using `MEM_NewInt`.

A safe approach is to use `MEM_New` in case of doubt, and to replace it by `MEM_NewInt` only if you are certain that the allocated memory will never contain pointers to other memory.

If you do not want to allocate a new block of memory but rather change the size of an already allocated block without changing its contents (e.g. to enlarge a buffer when it becomes full) use the function `MEM_Renew`. Often code similar to the following is used when reading some input whose size cannot be known beforehand.

```
int size = 10;
float *A = MEM_NewInt(float, size);

for (int i = 0; ! cin.eof(); i++) {
  if (i == size) {
    size += 10;
    A = MEM_Renew(float, A, size);
  }
  cin >> A[i];
}
```

In this example an array of 10 `float` objects is allocated beforehand. If the size of the array is exceeded while reading values from the standard input the array is re-allocated with a larger size.

As an alternative to the type based functions MEM_New, MEM_NewInt, and MEM_Renew you can also use the byte based functions MEM_Allocate, MEM_AllocateInt, and MEM_Reallocate. (Actually, the former are macros wrapped around the latter.) The following code shows the difference between these two sets of allocation functions.

```
char **C = MEM_New(char *, 1);
double *D = MEM_NewInt(double, 100);
C = MEM_Renew(char *, C, 20);

char **C = (char **)MEM_Allocate(sizeof(char *));
double *D = (double *)MEM_AllocateInt(sizeof(double)*100);
C = (char **)MEM_Reallocate((void *)C, sizeof(char *)*20);
```

Although the functionality is the same, using the first kind of allocation will make the code more readable and is therefore recommended.

3.2.2 Size Information

If you need to know how many objects of a given type fit into a given block of memory use the function MEM_Number. A possible application of this function is shown in the following example:

```
double AddAll(double *D)
{
  double R = 0.0;

  for (int i = 0; i < MEM_Number(double, D); i++)
    R += D[i];

  return R;
}
```

This function adds all double values in an array and returns the result. It is not necessary to pass the size of the array D as an additional argument, because MEM_Number returns this information.

Warning: This works only for pointers returned by one of the allocation functions provided by the STURM library. Passing a pointer to any other kind of memory to MEM_Number will result in undefined behaviour and may even crash the program.

As with the allocation functions, there is a byte oriented function MEM_Size which returns the size of a block of memory in bytes. One might use this function in the following context:

```
char *C = MEM_New(char, 5);

cout << MEM_Size((void *)C) - 5;
cout << " bytes are wasted when 5 characters are allocated.\n";
```

This code fragment checks how many bytes are actually allocated when the programmer asks for a block of memory large enough to store at least 5 characters. (In most implementation, the overhead will be 3, i.e. allocation requests are rounded up to the next multiple of 4 bytes.)

3.2.3 Deallocation

The garbage collection mechanism of the STURM library does not make it necessary to free memory once it is not used any more. When the heap is full, the garbage collector will free all memory blocks which are not referenced any more.

The efficiency (memory use and collection speed) of the garbage collector may be improved by two means:

1. Calling the function `MEM_Free` with a pointer to memory which is not used anymore as its argument.
2. Clearing unused pointers by setting them to `NULL`.

It is not guaranteed that these measures will indeed result in reduced memory usage or a speedup of the overall running time of a program. In particular, frequent allocation / deallocation cycles may even be slower than several allocations followed by a garbage collection. Furthermore, `MEM_Free` may postpone deallocation until the next garbage collection takes effect. Therefore, it is recommended to explicitly free only unusually large blocks of memory and to clear only pointers whose lifetime spans most of the runtime of the program.

In general, some experimentation will be required to fine tune a program using these means.

3.2.4 Global Variables

The garbage collection mechanism employed by the STURM library has one major drawback: It cannot handle global variables transparently. Therefore, it is necessary to make the garbage collector aware of global variables used for storing pointers to memory blocks. This is done by calling the function `MEM_Register`.

```
Type_Integer A[10];
static char *str;
...
int STURM_Main(int argc, char *argv[])
{
  MEM_Register((void *)A, 10*sizeof(Type_Integer));
  MEM_Register((void *)(&str), sizeof(char *));
  ...
  A[3] = 100;
  ...
  str = MEM_New(char, 81);
}
```

In this code fragment there are a global array `A` containing STURM objects of type `Type_Integer` and a static pointer `str` to an array of characters which will be allocated on the heap. This example shows two important points:

- The registering function must be passed *pointers to pointers* to heap memory.

  In the case of the `Type_Integer` array the pointers to heap memory may be located in the array, therefore `A`, the pointer to the array, is passed to `MEM_Register`. In the case of the array of characters, `str` is a global pointer to heap memory and therefore a pointer to `str` is passed to `MEM_Register`.

- The global variables must not be used before having been registered.

  For example, it would be incorrect to replace the declaration of `str` by

  ```
  static char *str = MEM_New(char, 81);
  ```

  because then `MEM_New` would be called before `STURM_Main` is entered, i.e. at a time when `str` had not yet been registered. This might cause undefined behaviour of the program.

3.2.5 Example Programs

The example program given below implements a simple string class which supports reading, writing, and concatenation (via the `+=` operator) of strings. It demonstrates the use of `MEM_NewInt`, `MEM_Renew`, and `MEM_Number`.

```
#include <string.h>
#include <iostream.h>
#include "Sturm/Kernel.h"


class MyString {
  char *c;
public:
  MyString(void) { c = NULL; }
  MyString(char *str)  {
    c = MEM_NewInt(char, strlen(str)+1);
    strcpy(c, str);
  }
  void operator+=(const MyString &S) {
    if (MEM_Number(char, c) <= strlen(c) + strlen(S.c))
      c = MEM_Renew(char, c, strlen(c) + strlen(S.c) + 1);
    strcat(c, S.c);
  }
  friend istream &operator>>(istream &is, MyString &S);
  friend ostream &operator<<(ostream &os, const MyString &S) {
    os << S.c; return os;
  }
};

istream &
operator>>(istream &is, MyString &S)
{
  int len = 10;
  S.c = MEM_NewInt(char, len);
  for (int i=0; ; i++) {
    if (i == len) {
      len += 10;
      S.c = MEM_Renew(char, S.c, len);
    }
    is.get(S.c[i]);
    if ((! is.good()) || S.c[i] == '\n') {
      S.c[i] = 0;
```

```
        break;
      }
    }
    return is;
  }


  int STURM_Main(int argc, char **argv)
  {
    MyString S, T("You entered ");

    cout << "Please enter a string: ";  cin >> S;
    T += S;
    cout << T << "\n";
  }
```

A session with this program looks as follows:

```
Please enter a string: A TESTSTRING
You entered A TESTSTRING
```

3.2.6 Implementation Details

The basic idea for the heap layout and the allocation routines were taken from the GNU "malloc" package by Mike Haertel. this package. A mark-and-sweep garbage collector was added and everything was re-implemented in C++.

3.2.6.1 Overview

The current implementation of memory management works as follows:

- During initialization the memory heap is allocated from the system. It is paginated into *pages* of 4 kB. Furthermore, an array of structures containing administrative information for a page is allocated. There is a one-to-one correspondence between the pages and the elements of this array, so an address into a page can be converted into the corresponding array index and vice versa.

  The pages are combined into *blocks* of certain types (free, allocated, locked). Due to allocation / freeing of memory, after some time the heap comprises a sequence of free and allocated blocks. All free blocks are linked into a list from which memory is taken for satisfying allocation requests. At initialization, the whole heap is set up as a single free chunk of memory.
- When the allocation function is called, a distinction between two cases is made:
  + If the requested size is smaller than half a page then the size is rounded to the next power of two. The *fragment list* for the given power of two is checked. If it contains fragments, a pointer to the first fragment is returned. Otherwise, a page is taken from the list of free blocks and split into fragments which are linked into the fragment list. Then a pointer to the first fragment is returned.
  + If the requested size is larger than half a page then the list of free blocks is searched for a block which is large enough to satisfy the allocation request. When such a block if found,

it is split into a part consisting of as few pages as needed to fit the requested size and a part consisting of the remaining pages. The remaining part is linked into the list of free blocks, the other part is marked as allocated and a pointer to it is returned.

In either case, if no free page or large enough block is found in the free list, the garbage collector is called.

- The garbage collector works in two steps:

  Mark: The stack and all pointers in the list set up by `MEM_Register` are traversed. If a value is found which might be interpreted as a pointer to the heap, all pages of the corresponding block are marked and the corresponding block is also searched for potential pointers.

  Sweep: The heap is traversed. All marks are cleared. Blocks which were marked are left untouched, blocks which were unmarked are linked into the list of free blocks (with adjacent free blocks being merged into a single large block).

3.2.6.2 Initialization

Before any function of the memory management package may be called the package must be initialized by the function `MEM_Init`. In general this function is not called directly. It is part of the initialization sequence being executed before `STURM_Main` is entered (see Section 2.2 [The Main Program], page 3).

3.3 Floating Point Unit

The functions described in this section provide a high level interface to the registers of the floating point unit. In particular, they allow to

- specify rounding modes,
- detect or mask exceptions.

The functions correspond to the IEEE standard 754.

The functions of the floating point unit are declared in the header file '`Sturm/Kernel.h`'.

3.3.1 Rounding Modes

The rounding mode specifies in which direction floating point operations are rounded. The current rounding mode is stored in a register of the floating point unit and can be modified by the following operations:

`FPU_Up` Round up.

`FPU_Down` Round down.

`FPU_ToNearest`
  Round to nearest.

`void FPU_ToZero`
: Round to zero.

Furthermore `FPU_SetRound(`*x*`)` and *x* `= FPU_GetRound()` allow to set and get the rounding mode, where *x* is one of `FPU_UP`, `FPU_DOWN`, `FPU_NEAREST`, `FPU_ZERO`.

3.3.2 Exception Handling

The IEEE standard 754 specifies the exceptions division by zero, overflow, underflow, invalid operation and imprecise result. Each of these exceptions can be detected or masked. If an exception is not masked, then the program is aborted if the exception is raised. By default all exceptions except division by zero and invalid operation are masked.

The following constants identify an exception:

`FPU_DIVBYZERO`
: Division by zero

`FPU_OVERFLOW`
: Overflow

`FPU_UNDERFLOW`
: Underflow

`FPU_INVALOP`
: Invalid operation

`FPU_IMPRECISE`
: Imprecise result

The function `FPU_IsRaised(`*X*`)` returns `TRUE` if exception *X* is raised and false otherwise. Exception *X* can be cleared by `FPU_UnRaise(`*X*`)`. It can be explicitly raised by `FPU_Raise(`*X*`)`.

The function `FPU_IsMasked(`*X*`)` returns `TRUE` if exception *X* is masked and false otherwise. Exception *X* can be masked by `FPU_Mask(`*X*`)`. It can be unmasked by `FPU_UnMask(`*X*`)`.

3.3.3 Status of Floating Point Unit

The function `FPU_Status()` prints information on the exception flags and the rounding mode.

For example

```
#include <Sturm/Kernel.h>

int
STURM_Main(int argc, char *argv[])
{
  FPU_Down();
  FPU_Mask(FPU_DIVBYZERO);
  FPU_Raise(FPU_UNDERFLOW);
```

```
  FPU_UnMask(FPU_OVERFLOW);

  FPU_Status();
}
```

outputs

```
Exception flags
===============      Masked  Raised
  Overflow             no      no
  Underflow            yes     yes
  Imprecise            yes     no
  Invalid              no      no
  Division by Zero     yes     no

Rounding mode
=============
  Down
```

3.3.4 Example Programs

The following program squares a double precision floating point number until overflow occurs.

```
#include "Sturm/Kernel.h"

int
STURM_Main(int argc, char *argv[])
{
  double x = 2.0;
  int    i = 0;

  while(!FPU_IsRaised(FPU_OVERFLOW))
  {
    x*=x;
    ++i;
  }
  cout << "Overflow after " << i << " square operations.\n";
}
```

The output of this program is

```
Overflow after 10 square operations.
```

However, if the program is compiled with optimizer flag and e.g. gcc 2.5.8, it never terminates. This shows that the optimizer assumes that floating point numbers and real numbers are equivalent, which is of course not the case. Thus, optimization can lead to wrong executables! Fortunately, it seems that rounding modes are treated correctly by the optimizer.

3.3.5 Implementation Details

The functions defined in the floating point unit package are system dependent and not portable. The function `FPU_Init()` sets the default exception mask and rounding mode and defines an error handler, which calls the STURM error handler with an appropriate error message.

3.4 Program Termination

Two functions are provided for terminating the execution of a program. Although these functions are similar to the ANSI-C functions `exit` and `abort` these latter two must not be used in STURM code.

The function `STURM_Exit(`*status*`)` closes and flushes all output streams and terminates the program. The value of the *status* argument is passed to the shell environment as the exit code of the program.

The function `STURM_Abort()` aborts the execution of a program. Flushing of output streams is not guaranteed. A coredump may be produced and a failure status is reported to the shell environment.

3.5 Clock

Two functions provide access to the system clock. These are a replacement for the ANSI-C function `clock` which must not be used in STURM programs.

The function `TMR_Clock()` returns the number of high-resolution clock ticks which passed since program startup. It should yield the highest possible resolution available from the system clock. Therefore, turnaround may be short. (Under one hour for a microsecond clock.) For converting high-resolution ticks into seconds the following constant is defined:

TMR_TICKS Constant of `unsigned long`
: The number of ticks per second as returned by the function `TMR_Clock`.

In general timing is done as follows:

```
unsigned long clk;
double secs;
...
t = TMR_Clock();
...
// Part of the program to be timed.
...
t = TMR_Clock() - t;
secs = (double)t/(double)TMR_TICKS;
cout << "Execution took " << secs << " seconds.\n";
```

The STURM library also provides a function which should be used for timing programs running over a long period of time. (In actual implementations, both functions may be the same.) `TMR_ClockLong()` returns the number of low-resolution clock ticks since program startup. This guarantees a turnaround time of more than 24 hours. (Several weeks for a millisecond clock.) For converting low-resolution ticks into seconds the following constant is defined:

TMR_LONGTICKS Constant on `unsigned long`

The number of ticks per second as returned by the funcion `TMR_ClockLong`.

3.6 Boolean Type

C++ does not provide a dedicated data type for boolean values but rather uses integer types for this purpose. For making the distinction between integer types and boolean types more explicit, STURM defines the following type:

Type_Bool Arithmetic type

This defines an unsigned integer type which ranges over the values `TRUE` and `FALSE`.

The only values which an object of `Type_Bool` can take are the following:

TRUE Constant of `Type_Bool`

This constant represents the boolean value "true", "1", "on", "yes", etc.

FALSE Constant of `Type_Bool`

This constant represents the boolean value "false", "0", "off", "no", etc.

This type can be used with C++ logical operators as shown in the following example.

```
#include "Sturm/Kernel.h"
  ...
  Type_Bool error = FALSE;
  ...
  if (error && debug_level > 3) {
    cout << "Error in function ...!\n";
    ...
  }
```

4 Integers

Operations on arbitrarily long integers (sometimes also called *bignums*) are based on the following type:

Type_Integer Arithmetic Type

An arithmetic type ranging over the integral numbers. Its size is not fixed and only restricted by the available memory.

In general *integer* in this manual denotes an object of type `Type_Integer`. An object of fixed size integral type such as `int`, `unsigned long`, etc. will be termed *C integer* or *C++ integer*.

C++ integers can be mixed with `Type_Integer` in most computations, but conversions always go towards the latter, i.e. it is not possible to convert a `Type_Integer` to e.g. an `int`.

The functions and types of the integer package are defined in the header file '`Sturm/Integer.h`'.

4.1 Construction and Conversion

When declaring integers there are two possibilities: A declaration with or without an initializing value.

Declaring and initializing an integer can be done as follows:

`Type_Integer` *I* `=` *j*`;`
: This initializes *I* to the value of *j*, where *j* is a variable or constant of any C++ integral type (such as `int`, `unsigned long`, etc.).

`Type_Integer` *I* `=` *s*`;`
: This initializes *I* to the value given in *s*, where *s* is a C++ null-terminated string containing the character representation of an integer.

`Type_Integer` *I* `=` *J*`;`
: This initializes *I* to the value of *J*, where *J* is also an object of `Type_Integer`.

Examples for such initializations are

```
Type_Integer I = 100, J = "-12345678900000000000";
Type_Integer A = I;
```

As with all C++ constructors the alternative form

```
Type_Integer I(...);
```

is also available. Furthermore, these constructors will be used for explicit and implicit type conversions.

A declaration without any initialization looks as follows:

```
Type_Integer I;
```

This declares I but does not set it to any meaningful value. If this variable is used e.g. as a summand the program's behaviour will be undefined. The only valid use which can be made with an uninitialized integer is assigning it a value by statements such as

```
I = 10;
```

or

```
GCDExtended(A, B, I, S, T);
```

In the second example I is assigned the value of the gcd of A and B, which is a valid use of an uninitialized variable. Using the uninitialized I in the position of e.g. A would most probably crash the program.

This type of declaration exists mainly to allow declaration of *global* objects of type `Type_Integer`. Global objects must not be initialized because it cannot be guaranteed that various setup functions for the STURM package are executed before the constructors for global objects are called. The void constructor does not require prior execution of any such setup function.

In addition to the constructors given above there are also three functions which return special values of integers.

`ZeroInteger`
: Returns the integer representation of 0. This is more efficient than using the C++ constant `0`.

`OneInteger`
: Returns the integer representation of 1. This is more efficient than using the C++ constant `1`.

`RandomInteger`
: Returns a random number of a given size.

4.2 Arithmetic

Arithmetic is done using the standard C++ operators `+`, `-`, `*`, `/`, `%`, `+=`, `-=`, `*=`, `/=`, `%=`, `++`, `--` with minor deviations from their standard semantics. The main differences in the operator semantics are:

- "Destructive" operators such as `+=` or `--` return `void` instead of an integer value. This means that one cannot write statements such as `R = (I += J)` or `R = 2*--I + J`.
- Integer division and remainder are uniquely specified. (ANSI C allows the sign of the remainder to be implementation dependent.)

Operators working on the bit representation of integers (such as `~` or `|`) are not available.

Here is some example code using operators on integers:

```
Type_Integer I = "9999999999999999999999", J;
int i;

for (i = 0; i < 100; i++)
  I += i;

J = (2 * I) % -101;
```

In addition to the operators various functions and procedures are available. (This is currently a minimal set which will be extended as the need arises.)

`QuotRem` Computes the quotient and the remainder in a single function call,

`GCD, GCDExtended`
Compute the greatest common divisor (optionally with co-factors),

`Abs, Sign` Compute the absolute value and the sign of an integer.

4.3 Predicates

Integers can be compared using the standard C++ operators `==`, `!=`, `<=`, `>=`, `<`, `>` with their standard semantics. The only difference to the standard operators is that the results are of type `Type_Bool` instead of `int`.

Some examples for integer comparisons appear in the code fragment given below:

```
Type_Integer I = 100, J = "876554332298765544332579667 4249";

if (I != J) {
  ...
}
if (J == 5) ...
while (I < J)
  J /= 10;
```

For efficient comparison with zero and one use the functions `IsZero`, `IsNonZero`, `IsNonNegative`, `IsNonPositive`, `IsNegative`, `IsPositive`, and `IsOne`. For example

```
for (Type_Integer I = 100; IsPositive(I); I--)
  ...
```

will run faster than

```
for (Type_Integer I = 100; 0 < I; I--)
  ...
```

4.4 Input and Output

Integers can be read and written using the standard C++ input/output operators `>>`, `<<` with their standard semantics.

For example, you can write

```
#include <iostream.h>
#include "Sturm/Integer.h"

...

  Type_Integer I, J;

  cout << "Please enter two integers:\n";
  cin >> I;  cin >> J;
  cout << "The sum of " << I << " and " << J " is " << I + J << ".\n";
```

Valid input to this program looks as follows:

```
Please enter two integers:
1234567890000000000000000000
-3141592
The sum of ...

Please enter two integers:
+888776543332  100
The sum of ...
```

4.5 Example Programs

The following program reads an integer, multiplies it with a constant and outputs the result:

```
#include <iostream.h>
#include "Sturm/Kernel.h"
#include "Sturm/Integer.h"

int
STURM_Main(int argc, char *argv[])
{
  Type_Integer A, B, C = "314159265359";

  cout << "Enter A: "; cin >> A;
  B = A * C;
  cout << A << " times " << C << " equals " << B << '\n';
}
```

A session with this program is given below:

```
Enter A: 12345
12345 times 314159265359 equals 3878296130856855
```

4.6 Implementation Details

The class `Type_Integer` is based upon the GNU multi-precision package "gmp" (see section “Introduction to MP” in *The GNU MP Manual*).

4.6.1 Data Structures

An integer is represented by the class `Type_Integer` which contains a structure of type `MP_INT` as defined by GMP:

```
class Type_Integer {
  MP_INT i;
  ...
}
```

This structure contains one word giving the size of the allocated memory, one word giving the number of word-digits and the sign, and a pointer to the allocated memory. These internals are never accessed directly by the functions of the integer package but only through the functions provided by GMP.

The only exception to this rule is the void constructor, which sets all entries to zero.

4.6.2 Package Initialization

The function `INT_Init` must be executed before any other function of the integer package may be called.

In general this function should not be called directly. It is part of the initialization sequence executed by `STURM_Init` before `STURM_Main()` is entered (see Section 2.2 [Main Program], page 3).

This function calls GMP’s `mp_set_memory_functions` to replace the default memory functions `malloc`, `realloc`, and `free` by `MEM_Allocate`, `MEM_Reallocate`, and `MEM_Free`. Furthermore, it sets up the private variables `Type_Integer::Zero` and `Type_Integer::One` which are used by the functions `ZeroInteger` and `OneInteger`.

4.6.3 Constructive vs. Destructive Functions

GMP requires that any `MP_INT` structure must be initialized before it is used for arithmetic or assignment.

To avoid problems with integers initialized by the void constructor all return arguments must always be initialized before calling a GMP function. For example in

```
inline
Type_Integer
GCD(const Type_Integer &I, const Type_Integer &J)
{
  Type_Integer G = 0;
  mpz_gcd(&G.i, &I.i, &J.i);
  return G;
}
```

the variable `G` is set to some value before being passed on to `mpz_gcd`, otherwise GMP would crash due to the pointer field of the `MP_INT` structure `G.i` being `NULL`.

4.6.4 Assignment

The semantics of the assignment operator for integers should be such that a new object is generated which is distinct from the object being assigned. For example, after the statements

```
I = J;
I += 100;
```

the value of `I` should be greater than the value of `J`, i.e. the destructive operator `+=` should only modify `I` but not `J`.

To accomplish this we considered two possibilities:

1. Having the assignment and copy operators construct a new object.
2. Implementing "destructive" operators non-destructively.

The current implementation takes the second approach. This means that when integers are assigned or copied the `MP_INT` structure is copied so that no allocation is necessary. Destructive operators such as `+=` always construct a new object instead of trying to reuse the memory allocated for the left-hand operand.

Warning: The functions of the GMP work destructively! This means that instead of

```
void operator+=(Type_Integer &I, const Type_Integer &J)
{
  mpz_add(&I.i, &I.i, &J.i);
}
```

one must write

```
void operator+=(Type_Integer &I, const Type_Integer &J)
{
  MP_INT tmp = I.i;
  mpz_init(&I.i);
  mpz_add(&I.i, &tmp, &J.i);
}
```

This preserves the original value of `I`, initializes `I`, and then constructively adds the original value of `I` and the value of `J` and assigns the result to `I`.

It would be interesting to implement the alternate scheme doing constructive assignments and reusing memory in destructive operators and to compare this to the current approach.

5 Rational Numbers

Operations on rational numbers of arbitrary precision are based on the following type:

Type_Rational Arithmetic Type

An arithmetic type ranging over the rational numbers. Its size is not fixed and only restricted by the available memory.

C++ integers and objects of type `Type_Integer` can be mixed with `Type_Rational` in most computations, but conversions always go towards the latter, i.e. it is not possible to convert a `Type_Rational` to e.g. a `Type_Integer`.

The functions and types of the rational package are defined in the header file '`Sturm/Rational.h`'

5.1 Construction and Conversion

When declaring rationals there are two possibilities: A declaration with or without an initializing value.

Declaring and initializing a rational can be done as follows:

`Type_Rational` *R* `=` *I*`;`

This initializes *R* to the value of *I*, where *I* is of type `Type_Integer` or of any C++ integral type (such as `int`, `unsigned long`, etc.).

`Type_Rational` *R*`(`*N*`,` *D*`);`

This initializes *R* to N/D, where *N* and *D* are objects of integral C++ type or of `Type_Integer`. If *D* is zero the result is undefined.

`Type_Rational` *R* `=` *s*`;`

This initializes *R* to the value given in *s*, where *s* is a C++ null-terminated string containing the character representation of a rational in the form

```
INTEGER / INTEGER
```

i.e. an integer (the numerator) followed by a slash and a non-zero integer (the denominator). The slash and the denominator are optional. Both integers may be preceded by optional signs, whitespace between the integers and the slash is ignored. If the denominator is zero the result is undefined.

`Type_Rational` *R* `=` *S*`;`

This initializes *R* to the value of *S*, where *S* is also an object of type `Type_Rational`.

Examples for such initializations are

```
Type_Rational R = 100, S = "-123/456";
Type_Rational T(93, 100), U = R;
```

As with all C++ constructors the alternative form

```
Type_Rational R(...);
```

is also available. Furthermore, these constructors will be used for explicit and implicit type conversions.

The rational numbers are always converted to a normal form with the denominator being positive and numerator and denominator being relatively prime. If the numerator is zero, the denominator is one.

The numerator and the denominator of a rational number, which are of type `Type_Integer`, can be extracted by the functions `Numerator` and `Denominator`.

A declaration without any initialization looks as follows:

```
Type_Rational R;
```

This declares *R* but does not set it to any meaningful value. If this variable is used e.g. as a summand the program's behaviour will be undefined. The only valid use which can be made with an uninitialized rational is assigning it a value by statements such as

```
R = "5/3";
```

This type of declaration exists mainly to allow declaration of *global* objects of type `Type_Rational`. Global objects must not be initialized because it cannot be guaranteed that various setup functions for the STURM package are executed before the constructors for global objects are called. The void constructor does not require prior execution of any such setup function.

5.2 Arithmetic

Arithmetic is done using the standard C++ operators which are meaningful for this type (`+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `/=`) with their standard semantics. The only difference in the operator semantics is that "destructive" operators such as `+=` or `*=` return `void` instead of a rational. This means that one cannot write statements such as `R = (S += T)` or `R = (S *= 2) + J`.

Here is some example code using operators on rationals:

```
Type_Integer R = "100000000000000000/3", S;
int i;

for (i = 0; i < 100; i++)
  R += i;

S = (R / 101) - i;
```

In addition to the operators various functions and procedures are available. (This is currently a minimal set which will be extended as the need arises.)

`Inverse` Computes the inverse of a rational.

`Abs, Sign` Computes the absolute value and the sign of a rational.

5.3 Predicates

Rationals can be compared using the standard C++ operators `==`, `!=`, `<=`, `>=`, `<`, `>` with their standard semantics. The only difference to the standard operators is that the results are of type `Type_Bool` instead of `int`.

Some examples for rational comparisons appear in the code fragment given below:

```
Type_Rational R = 100, S = "8765543322987655 / 443325796674249";

if (R != S) {
  ...
}
if (S == 5) ...
while (R < S)
  R *= 10;
```

For efficient comparison with zero and one use the functions `IsZero`, `IsNonZero`, `IsNonNegative`, `IsNonPositive`, `IsNegative`, `IsPositive`, and `IsOne`. For example

```
for (Type_Rational R = 100; IsPositive(R); R -= 2)
  ...
```

will run faster than

```
for (Type_Rational R = 100; 0 < R; R -= 2)
  ...
```

5.4 Input and Output

Rationals can be read and written using the standard C++ input/output operators `>>`, `<<` with their standard semantics.

For example, you can write

```
#include <iostream.h>
#include "Sturm/Rational.h"

...

  Type_Rational R, S;

  cout << "Please enter two rationals:\n";
  cin >> R;  cin >> S;
  cout << "The sum of " << R << " and " << S " is " << R + S << ".\n";
```

Valid input to this program looks as follows:

```
Please enter two rationals:
123456789 / 10000000000000000000
-3141592
The sum of ...

Please enter two rationals:
+888776543332  100/-2
The sum of ...
```

5.5 Examples

The following program reads a rational, multiplies it with a constant and outputs the result:

```
#include <iostream.h>
#include "Sturm/Kernel.h"
#include "Sturm/Rational.h"

int
STURM_Main(int argc, char *argv[])
{
  Type_Rational R, S, C = "314159265359 / 100000000000";

  cout << "Please enter R: "; cin >> R;
  S = R * C;
  cout << R << " times " << C << " equals " << S << '\n';
}
```

A session with this program looks as follows.

```
Please enter R: 123/45
41/15 times 314159265359/100000000000 equals 12880529879719/1500000000000
```

Note that the input value '`123/45`' is automatically reduced to '`41/15`' by dividing out the common factor 3 from the numerator and the denominator.

5.6 Implementation Details

The STURM class `Type_Rational` is based upon the `Type_Integer` class (see Chapter 4 [Integers], page 18).

5.6.1 Data Structures

A rational is represented by the class `Type_Rational` which contains two objects of `Type_Integer` representing the numerator and the denominator:

```
class Type_Rational {
  Type_Integer num; // The numerator.
  Type_Integer den; // The denominator.
  ...
}
```

5.6.2 Initialization

The rational package does not have an initialization function of its own. As it is based on the integer package, the corresponding initialization function `INT_Init` must be executed before any other function of the rational package may be called (see Section 4.6.2 [Package Initialization], page 22).

6 Double Precision Floating Point Numbers

Operations on double precision floating point numbers are based on the following type:

Type_Float64 Arithmetic Type

An arithmetic type ranging over 64 bit floating point numbers. `Type_Float64` corresponds to double precision floating point number as specified in the IEEE standard 754. In the sequel we will simply say floating point number instead of double precision floating point number.

Operations which are defined for floating point numbers are also defined for `Type_Float64`. Additional functions for `Type_Float64` are declared in the header file '`Sturm/Float64.h`'.

6.1 Construction and Conversion

A floating point number is declared as follows:

`Type_Float64` *A*;

The value of *A* is undefined.

6.2 Constants

Constants of `Type_Float` are implemented as functions which take no arguments and return a floating point number.

A `= MinFloat64();`

A is the smallest positive floating point number.

A `= MaxFloat64();`

A is the largest (finite) floating point number.

A `= InfFloat64();`

A is infinity.

A `= NanFloat64();`

A is not a number (Nan).

6.3 Arithmetic

Arithmetic is done using the standard C++ operators `+, -, *, /, +=, -=, *=, /=` with minor deviations from their standard semantics. The definition of the operations corresponds to the IEEE standard 754.

In addition to the operators the following functions and procedures are available:

In the description below A, B and the results of all operations are objects of `Type_Float64`.

`Abs` Computes the absolute value.

`Sqrt` Computes the square root.

`Sign` Computes the sign.

`Ceil` Round up to the nearest integer value.

`Floor` Round down to the nearest integer value.

`Max` Compute the maximum of two floating point numbers.

`Min` Compute the minimum of two floating point numbers.

`GetExponent, SetExponent`
Get and Set the exponent of the binary representation of A. These operations can be used for efficient multiplication of a floating point number by a power of two.

`Float64` Compute a floating point number from sign, fraction and exponent.

6.4 Predicates

Floating Point Numbers can be compared using the standard C++ operators `==, !=, <=, >=,<, >` with the semantics as specified by the IEEE standard 754.

Besides the comparison operators the following predicates are defined:

`IsNan` Tests whether a floating point number is Nan.

`IsInf` Tests whether a floating point number is plus or minus infinity.

`IsDenormalized`
Tests whether a floating point number is denormalized.

6.5 Input and Output

Floating Point Numbers can be read and written using the standard C++ input/output operators `>>, <<` with their standard semantics. In addition the functions `Read` and `Write` are provided, which accept symbolic names for the smallest and largest floating point number, for infinity and for Nan. Further, `Read` accepts a tuple of integers for the sign, low and high bits of the fraction and exponent. The function `Dump` prints information on the internal representation of a floating point number.

```
Type_Float64 A,B,C;

cout << "Enter three floating point numbers:\n";
Read(A); Read(B); Read(C);

cout << "\nThe numbers are:\n";
cout << A << " " << B << " " << C<< "\n";
cout << "\nInternal representation of the last number:\n";
Dump(C);
```

A session with this program is given below:

```
Enter three floating point numbers:
Max
3.7
(-1 7 15 0)

The numbers are:
1.79769e+308 3.7 -1.00001

Internal representation of the last number:
Exponent    0 (1023-1023)
Sign        negative
Significand 1.0000000000000000111100000000000000000000000000000111
Value       non-zero float -1.00001
```

6.6 Example Programs

The following program reads a floating point number, devides it by 10 and prints an internal representation of the result. This is done twice, where once the rounding mode is first set to round up and then set to round down. Whether or not the result of the real number division is a floating point number or not the printed floating numbers differ in the least significant bit.

```
#include <iostream.h>
#include "Sturm/Kernel.h"
#include "Sturm/Float64.h"

int
STURM_Main(int argc, char *argv[])
{
  Type_Float64 A,Bup,Bdown;

  cout << "Enter a floating point number:\n"; cin >> A;

  FPU_Up();
  Bup = A / 10.0;
  cout << "\nResult of division by 10 when rounded up:\n";
  Dump(Bup);

  FPU_Down();
  Bdown = A / 10.0;
  cout << "\nResult of division by 10 when rounded down:\n";
  Dump(Bdown);

  if(Bup == Bdown) cout << "\nThe results are equal\n";
  else             cout << "\nThe results are not equal\n";
}
```

A session with this program is given below:

```
Enter a floating point number:
7.5
```

```
Result of division by 10 when rounded up:
Exponent    -1 (1022-1023)
Sign        positive
Significand 1.1000000000000000000000000000000000000000000000000000
Value       non-zero float 0.75

Result of division by 10 when rounded down:
Exponent    -1 (1022-1023)
Sign        positive
Significand 1.1000000000000000000000000000000000000000000000000000
Value       non-zero float 0.75

The results are equal
```

In the example above, during the division no rounding error occurred because the real number 0.75 is a floating point number. If the input is e.g. 1, then the result is 0.1, which is not a floating point number:

```
Enter a floating point number
1

Result of division by 10 when rounded up:
Exponent    -4 (1019-1023)
Sign        positive
Significand 1.1001100110011001100110011001100110011001100110011010
Value       non-zero float 0.1

Result of division by 10 when rounded down:
Exponent    -4 (1019-1023)
Sign        positive
Significand 1.1001100110011001100110011001100110011001100110011001
Value       non-zero float 0.1

The results are not equal
```

6.7 Implementation Details

A floating point number is implemented by `typedef double Type_Float64`.

7 Double Precision Intervals

Operations on intervals are based on the following type:

Type_Interval64 Arithmetic Type
An arithmetic type ranging over real intervals.

The possible values of a double precision interval are

- The empty set.
- The set of real numbers r, where A `<=` r `<=` B for some finite `Type_Float64` numbers A, B.
- The set of real numbers r, where A `<=` r for some finite `Type_Float64` number A.
- The set of real numbers r, where r `<=` B for some finite `Type_Float64` number B.
- The set of all real numbers.

In the sequel we will simply say floating point interval or just interval instead of double precision floating point interval.

The functions and types of the interval package are declared in the header file '`Sturm/Interval64.h`'.

7.1 Construction and Conversion

A floating point interval is declared as follows:

`Type_Interval64` *A*`;`
The value of A is undefined.

7.2 Constants

The following two constant functions are available:

`EmptyInterval64()`
The empty interval

`AllInterval64()`
The interval of all real numbers

7.3 Arithmetic

Arithmetic is done using the standard C++ operators `+, -, *, /, +=, -=, *=, /=` with minor deviations from their standard semantics. The computed interval is the `Type_Interval64` number which contains the result of the corresponding real interval operation with smallest cardinality. One operand may be of type `Type_Float64` and the following cases have to be distinguished:

- If the floating point number operand is finite and not Nan, then it is interpreted as singleton interval.
- If the floating point number operand is plus infinity, then it is interpreted as the interval which contains all real numbers greater or equal to the largest representable floating point number. Minus infinity is treated analogously.
- If the floating point number operand is Nan, then it is interpreted as the interval which contains all real numbers.

This definition is motivated as follows: If the result of a floating point computation is infinity, then either overflow or division by zero occurred. Division by zero is definitely considered as an error. (By default the division by zero exception is not masked, see Section 3.3.2 [Exception Handling (FPU)], page 14). Thus the floating point number infinity must not be interpreted as a number which is greater than all real numbers but as a real number which is greater than the largest representable floating point number.

A Nan can be generated for example if two positive infinite floating point numbers are subtracted. An infinite floating point number is interpreted as a real number, hence Nan must be interpreted as a real number with arbitrary value.

Interval operands of the arithmetic operations must not be empty. It would be reasonable to assume that the result of an arithmetic operation, where one or both operands are empty, is the empty interval. However, this would require additional checks and degrade performance. This lack of generality should not introduce serious problems. It seems that usually, if an interval arithmetic algorithm calls a function which may return an empty interval, an explicit check whether the interval is empty or not is required anyways.

Finally, we have to mention that destructive operators such as `+=` or `=` return `void` instead of an interval.

In addition to the operators various functions and procedures are available.

`Lower, Upper`
: Compute the lower, upper bound of an interval.

`Mig, Mag`
: Compute the mignitude, magnitude of an interval.

`Abs`
: Compute the absolute value of an interval.

`Width, Rad`
: Compute the width, radius of an interval.

`Mid`
: Compute the midpoint of an interval.

`Sqr, Sqrt`
: Compute the square, square root of an interval.

`Hull`
: Compute the hull of two intervals, of a floating point number and an interval or of two floating point numbers.

`Intersect`
: Compute the intersection of two intervals.

7.4 Predicates

Intervals can be compared using the standard C++ operators `==`, `!=`, `<=`, `>=`, `<`, `>` with their standard semantics. For example an interval A is greater than an interval B if all elements of A are greater than all elements of B. It is greater or equal if all elements of A are greater or equal than all elements of B. Except for `==` and `!=` it is an error if one of the operands is empty.

In all comparison operators one of the operands may be of type `Type_Float64` and the following cases have to be distinguished:

- If the floating point number operand is finite and not Nan, then it is interpreted as singleton interval.
- If the floating point number operand is plus infinity, then it is different from all intervals and greater than all right bounded intervals. Any comparison except `!=` of plus infinity and a right open interval returns `FALSE`. Minus infinity is treated analogously.
- It is an error if the floating point number operand is Nan.

In addition to the comparison operators the following predicates are available:

`IsEmpty` Decide whether an interval is empty.

`IsElement`
: Decide whether a floating point number is element of an interval.

`IsSubset` Decide whether an interval is subset of an other interval.

`IsInside` Decide whether an interval is contained in the interior of an other interval.

`IsDisjoint`
: Decide whether two intervals are disjoint.

7.5 Input and Output

Intervals can be read and written using the standard C++ input/output operators `>>`, `<<` with their standard semantics.

For example, you can write

```
#include <iostream.h>

...

  Type_Interval64 A,B,C,D;

  cout << "Please enter four intervals:\n";
  cin >> A >> B >> C >> D;
```

A valid input to such a program is

```
Please enter three intervals:
[-3.7, 4.9]
```

```
2.0
[]
[-Inf,1.0]
```

where A is the interval [-3.7, 4.9], B is the singleton interval containing 2.0, C is the empty interval, and D is the interval which contains all real numbers less than 1. (Rounding errors in the input routine cause that A is only an approximation of [-3.7, 4.9].)

7.6 Example Programs

The following program reads two intervals, multiplies them and prints the result:

```
#include <iostream.h>
#include "Sturm/Kernel.h"
#include "Sturm/Interval64.h"

int
STURM_Main(int argc, char *argv[])
{
  Type_Interval64 A, B;

  cout << "Enter two intervals:\n":
  cin >> A >> B;

  cout << "\nThe product is " << A * B << ".\n";
}
```

A session with this program is given below:

```
Enter two intervals:
[-3,4]
[ 2,5]

The product is [-15 , 20].
```

7.7 Implementation Details

The internal representation of `Type_Interval64` is a pair of `Type_Float64`. Open intervals are represented by infinity at the corresponding endpoint, the empty interval has a Nan as its left endpoint and the value 1.239857e300 as its right endpoint.

Appendix A Function Specifications

A.1 Float64

A.1.1 Bitlevel

GetExponent Function on Type_Float64

Get the normalized exponent. E = `GetExponent(`A`)`

A: `Type_Float64`

E: `int` E The normailzed exponent of A.

Description

The normalized exponent of A is the exponent of the binary representation of A minus the offset 1023. It ranges between -1023 and 1024.

Header File

'`Sturm/Float64.h`'

See Also

`SetExponent` , Chapter 6 [Floating Point Numbers], page 30.

IsDenormalized Function on Type_Float64

Decide whether a `Type_Float64` value is denormalized: R = `IsDenormalized(`A`)`

A: `Type_Float64`

R: `Type_Bool`

`TRUE` if A is denormalized, `FALSE` otherwise.

Description

According to the IEEE Standard 754 a `Type_Float64` value A is denormalized, if the exponent of A is zero and the fraction of A is nonzero.

Header File

'`Sturm/Float64.h`'

See Also

`IsInf`, `IsNan` , Chapter 6 [Floating Point Numbers], page 30.

IsInf Function on Type_Float64

Decide whether a `Type_Float64` value is infinite: R = `IsInf(`A`)`

A: `Type_Float64`

R: `Type_Bool`

`TRUE` if A is plus or minus infinity, `FALSE` otherwise.

Description
According to the IEEE Standard 754 a `Type_Float64` number A is infinite if the exponent of A is 2047 and the fraction of A is zero.

Header File
'`Sturm/Float64.h`'

See Also
`IsNan`, `IsDenormalized` , Chapter 6 [Floating Point Numbers], page 30.

IsNan — Function on Type_Float64

Decide whether a `Type_Float64` value is not a number: R `= IsNan(`A`)`

A: `Type_Float64`

R: `Type_Bool`

`TRUE` if A is not a number, `FALSE` otherwise.

Description
According to the IEEE Standard 754 a `Type_Float64` value A is not a number if the exponent of A is 2047 and the fraction of A is nonzero.

Header File
'`Sturm/Float64.h`'

See Also
`IsInf`, `IsDenormalized` , Chapter 6 [Floating Point Numbers], page 30.

SetExponent — Procedure on Type_Float64

Set the exponent. `SetExponent(`A`, `E`)`

A: `Type_Float64 &`

E: `int` The normalized exponent for A.

Description
The exponent of the binary representation of A is set to E `+ 1023`.

Header File
'`Sturm/Float64.h`'

See Also
`GetExponent` , Chapter 6 [Floating Point Numbers], page 30.

A.1.2 Constants

InfFloat64 Function on Type_Float64

description: R = `InfFloat64()`

R: `Type_Float64`

Infinity.

Header File

'`Sturm/Float64.h`'

See Also

`MinFloat64`, `MaxFloat64`, `NanFloat64`, `OneFloat64`, `ZeroFloat64` , Chapter 6 [Floating Point Numbers], page 30.

MaxFloat64 Function on Type_Float64

description: R = `MaxFloat64()`

R: `Type_Float64`

The largest finite floating point number.

Header File

'`Sturm/Float64.h`'

See Also

`MinFloat64`, `InfFloat64`, `NanFloat64`, `OneFloat64`, `ZeroFloat64` , Chapter 6 [Floating Point Numbers], page 30.

MinFloat64 Function on Type_Float64

description: R = `MinFloat64()`

R: `Type_Float64`

The smallest positive floating point number.

Header File

'`Sturm/Float64.h`'

See Also

`MaxFloat64`, `InfFloat64`, `NanFloat64`, `OneFloat64`, `ZeroFloat64` , Chapter 6 [Floating Point Numbers], page 30.

NanFloat64 Function on Type_Float64

description: R = `NanFloat64()`

R: `Type_Float64`

The Nan with positive sign and all but the least significant fraction bits zero.

Header File

'`Sturm/Float64.h`'

See Also
`MinFloat64`, `MaxFloat64`, `InfFloat64`, `OneFloat64`, `ZeroFloat64` , Chapter 6 [Floating Point Numbers], page 30.

OneFloat64

Function on Type_Float64

description: R = `OneFloat64()`

R: `Type_Float64`
1.0

Header File
'`Sturm/Float64.h`'

See Also
`MinFloat64`, `MaxFloat64`, `InfFloat64`, `NanFloat64`, `ZeroFloat64` , Chapter 6 [Floating Point Numbers], page 30.

ZeroFloat64

Function on Type_Float64

description: R = `ZeroFloat64()`

R: `Type_Float64`
0.0

Header File
'`Sturm/Float64.h`'

See Also
`MinFloat64`, `MaxFloat64`, `InfFloat64`, `NanFloat64`, `OneFloat64` , Chapter 6 [Floating Point Numbers], page 30.

A.1.3 Constructors

Float64

Function on Type_Float64

Construct a floating point number from binary representation: `R = Float64(`S`,` L`,` H`,` E`)`

S: `short` Sign, `+1` or `-1`.

L: `unsigned`
The 32 low significant bits of the fraction.

H: `unsigned`
The 20 high significant bits of the fraction.

E: `int` The normalized exponent, `-1023 ... 1024`.

R: `Type_Float64`
The corresponding floating point number.

Header File
'`Sturm/Float64.h`'

See Also
`Dump` , Chapter 6 [Floating Point Numbers], page 30.

A.1.4 Functions

Abs Function on Type_Float64

Compute the absolute value: R = `Abs(`A`)`

A: `Type_Float64`
Not Nan.

R: `Type_Float64`
The absolute value of A.

Header File
'`Sturm/Float64.h`'

See Also
`Sign` , Chapter 6 [Floating Point Numbers], page 30.

Ceil Function on Type_Float64

Compute the smallest integer which is greater or equal A. R = `Ceil(`A`)`

A: `Type_Float64`
Not Nan.

A: `Type_Float64`
The smallest integer which greater or equal A.

Header File
'`Sturm/Float64.h`'

See Also
`Floor` , Chapter 6 [Floating Point Numbers], page 30.

Floor Function on Type_Float64

Compute the largest integer which is less or equal A. R = `Floor(`A`)`

A: `Type_Float64`
Not Nan.

R: `Type_Float64`
The largest integer which less or equal A.

Header File
'`Sturm/Float64.h`'

See Also
`Ceil` , Chapter 6 [Floating Point Numbers], page 30.

Max Function on Type_Float64

Compute the maximum of two floating point numbers. C = `Max(`A`, `B`)`

A, B: `Type_Float64`

Not Nan.

C: `Type_Float64`

The maximum of A and B.

Header File

'`Sturm/Float64.h`'

See Also

`Min` , Chapter 6 [Floating Point Numbers], page 30.

Min Function on Type_Float64

Compute the minimum of two floating point numbers. C = `Min(`A`, `B`)`

A, B: `Type_Float64`

Not Nan.

C: `Type_Float64`

The minimum of A and B.

Header File

'`Sturm/Float64.h`'

See Also

`Max` , Chapter 6 [Floating Point Numbers], page 30.

Sign Function on Type_Float64

Compute the sign: R = `Sign(`A`)`

A: `Type_Float64`

Not Nan.

R: `Type_Float64`

The sign of A.

Header File

'`Sturm/Float64.h`'

See Also

`Abs` , Chapter 6 [Floating Point Numbers], page 30.

Sqrt Function on Type_Float64

Compute the square root: R = `Sqrt(`A`)`

A: `Type_Float64`

Not Nan.

R: `Type_Float64`

The square root of A.

Header File
'`Sturm/Float64.h`'

See Also
Chapter 6 [Floating Point Numbers], page 30.

A.1.5 IO

Dump Procedure on Type_Float64

Print internal representation. `Dump(`A`)`

A: `Type_Float64`

Description
Print information on the internal representation of A, i.e. sign, fraction bits, exponent, and the interpretation of A as a real number.

Header File
'`Sturm/Float64.h`'

See Also
`Float64`, `Read`, `Write` , Chapter 6 [Floating Point Numbers], page 30.

Read Procedure on Type_Float64

Read floating point number from standard input. `Read(`A`)`

A: `Type_Float64 &`

Description
In addition to the standard IO representation of floating point numbers the strings `Smallest`, `Infinity`, `Largest` and `Nan` are allowed. A string (*sign*,*low*,*high*,*exponent*) is interpreted as by the function `Float64`.

Header File
'`Sturm/Float64.h`'

See Also
`Float64`,`Read`, `Write` , Chapter 6 [Floating Point Numbers], page 30.

Read Procedure on Type_Float64

Read floating point number from a stream. `Read(`A`)`

is: `istream &`

A: `Type_Float64 &`

Description
In addition to the standard IO representation of floating point numbers the strings `Min`, `Max`, `Inf` and `Nan` for smallest positive floating point number, largest floating point number, infinity and Nan are allowed. A string (*sign*,*low*,*high*,*exponent*) is interpreted as by the function `Float64`.

Header File
'`Sturm/Float64.h`'

See Also
`Float64`,`Read`, `Write` , Chapter 6 [Floating Point Numbers], page 30.

Write Procedure on Type_Float64
Write floating point number to standard output. `Write(`*A*`)`

A: `Type_Float64`

Description
If *A* is plus or minus infinty, smallest, largest or Nan, a corresponding string is written. Otherwise *A* is written in the standard IO format of `double`.

Header File
'`Sturm/Float64.h`'

See Also
`Float64`,`Read`, `Write` , Chapter 6 [Floating Point Numbers], page 30.

Write Procedure on Type_Float64
Write floating point number to a stream. `Write(`*A*`)`

os: `ostream &`

A: `Type_Float64`

Description
If *A* is plus or minus infinty, smallest, largest or Nan, a corresponding string is written. Otherwise *A* is written in the standard IO format of `double`.

Header File
'`Sturm/Float64.h`'

See Also
`Float64`,`Read`, `Write` , Chapter 6 [Floating Point Numbers], page 30.

A.2 Int

A.2.1 Constants

A.3 Integer

A.3.1 Arithmetic

Abs Function on Type_Integer

Integer absolute value: A = `Abs(`I`)`

I: `const Type_Integer &`

A: `Type_Integer`

The absolute value of I.

Header File
'`Sturm/Integer.h`'

See Also
`Sign` on `Type_Integer` , Chapter 4 [Integers], page 18.

GCD Function on Type_Integer

Integer greatest common divisor: G = `GCD(`I`,` J`)`

I, J: `const Type_Integer &`

G: `Type_Integer`

The greatest common divisor of I and J.

Header File
'`Sturm/Integer.h`'

See Also
`GCDExtended` on `Type_Integer` , Chapter 4 [Integers], page 18.

GCDExtended Function on Type_Integer

Integer greatest common divisor and cofactors: G = `GCDExtended(`I`,` J`,` G`,` S`,` T`)`

I, J: `const Type_Integer &`

G: `Type_Integer &`

The greatest common divisor of I and J.

S, T: `Type_Integer &`

The cofactors for I and J, resp. The cofactors are such that $G = I\ S$ + J T.

Header File
'`Sturm/Integer.h`'

See Also
`GCD` on `Type_Integer` , Chapter 4 [Integers], page 18.

QuotRem

Function on Type_Integer

Integer quotient and remainder: `QuotRem(`I`, `J`, `Q`, `R`)`

I, J: `const Type_Integer &`
: The dividend and the divisor; J non-zero.

Q, R: `Type_Integer &`
: The quotient and the remainder of I and J.

Description
Q and R are such that I `=` Q `*` J `+` R where

- the sign of Q is the product of the signs of I and J,
- the sign of R is equal to the sign of I, and
- the absolute value of R is less than the absolute value of J.

Remarks
The statement

```
QuotRem(I, J, Q, R);
```

in general will be faster than the semantically equivalent sequence

```
Q = I / J; R = I % J;
```

Header File
'`Sturm/Integer.h`'

See Also
Chapter 4 [Integers], page 18.

Sign

Function on Type_Integer

Integer sign: s `= Sign(`I`)`

I: `const Type_Integer &`

s: `long` The sign of I.

Header File
'`Sturm/Integer.h`'

See Also
`Abs` on `Type_Integer` , Chapter 4 [Integers], page 18.

+

Arithmetic operator on Type_Integer

Add integers: R `=` I `+` J

I, *J*: `const Type_Integer &`
R: `Type_Integer`
The sum of *I* and *J*.

Header File
'`Sturm/Integer.h`'

See Also
Chapter 4 [Integers], page 18.

`+=` Arithmetic operator on Type_Integer
Add and assign an integer: *I* `+=` *J*

I: `Type_Integer &`
J: `const Type_Integer &`

Description
The values of *I* and *J* are added and the result is stored in *I*.

Header File
'`Sturm/Integer.h`'

See Also
Chapter 4 [Integers], page 18.

`+=` Arithmetic operator on Type_Integer
Add and assign a C integer to an integer: *I* `+=` *j*

I: `Type_Integer &`
j: `long`

Description
The values of *I* and *j* are added and the result is stored in *I*.

Header File
'`Sturm/Integer.h`'

See Also
Chapter 4 [Integers], page 18.

`+=` Arithmetic operator on Type_Integer
Add and assign a C integer to an integer: *I* `+=` *j*

I: `Type_Integer &`
j: `unsigned long`

Description
The values of *I* and *j* are added and the result is stored in *I*.

Header File
'Sturm/Integer.h'

See Also
Chapter 4 [Integers], page 18.

/ — Arithmetic operator on Type_Integer

Divide integers: R `=` I `/` J

I, J: `const Type_Integer &`
: J non-zero.

R: `Type_Integer &`
: The integer quotient of I and J. For a detailed specification of the quotient see the description of the function `QuotRem`.

Header File
'Sturm/Integer.h'

See Also
`QuotRem` on `Type_Integer` , Chapter 4 [Integers], page 18.

/= — Arithmetic operator on Type_Integer

Divide and assign integers: I `/=` J

I: `Type_Integer &`

J: `const Type_Integer &`
: A non-zero integer.

Description
The value of I is divided by the value of J and the result is stored in I. For a detailed specification of division see the description of the function `QuotRem`.

Header File
'Sturm/Integer.h'

See Also
`QuotRem` on `Type_Integer` , Chapter 4 [Integers], page 18.

/= — Arithmetic operator on Type_Integer

Divide an integer by a C integer and assign: I `/=` j

I: `Type_Integer &`

j: `long` A non-zero integer.

Description
The value of I is divided by the value of j and the result is stored in j. For a detailed specification of division see the description of the function `QuotRem`.

Header File
'Sturm/Integer.h'

See Also
`QuotRem` on `Type_Integer` , Chapter 4 [Integers], page 18.

/= Arithmetic operator on Type_Integer

Divide an integer by a C integer and assign: *I* `/=` *j*

I: `Type_Integer &`

j: `unsigned long`

A non-zero integer.

Description
The value of *I* is divided by the value of *j* and the result is stored in *j*. For a detailed specification of division see the description of the function `QuotRem`.

Header File
'Sturm/Integer.h'

See Also
`QuotRem` on `Type_Integer` , Chapter 4 [Integers], page 18.

- Arithmetic operator on Type_Integer

Negate integer: *R* `= -`*I*

I: `const Type_Integer &`

R: `Type_Integer`

The negative of *I*.

Header File
'Sturm/Integer.h'

See Also
Chapter 4 [Integers], page 18.

% Arithmetic operator on Type_Integer

Compute the modulo of integers: *I* `%` *J*

I, *J*: `const Type_Integer &`

J non-zero.

R: `Type_Integer &`

The remainder of the integer division of *I* and *J*. For a detailed specification of division see the description of the function `QuotRem`.

Header File
'Sturm/Integer.h'

See Also
`QuotRem` on `Type_Integer` , Chapter 4 [Integers], page 18.

%= Arithmetic operator on Type_Integer

Compute modulo of integers and assign: *I* `%=` *J*

I: `Type_Integer &`

J: `const Type_Integer &`
A non-zero integer.

Description
The value of *I* is divided by the value of *J* and the remainder of the division is stored in *I*. For a detailed specification of division see the description of the function `QuotRem`.

Header File
'`Sturm/Integer.h`'

See Also
`QuotRem` on `Type_Integer` , Chapter 4 [Integers], page 18.

%= Arithmetic operator on Type_Integer

Compute the modulo of an integer and a C integer and assign: *I* `%=` *j*

I: `Type_Integer &`

j: `long` A non-zero integer.

Description
The value of *I* is divided by the value of *j* and the remainder of the division is stored in *I*. For a detailed specification of division see the description of the function `QuotRem`.

Header File
'`Sturm/Integer.h`'

See Also
`QuotRem` on `Type_Integer` , Chapter 4 [Integers], page 18.

%= Arithmetic operator on Type_Integer

Compute the modulo of an integer and a C integer and assign: *I* `%=` *j*

I: `Type_Integer &`

j: `unsigned long`
A non-zero integer.

Description
The value of *I* is divided by the value of *j* and the remainder of the division is stored in *I*. For a detailed specification of division see the description of the function `QuotRem`.

Header File
'Sturm/Integer.h'

See Also
`QuotRem` on `Type_Integer` , Chapter 4 [Integers], page 18.

* Arithmetic operator on Type_Integer

Multiply integers: R `=` I `*` J

I, J: `const Type_Integer &`

R: `Type_Integer &`

The product of I and J.

Header File
'Sturm/Integer.h'

See Also
Chapter 4 [Integers], page 18.

*= Arithmetic operator on Type_Integer

Multiply and assign an integer: I `*=` J

I: `Type_Integer &`

J: `const Type_Integer &`

Description
The values of I and J are multiplied and the result is stored in I.

Header File
'Sturm/Integer.h'

See Also
Chapter 4 [Integers], page 18.

*= Arithmetic operator on Type_Integer

Multiply an integer by a C integer and assign: I `*=` j

I: `Type_Integer &`

j: `long`

Description
The values of I and j are multiplied and the result is stored in I.

Header File
'Sturm/Integer.h'

See Also
Chapter 4 [Integers], page 18.

***=** Arithmetic operator on Type_Integer

Multiply an integer by a C integer and assign: *I* `*=` *j*

I: `Type_Integer &`

j: `unsigned long`

Description
The values of *I* and *j* are multiplied and the result is stored in *I*.

Header File
'`Sturm/Integer.h`'

See Also
Chapter 4 [Integers], page 18.

– Arithmetic operator on Type_Integer

Decrement an integer: *I*`--`

I: `Type_Integer &`

Description
The value of *I* is decremented by 1.

Header File
'`Sturm/Integer.h`'

See Also
prefix `--` on `Type_Integer` , Chapter 4 [Integers], page 18.

++ Arithmetic operator on Type_Integer

Increment an integer: *I*`++`

I: `Type_Integer &`

Description
The value of *I* is incremented by 1.

Header File
'`Sturm/Integer.h`'

See Also
prefix `++` on `Type_Integer` , Chapter 4 [Integers], page 18.

– Arithmetic operator on Type_Integer

Decrement an integer: `--`*I*

I: `Type_Integer &`

Description
The value of *I* is decremented by 1.

Header File
'`Sturm/Integer.h`'

See Also
postfix `--` on `Type_Integer` , Chapter 4 [Integers], page 18.

`++` Arithmetic operator on Type_Integer

Increment an integer: `++`*I*

I: `Type_Integer &`

Description
The value of *I* is incremented by 1.

Header File
'`Sturm/Integer.h`'

See Also
postfix `++` on `Type_Integer` , Chapter 4 [Integers], page 18.

`-` Arithmetic operator on Type_Integer

Subtract integers: *R* `=` *I* `-` *J*

I, *J*: `const Type_Integer &`

R: `Type_Integer`

The difference of *I* and *J*.

Header File
'`Sturm/Integer.h`'

See Also
Chapter 4 [Integers], page 18.

`-=` Arithmetic operator on Type_Integer

Subtract and assign an integer: *I* `-=` *J*

I: `Type_Integer &`

J: `const Type_Integer &`

Description
The value *J* is subtracted from the value of *I* and the result is stored in *I*.

Header File
'Sturm/Integer.h'

See Also
Chapter 4 [Integers], page 18.

-= Arithmetic operator on Type_Integer

Subtract a C integer from an integer and assign: *I* `-=` *j*

I: `Type_Integer &`

j: `long`

Description
The value *j* is subtracted from the value of *I* and the result is stored in *I*.

Header File
'Sturm/Integer.h'

See Also
Chapter 4 [Integers], page 18.

-= Arithmetic operator on Type_Integer

Subtract a C integer from an integer and assign: *I* `-=` *j*

I: `Type_Integer &`

j: `unsigned long`

Description
The value *j* is subtracted from the value of *I* and the result is stored in *I*.

Header File
'Sturm/Integer.h'

See Also
Chapter 4 [Integers], page 18.

A.3.2 Assignment

= Operator on Type_Integer

Assign integers: *I* `=` *J*

I: `Type_Integer`

j: `const Type_Integer &`

Description
I is set to the value of *J*.

Header File
'Sturm/Integer.h'

See Also
Chapter 4 [Integers], page 18.

= Operator on Type_Integer

Assign a C string to an integer: I `=` s

I: `Type_Integer`

s: `char *` A null-terminated string.

Description
I is set to the value given in s, where s is a C++ null-terminated string containing the character representation of an integer in the form

```
[+|-]DIGIT DIGIT*
```

i.e. an optional sign followed by one or more decimal digits.

Header File
'Sturm/Integer.h'

See Also
Chapter 4 [Integers], page 18.

= Operator on Type_Integer

Assign a C integer to an integer: I `=` j

I: `Type_Integer`

j: `long`

Description
I is set to the value of j.

Header File
'Sturm/Integer.h'

See Also
Chapter 4 [Integers], page 18.

= Operator on Type_Integer

Assign a C integer to an integer: I `=` j

I: `Type_Integer`

j: `unsigned long`

Description
I is set to the value of *j*.

Header File
'`Sturm/Integer.h`'

See Also
Chapter 4 [Integers], page 18.

A.3.3 Constructors

Initialize Function on Type_Integer

Initialize an integer: `Initialize(`*I*`)`

I: `Type_Integer *`

A pointer to the integer to be initialized.

Description
The integer is set to a state identical to the one set by the `void` constructor. This means that the initialized integer must not be used in computations but may only be assigned a value.

Header File
'`Sturm/Integer.h`'

See Also
`Type_Integer(void)` , Chapter 4 [Integers], page 18.

Type_Integer Private Constructor on Type_Integer

Initialize an integer from an internal representation: `Type_Integer` *I*(*j*)

j: `const MP_INT &`

The internal representation of an integer.

Description
The internal representation of *I* is set to *j*.

Warnings
This function is for internal use only.

Header File
'`Sturm/Integer.h`'

See Also
Chapter 4 [Integers], page 18, section "Introduction to MP" in *The GNU MP Manual*.

Type_Integer Constructor on Type_Integer

Initialize an integer from a string: `Type_Integer` $I(s)$

s: `char *`

Description

I is set to the value given in s, where s is a C`++` null-terminated string containing the character representation of an integer in the form

```
[+|-]DIGIT DIGIT*
```

i.e. an optional sign followed by one or more decimal digits.

Header File

'`Sturm/Integer.h`'

See Also

Chapter 4 [Integers], page 18.

Type_Integer Constructor on Type_Integer

Initialize an integer from a C integer: `Type_Integer` $I(j)$

j: `long`

Description

I is set to the value of j.

Header File

'`Sturm/Integer.h`'

See Also

Chapter 4 [Integers], page 18.

Type_Integer Constructor on Type_Integer

Initialize an integer from a C integer: `Type_Integer` $I(j)$

j: `unsigned long`

Description

I is set to the value of j.

Header File

'`Sturm/Integer.h`'

See Also

Chapter 4 [Integers], page 18.

Type_Integer Constructor on Type_Integer

Initialize an integer: `Type_Integer` *I*

Description

Default constructor for an object of type `Type_Integer`.

Warnings

I is *not* set to any meaningful value (such as 0)! You must assign a value to the variable before using it.

Header File

'`Sturm/Integer.h`'

See Also

`Initialize` on `Type_Integer` , Chapter 4 [Integers], page 18.

OneInteger Function

The integer representation of 1: *I* `= OneInteger()`

I: `const Type_Integer &`

The integer representation of 1.

Remarks

This function returns a constant reference to a variable of type `Type_Integer` containing the representation of 1. Thus, construction of a new 1 integer is avoided.

Header File

'`Sturm/Integer.h`'

See Also

`ZeroInteger` , Chapter 4 [Integers], page 18.

RandomInteger Function

A random integer: *I* `= RandomInteger(`*s*`)`

s: `unsigned long`

The maximal size of the random integer in bits. (Will be rounded up to the next word size.)

I: `Type_Integer`

A random integer.

Remarks

In the current implementation no special features of randomness are guaranteed ...

Header File

'`Sturm/Integer.h`'

See Also

Chapter 4 [Integers], page 18.

ZeroInteger Function

The integer representation of 0: *I* = `ZeroInteger()`

I: `const Type_Integer &`

The integer representation of 0.

Remarks

This function returns a constant reference to a variable of type `Type_Integer` containing the representation of 0. Thus, construction of a new 0 integer is avoided.

Header File

'`Sturm/Integer.h`'

See Also

`OneInteger` , Chapter 4 [Integers], page 18.

A.3.4 IO

`>>` I/O operator on Type_Integer

Read an integer: *is* `>>` *I*

is: `istream &`

A C`++` input stream.

I: `Type_Integer &`

Description

I is set to an integer value read from the input stream. The input format consists of

- an optional sign ('`+`' or '`-`'), followed by
- an arbitrary number of decimal digits ('`0`' ... '`9`').

Leading spaces and spaces between the sign and the first digit are ignored.

Header File

'`Sturm/Integer.h`'

See Also

`<<` on `Type_Integer` , Chapter 4 [Integers], page 18.

`<<` I/O operator on Type_Integer

Write an integer: *os* `<<` *I*

os: `ostream &`

A C`++` output stream.

I: `const Type_Integer &`

Description
I is written to the output stream. The output format consists of

- an minus sign (‘`-`’) if *I* is negative, followed by
- a sequence decimal digits (‘`0`’ ... ‘`9`’) giving the decimal representation of the value of *I*.

Header File
‘`Sturm/Integer.h`’

See Also
`>>` on `Type_Integer` , Chapter 4 [Integers], page 18.

A.3.5 Initialization

INT_Init Function on Type_Integer

Initialize the integer package: `INT_Init()`

Description
This functions does various initializations for the integer package. It must be executed before any other function of the integer package is called. It requires the memory management package.

Remarks
In general this function should not be called explicitly by application programs. It is part of the initialization sequence done by `STURM_Init`.

Header File
‘`Sturm/Integer.h`’

See Also
`STURM_Init` , Chapter 4 [Integers], page 18.

A.3.6 Predicates

IsNegative Predicate on Type_Integer

Compare an integer with zero (less): *f* `= IsNegative(`*I*`)`

I: `const Type_Integer &`

f: `Type_Bool`

`TRUE` if *I* is less than zero; `FALSE` otherwise.

Header File
‘`Sturm/Integer.h`’

See Also
Chapter 4 [Integers], page 18.

IsNonNegative
Predicate on Type_Integer

Compare an integer with zero (not less): f = `IsNonNegative(`I`)`

I: `const Type_Integer &`

f: `Type_Bool`

`TRUE` if I is equal to or greater than zero; `FALSE` otherwise.

Header File
'`Sturm/Integer.h`'

See Also
Chapter 4 [Integers], page 18.

IsNonPositive
Predicate on Type_Integer

Compare an integer with zero (not greater): f = `IsNonPositive(`I`)`

I: `const Type_Integer &`

f: `Type_Bool`

`TRUE` if I is less than or equal to zero; `FALSE` otherwise.

Header File
'`Sturm/Integer.h`'

See Also
Chapter 4 [Integers], page 18.

IsNonZero
Predicate on Type_Integer

Compare an integer with zero (not equal): f = `IsNonZero(`I`)`

I: `const Type_Integer &`

f: `Type_Bool`

`TRUE` if I is not equal to zero; `FALSE` otherwise.

Header File
'`Sturm/Integer.h`'

See Also
Chapter 4 [Integers], page 18.

IsOne
Predicate on Type_Integer

Compare an integer with one (equal): f = `IsOne(`I`)`

I: `const Type_Integer &`

f: `Type_Bool`

`TRUE` if I is equal to one; `FALSE` otherwise.

Header File
'`Sturm/Integer.h`'

See Also
Chapter 4 [Integers], page 18.

IsPositive — Predicate on Type_Integer

Compare an integer with zero (greater): f `= IsPositive(`I`)`

I: `const Type_Integer &`

f: `Type_Bool`

`TRUE` if I is greater than zero; `FALSE` otherwise.

Header File
'`Sturm/Integer.h`'

See Also
Chapter 4 [Integers], page 18.

IsZero — Predicate on Type_Integer

Compare an integer with zero (equal): f `= IsZero(`I`)`

I: `const Type_Integer &`

f: `Type_Bool`

`TRUE` if I is equal to zero; `FALSE` otherwise.

Header File
'`Sturm/Integer.h`'

See Also
Chapter 4 [Integers], page 18.

== — Comparison operator on Type_Integer

Compare integers (equal): r `=` I `==` J

I, J: `const Type_Integer &`

r: `Type_Bool`

`TRUE` if I and J are equal; `FALSE` otherwise.

Header File
'`Sturm/Integer.h`'

See Also
`!=` on `Type_Integer`, `<=` on `Type_Integer`, `>=` on `Type_Integer`, `<` on `Type_Integer`, `>` on `Type_Integer` , Chapter 4 [Integers], page 18.

> Comparison operator on Type_Integer

Compare integers (greater): r `=` I `>` J

I, J: `const Type_Integer &`

r: `Type_Bool`

`TRUE` if I is greater than J; `FALSE` otherwise.

Header File

'`Sturm/Integer.h`'

See Also

`==` on `Type_Integer`, `!=` on `Type_Integer`, `<=` on `Type_Integer`, `>=` on `Type_Integer`, `<` on `Type_Integer` , Chapter 4 [Integers], page 18.

>= Comparison operator on Type_Integer

Compare integers (greater or equal): r `=` I `>=` J

I, J: `const Type_Integer &`

r: `Type_Bool`

`TRUE` if I is greater than or equal to J; `FALSE` otherwise.

Header File

'`Sturm/Integer.h`'

See Also

`==` on `Type_Integer`, `!=` on `Type_Integer`, `<=` on `Type_Integer`, `<` on `Type_Integer`, `>` on `Type_Integer` , Chapter 4 [Integers], page 18.

< Comparison operator on Type_Integer

Compare integers (less): r `=` I `>` J

I, J: `const Type_Integer &`

r: `Type_Bool`

`TRUE` if I is less than J; `FALSE` otherwise.

Header File

'`Sturm/Integer.h`'

See Also

`==` on `Type_Integer`, `!=` on `Type_Integer`, `<=` on `Type_Integer`, `>=` on `Type_Integer`, `>` on `Type_Integer` , Chapter 4 [Integers], page 18.

<= Comparison operator on Type_Integer

Compare integers (less or equal): r `=` I `<=` J

I, J: `const Type_Integer &`

r: `Type_Bool`

`TRUE` if *I* is less than or equal to *J*; `FALSE` otherwise.

Header File
'`Sturm/Integer.h`'

See Also
`==` on `Type_Integer`, `!=` on `Type_Integer`, `>=` on `Type_Integer`, `<` on `Type_Integer`, `>` on `Type_Integer` , Chapter 4 [Integers], page 18.

!= Comparison operator on Type_Integer

Compare integers (not equal): r `=` I `!=` J

I, *J*: `const Type_Integer &`

r: `Type_Bool`

`TRUE` if *I* and *J* are not equal; `FALSE` otherwise.

Header File
'`Sturm/Integer.h`'

See Also
`==` on `Type_Integer`, `<=` on `Type_Integer`, `>=` on `Type_Integer`, `<` on `Type_Integer`, `>` on `Type_Integer` , Chapter 4 [Integers], page 18.

A.4 Interval64

A.4.1 Arithmetic

A.4.1.1 Constructive

`+` Arithmetic operator on Type_Interval64

Add a floating point number and an interval: C `=` A `+` B

A: `Type_Float64`

B: `Type_Interval64`

Not empty.

C: `Type_Interval64`

Header File
'`Sturm/Interval64.h`'

See Also
Chapter 7 [Intervals], page 34.

+ Arithmetic operator on Type_Interval64

Add an interval and a floating point number: C `=` A `+` B

A: `Type_Interval64`

B: `Type_Float64`

Not Nan.

C: `Type_Interval64`

Header File

'`Sturm/Interval64.h`'

See Also

Chapter 7 [Intervals], page 34.

+ Arithmetic operator on Type_Interval64

Add two intervals: C `=` A `+` B

A: `Type_Interval64`

Not empty.

B: `Type_Interval64`

Not empty.

C: `Type_Interval64`

Header File

'`Sturm/Interval64.h`'

See Also

Chapter 7 [Intervals], page 34.

/ Arithmetic operator on Type_Interval64

Divide a floating point number by an interval: C `=` A `/` B

A: `Type_Float64`

B: `Type_Interval64`

Not empty, does not contain zero.

C: `Type_Interval64`

Header File

'`Sturm/Interval64.h`'

See Also

Chapter 7 [Intervals], page 34.

/ Arithmetic operator on Type_Interval64

Divide an interval by a floating point number: C `=` A `/` B

A: `Type_Interval64`
    Not empty.

B: `Type_Float64`

C: `Type_Interval64`

Header File
'`Sturm/Interval64.h`'

See Also
Chapter 7 [Intervals], page 34.

/ Arithmetic operator on Type_Interval64

Divide two intervals: C `=` A `/` B

A: `Type_Interval64`
    Not empty.

B: `Type_Interval64`
    Not empty, does not contain zero.

C: `Type_Interval64`

Header File
'`Sturm/Interval64.h`'

See Also
Chapter 7 [Intervals], page 34.

`-` Arithmetic operator on Type_Interval64

Negative interval. C `= -` A

A: `Type_Float64`
    Not empty.

C: `Type_Interval64`

Header File
'`Sturm/Interval64.h`'

See Also
Chapter 7 [Intervals], page 34.

* Arithmetic operator on Type_Interval64

Multiply a floating point number and an interval: C `=` A `*` B

A: `Type_Float64`

B: `Type_Interval64`
    Not empty.

C: `Type_Interval64`

Header File
'`Sturm/Interval64.h`'

See Also
Chapter 7 [Intervals], page 34.

`*` Arithmetic operator on Type_Interval64

Multiply an interval and a floating point number: C = A * B

A: `Type_Interval64`
  Not empty.

B: `Type_Float64`

C: `Type_Interval64`

Header File
'`Sturm/Interval64.h`'

See Also
Chapter 7 [Intervals], page 34.

`*` Arithmetic operator on Type_Interval64

Multiply two intervals: C = A * B

A: `Type_Interval64`
  Not empty.

B: `Type_Interval64`
  Not empty.

C: `Type_Interval64`

Header File
'`Sturm/Interval64.h`'

See Also
Chapter 7 [Intervals], page 34.

`-` Arithmetic operator on Type_Interval64

Subtract a floating point number and an interval: C = A - B

A: `Type_Float64`

B: `Type_Interval64`
  Not empty.

C: `Type_Interval64`

Header File
'`Sturm/Interval64.h`'

See Also
Chapter 7 [Intervals], page 34.

- Arithmetic operator on Type_Interval64

Subtract an interval and a floating point number: C `=` A `-` B

A: `Type_Interval64`
Not empty.

B: `Type_Float64`

C: `Type_Interval64`

Header File
'`Sturm/Interval64.h`'

See Also
Chapter 7 [Intervals], page 34.

- Arithmetic operator on Type_Interval64

Subtract two intervals: C `=` A `-` B

A: `Type_Interval64`
Not empty.

B: `Type_Interval64`
Not empty.

C: `Type_Interval64`

Header File
'`Sturm/Interval64.h`'

See Also
Chapter 7 [Intervals], page 34.

A.4.1.2 Destructive

+= Arithmetic operator on Type_Interval64

Add a floating point number to an interval: A `+=` B

A: `Type_Interval64 &`
Not empty.

B: `Type_Float64`

Warnings
The return value of A `+=` B is `void` and not `Type_Interval64`.

Header File
'`Sturm/Interval64.h`'

See Also
Chapter 7 [Intervals], page 34.

`+=` Arithmetic operator on Type_Interval64

Add an interval to an interval: *A* `+=` *B*

A: `Type_Interval64 &`
Not empty.

B: `Type_Interval64`
Not empty.

Warnings
The return value of *A* `+=` *B* is `void` and not `Type_Interval64`.

Header File
'`Sturm/Interval64.h`'

See Also
Chapter 7 [Intervals], page 34.

`/=` Arithmetic operator on Type_Interval64

Divide an interval by a floating point number: *A* `/=` *B*

A: `Type_Interval64 &`
Not empty.

B: `Type_Float64`
Not zero.

Warnings
The return value of *A* `/=` *B* is `void` and not `Type_Interval64`.

Header File
'`Sturm/Interval64.h`'

See Also
Chapter 7 [Intervals], page 34.

`/=` Arithmetic operator on Type_Interval64

Divide an interval by an interval: *A* `/=` *B*

A: `Type_Interval64 &`
Not empty.

B: `Type_Interval64`
Not empty, does not contain zero.

Warnings
The return value of *A* `/=` *B* is `void` and not `Type_Interval64`.

Header File
'Sturm/Interval64.h'

See Also
Chapter 7 [Intervals], page 34.

*= Arithmetic operator on Type_Interval64

Multiply an interval by a floating point number: A `*=` B

A: `Type_Interval64 &`
Not empty.

B: `Type_Float64`

Warnings
The return value of A `*=` B is `void` and not `Type_Interval64`.

Header File
'Sturm/Interval64.h'

See Also
Chapter 7 [Intervals], page 34.

*= Arithmetic operator on Type_Interval64

Multiply an interval by an interval: A `*=` B

A: `Type_Interval64 &`
Not empty.

B: `Type_Interval64`
Not empty.

Warnings
The return value of A `*=` B is `void` and not `Type_Interval64`.

Header File
'Sturm/Interval64.h'

See Also
Chapter 7 [Intervals], page 34.

-= Arithmetic operator on Type_Interval64

Subtract a floating point number from an interval: A `-=` B

A: `Type_Interval64 &`
Not empty.

B: `Type_Float64`

Warnings
The return value of A `-=` B is `void` and not `Type_Interval64`.

Header File
'Sturm/Interval64.h'

See Also
Chapter 7 [Intervals], page 34.

-= Arithmetic operator on Type_Interval64

Subtract an interval from an interval: A `-=` B

A: `Type_Interval64 &`
Not empty.

B: `Type_Interval64`
Not empty.

Warnings
The return value of A `-=` B is `void` and not `Type_Interval64`.

Header File
'Sturm/Interval64.h'

See Also
Chapter 7 [Intervals], page 34.

A.4.2 Assignment

= Arithmetic operator on Type_Interval64

Assign a floating point number to an interval: A `=` B

A: `Type_Interval64`

B: `Type_Float64`
Not Nan.

Description
If B is finite, then A will be the one element interval containing B. If B is plus infinity then A will be the interval containing all real numbers which are greater or equal to `MaxFloat64()`. If B is minus infinity then A will will be the set of all real numbers less or equal to `-MaxFloat64()`.

Warnings
The return value of A `=` B is `void` and not `Type_Interval64`.

Header File
'Sturm/Interval64.h'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

= Arithmetic operator on Type_Interval64

Assign an interval to an interval: A = B

A: `Type_Interval64`

B: `Type_Interval64`

Warnings
The return value of A = B is `void` and not `Type_Interval64`.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

A.4.3 Constants

AllInterval64 Function on Type_Interval64

description: R = `AllInterval64()`

R: `Type_Interval64`

The interval which contains all real numbers.

Header File
'`Sturm/Interval64.h`'

See Also
`ZeroInterval64`, `OneInterval64`, `EmptyInterval64` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

EmptyInterval64 Function on Type_Interval64

description: R = `EmptyInterval64()`

R: `Type_Interval64`

The empty interval.

Header File
'`Sturm/Interval64.h`'

See Also
`ZeroInterval64`, `OneInterval64`, `AllInterval64` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

OneInterval64 Function on Type_Interval64

description: R = `OneInterval64()`

R: `Type_Interval64`

The single element interval containing zero.

Header File
'`Sturm/Interval64.h`'

See Also
`ZeroInterval64`, `EmptyInterval64`, `AllInterval64` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

ZeroInterval64 Function on Type_Interval64

description: R `= ZeroInterval64()`

R: `Type_Interval64`

The single element interval containing zero.

Header File
'`Sturm/Interval64.h`'

See Also
`OneInterval64`, `EmptyInterval64`, `AllInterval64` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

A.4.4 Constructors

Interval64 Function on Type_Interval64

Construct an interval from bounding floating point numbers: `R = Interval64(`A`, `B`)`

A: `Type_Float64`

The lower bound, not Nan, not infinity.

B: `Type_Float64`

The upper bound, not Nan, not minus infinity. A must be less or equal B.

R: `Type_Interval64`

If A is finite, then the lower bound of R is A, otherwise R is left open. If B is finite, then the upper bound of R is B, otherwise R is right open.

Header File
'`Sturm/Interval64.h`'

See Also
`Lower`, `Upper` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

A.4.5 Functions

A.4.5.1 Number

Abs Function on Type_Interval64

Absolute value interval: `R = Abs(`A`)`

A: `Type_Interval64`
: Not empty.

R: `Type_Interval64`
: The interval consisting of the absolute values of the elements of A.

Header File
'`Sturm/Interval64.h`'

See Also
`Mag`, `Mig` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

Distance Function on Type_Interval64

Distance of two intervals: `R = Distance(`A`, `B`)`

A: `Type_Interval64`
: Not empty.

B: `Type_Interval64`
: Not empty.

R: `Type_Float64`
: If A and B are bounded then R is the smallest floating point number which is greater or equal to the distance of A and B. Otherwise R is infinity.

Header File
'`Sturm/Interval64.h`'

See Also
`Width`, `Rad` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

Gdiv Procedure on Type_Interval64

Generalized division: `R = Gdiv(`A`, `B`, `C`, `D`)`

A: `Type_Interval64`
: The numerator, not empty.

B: `Type_Interval64`
: The denominator, not empty, may contain zero.

C, D: `Type_Interval64 &`

Description
C and D are disjoint. If C is empty then D is empty. If C and D are both nonempty then C is less than D. If x is a real number such that B times x intersects A then x is either in C or in D.

Header File
'Sturm/Interval64.h'

See Also
`Hdiv` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

Hdiv

Function on Type_Interval64

Hull division: `C = Hdiv(`A`, `B`)`

A: `Type_Interval64`
  The numerator, not empty.

B: `Type_Interval64`
  The denominator, not empty, may contain zero.

C: `Type_Interval64`
  The smallest interval which contains `Gdiv(`A`,`B`)`.

Header File
'Sturm/Interval64.h'

See Also
`Gdiv` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

Mag

Function on Type_Interval64

Magnitude: `R = Mag(`A`)`

A: `Type_Interval64`
  Not empty.

R: `Type_Interval64`
  The absolute value of the absolutely largest element of A.

Header File
'Sturm/Interval64.h'

See Also
`Mig` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

Mid

Function on Type_Interval64

Approximate the midpoint of an interval: `R = Mid(`A`)`

A: `Type_Interval64`
  Not empty.

R: `Type_Float64`
  Approximation of the midpoint of A.

Description
R is element of A. If A contains more than 2 floating point numbers then R is in the interior of A. If A is the set of all real numbers then R is zero. Otherwise, if A is right

open, then R is the largest floating point number and if A is left open then R is the smallest negative floating point number. description

Header File
'Sturm/Interval64.h'

See Also
`Width`, `MidMid` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

MidRad
Procedure on Type_Interval64

Get a midpoint radius representation of an interval: `MidRad(`A`, `M`, `R`)`

A: `Type_Interval64`
: Not empty.

M: `Type_Interval64 &`
: An approximation of the midpoint of A.

R: `Type_Interval64 &`
: An overestimation of the radius of A.

Description
The interval which is defined by M and R is a suberset of A or equal A.

Header File
'Sturm/Interval64.h'

See Also
`Hdiv` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

Mig
Function on Type_Interval64

Mignitude: `R = Mig(`A`)`

A: `Type_Interval64`
: Not empty.

R: `Type_Interval64`
: The absolute value of the absolutely smallest element of A.

Header File
'Sturm/Interval64.h'

See Also
`Mag` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

Rad
Function on Type_Interval64

Compute the radius of an interval: `R = Rad(`A`)`

A: `Type_Interval64`
: Not empty.

R: `Type_Interval64`
The smallest floating point number which is greater or equal to the radius of *A*.

Header File
'`Sturm/Interval64.h`'

See Also
`Width`, `MidRad` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

Sqr
Function on Type_Interval64

Compute the square of an interval: `R = Sqr(`*A*`)`

A: `Type_Interval64`
Not empty.

R: `Type_Interval64`
The smallest floating point interval which contains the set of all squares of elements of *A*.

Header File
'`Sturm/Interval64.h`'

See Also
`Sqrt` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

Sqrt
Function on Type_Interval64

Compute the square root of an interval: `R = Sqr(`*A*`)`

A: `Type_Interval64`
Not empty, contains no negative numbers.

R: `Type_Interval64`
The smallest floating point interval which contains the set of all square roots of elements of *A*.

Header File
'`Sturm/Interval64.h`'

See Also
`Sqr` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

Width
Function on Type_Interval64

Compute the width of an interval: `R = Width(`*A*`)`

A: `Type_Interval64`
Not empty.

R: `Type_Interval64`
The smallest floating point number which is greater or equal to the width of *A*.

Header File
'Sturm/Interval64.h'

See Also
`Width` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

A.4.5.2 Set

Bisect Procedure on Type_Interval64

Bisect an interval at its midpoint: `Bisect(`A`, `L`, `R`)`

A: `Type_Interval64`
: Not empty.

L: `Type_Interval64 &`
: The left half of A.

R: `Type_Interval64 &`
: The right half of A.

Description
A is split into two subintervals L and R, which share exactly one point `Mid(A)`.

Header File
'Sturm/Interval64.h'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

Hull Function on Type_Interval64

Hull of two floating point numbers: C `= Hull(`A`, `B`)`

A: `Type_Float64`

B: `Type_Float64`

C: `Type_Interval64`
: The smallest interval which contains A and B.

Header File
'Sturm/Interval64.h'

See Also
, ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

Hull Function on Type_Interval64

Hull of an interval and a floating point number: C `= Hull(`A`, `B`)`

A: `Type_Float64`

B: `Type_Interval64`

C: `Type_Interval64`

The smallest super interval of B which contains A.

Header File

'`Sturm/Interval64.h`'

See Also

, ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

Hull

Function on Type_Interval64

Hull of an interval and a floating point number: C = `Hull(`A`, `B`)`

A: `Type_Interval64`

B: `Type_Float64`

C: `Type_Interval64`

The smallest super interval of A which contains B.

Header File

'`Sturm/Interval64.h`'

See Also

, ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

Hull

Function on Type_Interval64

Hull of two intervals: C = `Hull(`A`, `B`)`

A: `Type_Interval64`

B: `Type_Interval64`

C: `Type_Interval64`

The smallest interval which contains the union of A and B.

Header File

'`Sturm/Interval64.h`'

See Also

, ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

Intersect

Function on Type_Interval64

Intersection of two intervals: C = `Intersect(`A`, `B`)`

A: `Type_Interval64`

B: `Type_Interval64`

C: `Type_Interval64`

The intersection of A and B.

Header File

'`Sturm/Interval64.h`'

See Also
`Hull` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

A.4.6 IO

`>>` IO operator on Type_Interval64

Read an interval from a stream. *is* `>>` A

is: `istream &`

A: `Type_Interval64 &`

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

`<<` IO operator on Type_Interval64

Write an interval to a stream. *os* `<<` A

os: `ostream &`

A: `Type_Interval64`

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

A.4.7 Predicates

A.4.7.1 Number

`>` Comparison operator on Type_Interval64

Comparison of a floating point number and an interval: C = A > B

A: `Type_Float64`
Not Nan.

B: `Type_Interval64`
Not empty.

C: `Type_Bool`
If A is greater than every element of B then `TRUE`, else `FALSE`.

Header File
'Sturm/Interval64.h'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

> Comparison operator on Type_Interval64

Comparison of a floating point number and an interval: C = A > B

A: `Type_Interval64`
Not empty.

B: `Type_Float64`
Not Nan.

C: `Type_Bool`
If every element of A is greater than B then `TRUE`, else `FALSE`.

Header File
'Sturm/Interval64.h'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

> Comparison operator on Type_Interval64

Comparison of two intervals: C = A > B

A: `Type_Interval64`
Not empty.

B: `Type_Interval64`
Not empty.

C: `Type_Bool`
If every element of A is greater than every element of B then `TRUE`, else `FALSE`.

Header File
'Sturm/Interval64.h'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

>= Comparison operator on Type_Interval64

Comparison of a floating point number and an interval: C = A >= B

A: `Type_Float64`
Not Nan.

B: `Type_Interval64`
Not empty.

C: `Type_Bool`
If A is greater or equal than every element of B then `TRUE`, else `FALSE`.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

`>=` Comparison operator on Type_Interval64
Comparison of a floating point number and an interval: C `=` A `>=` B

A: `Type_Interval64`
Not empty.
B: `Type_Float64`
Not Nan.
C: `Type_Bool`
If every element of A is greater or equal than B then `TRUE`, else `FALSE`.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

`>=` Comparison operator on Type_Interval64
Comparison of two intervals: C `=` A `>=` B

A: `Type_Interval64`
Not empty.
B: `Type_Interval64`
Not empty.
C: `Type_Bool`
If every element of A is greater or equal than every element of B then `TRUE`, else `FALSE`.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

`<` Comparison operator on Type_Interval64
Comparison of a floating point number and an interval: C `=` A `<` B

A: `Type_Float64`
Not Nan.

B: `Type_Interval64`

Not empty.

C: `Type_Bool`

If A is less than every element of B then `TRUE`, else `FALSE`.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

`<` — Comparison operator on Type_Interval64

Comparison of a floating point number and an interval: $C = A < B$

A: `Type_Interval64`

Not empty.

B: `Type_Float64`

Not Nan.

C: `Type_Bool`

If every element of A is less than B then `TRUE`, else `FALSE`.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

`<` — Comparison operator on Type_Interval64

Comparison of two intervals: $C = A < B$

A: `Type_Interval64`

Not empty.

B: `Type_Interval64`

Not empty.

C: `Type_Bool`

If every element of A is less than every element of B then `TRUE`, else `FALSE`.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

`<=` — Comparison operator on Type_Interval64

Comparison of a floating point number and an interval: $C = A <= B$

A: `Type_Float64`
  Not Nan.

B: `Type_Interval64`
  Not empty.

C: `Type_Bool`
  If A is less or equal than every element of B then `TRUE`, else `FALSE`.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

<= Comparison operator on Type_Interval64

Comparison of a floating point number and an interval: C `=` A `<=` B

A: `Type_Interval64`
  Not empty.

B: `Type_Float64`
  Not Nan.

C: `Type_Bool`
  If every element of A is less or equal than B then `TRUE`, else `FALSE`.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

<= Comparison operator on Type_Interval64

Comparison of two intervals: C `=` A `<=` B

A: `Type_Interval64`
  Not empty.

B: `Type_Interval64`
  Not empty.

C: `Type_Bool`
  If every element of A is less or equal than every element of B then `TRUE`, else `FALSE`.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

A.4.7.2 NumberAndSet

== Comparison operator on Type_Interval64

Comparison of a floating point number and an interval: C = A == B

A: `Type_Float64`

Not Nan.

B: `Type_Interval64`

C: `Type_Bool`

If B is the singleton interval which contains A then `TRUE`, else `FALSE`.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

== Comparison operator on Type_Interval64

Comparison of a floating point number and an interval: C = A == B

A: `Type_Interval64`

B: `Type_Float64`

Not Nan.

C: `Type_Bool`

If A is the singleton interval which contains B then `TRUE`, else `FALSE`.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

== Comparison operator on Type_Interval64

Comparison of two interval: C = A == B

A: `Type_Interval64`

B: `Type_Interval64`

C: `Type_Bool`

If A and B contain the same elements then `TRUE`, else `FALSE`.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

!= Comparison operator on Type_Interval64

Comparison of a floating point number and an interval: C = A != B

A: `Type_Float64`

Not Nan.

B: `Type_Interval64`

C: `Type_Bool`

If B is the singleton interval which contains A then `FALSE`, else `TRUE`.

Header File

'`Sturm/Interval64.h`'

See Also

⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

== Comparison operator on Type_Interval64

Comparison of a floating point number and an interval: C = A != B

A: `Type_Interval64`

B: `Type_Float64`

Not Nan.

C: `Type_Bool`

If A is the singleton interval which contains B then `FALSE`, else `TRUE`.

Header File

'`Sturm/Interval64.h`'

See Also

⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

!= Comparison operator on Type_Interval64

Comparison of two interval: C = A != B

A: `Type_Interval64`

B: `Type_Interval64`

C: `Type_Bool`

If A and B contain the same elements then `FALSE`, else `TRUE`.

Header File

'`Sturm/Interval64.h`'

See Also

⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

A.4.7.3 Set

IsDisjoint

Function on Type_Interval64

Decide whether two intervals are disjoint: C `= IsDisjoint(`A`, `B`)`

A, B: `Type_Interval64`

C: `Type_Bool`

`TRUE` if A and B are disjoint, `FALSE` otherwise.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

IsElement

Function on Type_Interval64

Decide whether a floating point number is element of an interval: C `= IsSuperset(`A`,` B`)`

A: `Type_Float64`

Not Nan.

B: `Type_Interval64`

C: `Type_Bool`

`TRUE` if A is a element of B, `FALSE` otherwise.

Description
If A is infinty then A is element of B if and only if B is right open. If A is minus infinty then A is element of B if and only if B is left open.

Warnings
The floating point number infinity is not interpreted as a number which is greater than all real numbers but as a real number which is greater than the largest finite floating point number. This seems to be reasonable because it avoids the necessity of checking for overflows.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

IsEmpty

Function on Type_Interval64

Decide whether an interval is empty. R `= IsEmpty(`A`)`

A: `Type_Interval64`

R: `Type_Bool`

`TRUE` if A is a empty, `FALSE` otherwise.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

IsElement
Function on Type_Interval64

Decide whether a floating point number is in the interior of an interval: C `= IsSuperset(`A`, `B`)`

A: `Type_Float64`
: Not Nan.

B: `Type_Interval64`

C: `Type_Bool`
: `TRUE` if A is in the interior of B, `FALSE` otherwise.

Description
If A is infinty then A is inside B if and only if B is right open. If A is minus infinty then A is element of B if and only if B is left open.

Warnings
The floating point number infinity is not interpreted as a number which is greater than all real numbers but as a real number which is greater than the largest finite floating point number. This seems to be reasonable because it avoids the necessity of checking for overflows.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

IsInside
Function on Type_Interval64

Decide whether an interval is in the interior of another interval: C `= IsInside(`A`, `B`)`

A, B: `Type_Interval64`

C: `Type_Bool`
: `TRUE` if A is in the interior if B, `FALSE` otherwise.

Header File
'`Sturm/Interval64.h`'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

IsProperSubset
Function on Type_Interval64

Decide whether an interval is a proper subset of another interval: C `= IsProperSubset(`A`, `B`)`

A, B: `Type_Interval64`

C: `Type_Bool`

TRUE if A is a proper subset of B, FALSE otherwise.

Header File
'Sturm/Interval64.h'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

IsSubset Function on Type_Interval64

Decide whether an interval is a subset of another interval: C = IsSubset(A, B)

A, B: `Type_Interval64`

C: `Type_Bool`

TRUE if A is a subset of B, FALSE otherwise.

Header File
'Sturm/Interval64.h'

See Also
⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

A.4.8 Selectors

Lower Function on Type_Interval64

Get the lower bound of an interval: R = Lower(A)

A: `Type_Interval64`

Not empty.

R: `Type_Float64`

The lower bound of A if A is left bounded, minus infinity else.

Header File
'Sturm/Interval64.h'

See Also
Interval64, Upper , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

Upper Function on Type_Interval64

Get the upper bound of an interval: R = Upper(A)

A: `Type_Interval64`

Not empty.

R: `Type_Float64`

The upper bound of A if A is right bounded, plus infinity else.

Header File
'Sturm/Interval64.h'

See Also
`Interval64`, `Lower` , ⟨undefined⟩ [Interval64s], page ⟨undefined⟩.

A.5 Kernel

A.5.1 Error

A.5.1.1 Call

ERR_CallHandler Procedure

Call an error handler: `ERR_CallHandler(`*fn*`,`*msg*`,`*argc*`,`*argv*`)`

fn: `char *` The name of the function in which the error occured.

msg: `char *`
The error message.

argc The number of optional arguments to the error handler.

argv The values of the optional arguments.

Description
If an error handler is defined for function *fn* then this handler is called with the arguments *fn*, *msg*, the file name and line number where `ERR_CallHandler` was called, *argc* and *argv*. Otherwise the default error handler is called.

Header File
'Sturm/Kernel.h'

See Also
`ERR_DefineHandler` , Section 3.1 [Error Handler], page 6.

A.5.1.2 Define

ERR_DefineHandler Procedure

Define an error handler: `ERR_DefineHandler(`*fn*`,`*handler*`)`

fn: `char *` The name of the function for which a handler is defined.

handler: void (*) (char *, char *, char *, int, int , char**)
Pointer to the error handler function

Description
The function pointed to by *handler* is becomes the error handler for function *fn*. If *fn*

is the empty string then *handler* is the new default error handler. When *handler* is called thorugh `ERR_CallHandler(`*fn*`,`*msg*`,`*argc*`,`*argv*`)` it gets the arguments *fn*, *msg*, the name of the file and the line number of the `ERR_CallHandler` statement, *argc* and *argv*.

Header File
'`Sturm/Kernel.h`'

See Also
`ERR_CallHandler` , Section 3.1 [Error Handler], page 6.

A.5.1.3 Initialization

ERR_Init Procedure

Initialize the error handler: `ERR_Init()`

Description
Define a default floating error handler and initialize the list of used defined error handlers.

Header File
'`Sturm/Kernel.h`'

See Also
Section 3.1 [Error Handler], page 6.

A.5.2 Fpu

A.5.2.1 Initialization

A.5.3 Initialization

STURM_Cleanup Function

Call STURM cleanup functions: `STURM_Cleanup()`

Remarks
For internal use only.

Header File
'`Kernel/Initialization.h`'

See Also
`STURM_Init`, `STURM_GetOpts`, `main` , Section 2.5 [Implementation (INI)], page 4.

STURM_GetOpts Function

Parse STURM command line arguments: *f* = `STURM_GetOpts(`*ac*`,` *av*`)`

av: `int *` A pointer to a variable containing the number of command line arguments.

ac: `char **`
: An array of *av* pointers to strings containing the separate command line arguments.

f: `int` A flag indicating whether option '`--help`' is given: 1 if the option is detected, 0 otherwise.

Description

Parses the command line options given in *av* and sets the global STURM options in `GLOBAL_Opts` accordingly. *ac* and *av* are modified such that correctly parsed options are removed. STURM options which are not set from the command line are not modified.

Remarks

For internal use only.

Header File

'`Kernel/Initialization.h`'

See Also

`STURM_Cleanup`, `STURM_Init`, `main` , Section 2.5 [Implementation (INI)], page 4.

STURM_Init Function

Call STURM initialization functions: *f* = `STURM_Init()`

f: `int` A flag indicating success; 1 in case of success, 0 otherwise.

Description

Calls various STURM initialization functions, using option values set in `GLOBAL_Opts`.

Remarks

For internal use only.

Header File

'`Kernel/Initialization.h`'

See Also

`STURM_Cleanup`, `STURM_GetOpts`, `main` , Section 2.5 [Implementation (INI)], page 4.

A.5.4 Memory

MEM_New Function

Allocate objects on the heap: *ptr* = `MEM_New(`*t*`,` *n*`)`

t The type of the objects to be allocated.

n: `size_t`	The number of objects of type *t* to be allocated. *n* must be greater than 0.
ptr: *t* `*`	A pointer to the allocated memory.

Remarks
`MEM_New` is a macro wrapped around the function `MEM_Allocate`.

Header File
'`Sturm/Kernel.h`'

See Also
`MEM_NewInt`, `MEM_Renew`, `MEM_Allocate` , Section 3.2 [Memory Management], page 7

MEM_NewInt Function

Allocate objects on the heap: *ptr* `=` `MEM_NewInt(`*t*`,` *n*`)`

t	The type of the objects to be allocated.
n: `size_t`	The number of objects of type *t* to be allocated. *n* must be greater than 0.
ptr: *t* `*`	A pointer to the allocated memory.

Remarks
`MEM_NewInt` is a macro wrapped around the function `MEM_AllocateInt`.

Warnings
It must be guaranteed that memory allocated by this function will never contain pointers to objects allocated on the heap. This implies that this function must not be used for allocating memory for Sturm objects (such as `Type_Integer` objects) as it can never be guaranteed that a Sturm object does not contain any pointers.

Header File
'`Sturm/Kernel.h`'

See Also
`MEM_New`, `MEM_Renew`, `MEM_AllocateInt` , Section 3.2 [Memory Management], page 7

MEM_Number Function

Return the number of objects in a block allocated on the heap: *n* `=` `MEM_Number(`*t*`,` *p*`)`

t	The type of the objects stored in the allocated block.
p: *t* `*`	A pointer to objects of type *t* allocated by `MEM_New` or `MEM_NewInt`.
n: `size_t`	The number of objects of type *t* allocated in the block pointed to by *p*.

Remarks

- `MEM_Number` is a macro wrapped around the function `MEM_Size`.
- The inequality `n >= MEM_Number(t, MEM_New(t, n))` always holds. Equality cannot be guaranteed due to alignment requirements.

Header File
'Sturm/Kernel.h'

See Also
`MEM_New`, `MEM_NewInt`, `MEM_Renew`, `MEM_Size` , Section 3.2 [Memory Management], page 7

MEM_Renew

Function

Change the size of a block allocated on the heap: *ptr* = `MEM_Renew(`*t*, *p*, *n*`)`

t	The type of the objects stored in the allocated block.
p: *t* `*`	A pointer to objects of type *t* allocated by `MEM_New` or `MEM_NewInt`.
n: `size_t`	The number of objects of type *t* to be allocated. *n* must be greater than 0.
ptr: *t* `*`	A pointer to the allocated memory.

Description
This function allocates memory for storing *n* objects of type *t*. As many objects as fit into the newly allocated memory are copied from the location pointed to by *p*. Then the memory pointed to by *p* is deallocated.

Remarks
`MEM_Renew` is a macro wrapped around the function `MEM_Reallocate`.

Warnings

- Although an attempt is made to keep *ptr* and *p* the same and simply shrink or extend the referenced block of memory, this cannot be guaranteed. Therefore, *p* should always be regarded as an invalid pointer after a call to `MEM_Renew`.
- If the memory pointed to by *p* was allocated by `MEM_NewInt` the restrictions for this type of memory also hold for the memory pointed to by *ptr*.

Header File
'Sturm/Kernel.h'

See Also
`MEM_New`, `MEM_NewInt`, `MEM_Reallocate` , Section 3.2 [Memory Management], page 7

MEM_Init

Function

Initialize memory management: *f* = `MEM_Init(`*stack*, *heapsize*`)`

stack: `void *`

A pointer to a variable which must be located on the program stack *before* any object containing pointers to the heap (e.g. objects of a type provided by the STURM library).

heapsize: `size_t`

The size of the heap in bytes. This is a lower bound.

f: `Type_Bool`

  `TRUE` if the initialization was successful, `FALSE` otherwise. If the result is *FALSE* the functions of the memory management package may not safely be called.

Description
Initializes the memory management package. This function must be executed successfully before any other function of the memory management package may be called.

Remarks
In general this function is not called directly. It is part of the initialization sequence being executed before `STURM_Main` is entered.

Header File
'`Sturm/Kernel.h`'

See Also
Section 3.2 [Memory Management], page 7, Section 2.2 [Main Program], page 3.

MEM_Allocate — Function

Allocate a chunk of memory on the heap: *ptr* `= MEM_Allocate(`*size*`)`

size: `size_t`

  The number of bytes to be allocated. (Due to alignment requirements more memory than asked for might be allocated.)

ptr: `void *`

  A pointer to the allocated chunk of memory.

Header File
'`Sturm/Kernel.h`'

See Also
Section 3.2 [Memory Management], page 7

MEM_AllocateInt — Function

Allocate a chunk of memory on the heap: *ptr* `= MEM_AllocateInt(`*size*`)`

size: `size_t`

  The number of bytes to be allocated. (Due to alignment requirements more memory than asked for might be allocated.)

ptr: `void *`

  A pointer to the allocated chunk of memory. It must be guaranteed that this chunk of memory will never be used for storing pointers to objects allocated on the heap.

Remarks
Using this allocation function may cause garbage collection to run faster. But note the restrictions on the objects which may be stored in the allocated chunk of memory: for example, it is acceptable to use it as an `int` array, but not as an array of `Type_Integer`.

Header File
'Sturm/Kernel.h'

See Also
Section 3.2 [Memory Management], page 7

MEM_Reallocate

Function

Change the size of a chunk of memory: *ptr* = `MEM_Reallocate(`*optr*`,` *size*`)`

optr: `void *`
: A pointer returned by a call to `MEM_Allocate()` or `MEM_Reallocate()`.

size: `size_t`
: The new size of the chunk in bytes. (Due to alignment requirements more memory than asked for might be allocated.)

ptr: `void *`
: A pointer to the allocated chunk of memory. The new chunk of memory will contain as much data from the old chunk as fits into the new size.

Remarks
It is not guaranteed that *ptr* is equal to *optr*. If the two pointers are *not* equal and *ptr* is non-`NULL`, then *optr* is an invalid reference after the call to `MEM_Reallocate()`.

Header File
'Sturm/Kernel.h'

See Also
Section 3.2 [Memory Management], page 7

MEM_Size

Function

Return the size of a chunk of memory: *size* = `MEM_Size(`*ptr*`)`

ptr: `void *`
: A pointer returned by a call to `MEM_Allocate()` or `MEM_Reallocate()`.

size: `size_t`
: The number of bytes in the chunk referenced by *ptr*.

Remarks
The inequality

```
size >= MEM_Size(MEM_Allocate(size))
```

always holds. Equality cannot be guaranteed due to alignment requirements.

Header File
'Sturm/Kernel.h'

See Also
Section 3.2 [Memory Management], page 7

MEM_Free
Function

Deallocate a chunk of memory: `MEM_Free(`*ptr*`)`

ptr: `void *`

A pointer returned by a call to `MEM_Allocate()` or `MEM_Reallocate()`.

Remarks

This function may postpone deallocation until the next garbage collection takes effect if this is more efficient than immediate deallocation. In any case *ptr* becomes an invalid reference.

Header File

'`Sturm/Kernel.h`'

See Also

Section 3.2 [Memory Management], page 7

MEM_Register
Function

Register potential non-stack references to heap objects with the garbage collector: `MEM_Register(`*ptr*, *size*`)`

ptr: `void *`

A pointer referencing an array of potential pointers to objects on the heap.

size: `size_t`

The number of bytes in the array refenced by *ptr*.

Description

This function must be called on global or static objects and arrays of objects which during program execution may contain pointers returned by calls to `MEM_Allocate()` or `MEM_Reallocate()`. These objects must be registered *before* any pointers are stored, otherwise the garbage collector may deallocate the referenced memory.

Header File

'`Sturm/Kernel.h`'

See Also

Section 3.2 [Memory Management], page 7

A.5.5 Termination

STURM_Abort
Function

Abort the execution of a program: `STURM_Abort()`

Description

This function aborts the execution of a program. Flushing of output streams is not guaranteed. A coredump may be produced and a failure status is reported to the shell environment.

Header File
'Sturm/Kernel.h'

See Also
`STURM_Exit` , Section 3.4 [Program Termination], page 16.

STURM_Exit
Function

Terminate the execution of a program: `STURM_Exit(`*s*`)`

s: `int` The exit code of the program.

Description
This function closes and flushes all output streams and terminates the program. The value of the *status* argument is passed to the shell environment as the exit code of the program. The function never returns.

Header File
'Sturm/Kernel.h'

See Also
`STURM_Abort` , Section 3.4 [Program Termination], page 16.

A.5.6 Time

TMR_Clock
Function

Return the system clock: *t* `= TMR_Clock()`

t: `unsigned long`

The number of high-resolution clock ticks since program startup.

Description
This function should yield the highest possible resolution available from the system clock. Therefore, turnaround may be short. (Under one hour for a microsecond clock.) For converting high-resolution ticks into seconds use the formula

```
secs = TMR_Clock()/TMR_TICKS;
```

Header File
'Sturm/Kernel.h'

See Also
`TMR_ClockLong`, `TMR_TICKS` , Section 3.5 [Clock], page 16.

TMR_ClockLong
Function

Return the system clock: *t* `= TMR_ClockLong()`

t: `unsigned long`

The number of low-resolution clock ticks since program startup.

Description
This function is guaranteed to have a turnaround time of more than 24 hours. (Several weeks for a millisecond clock.) For converting low-resolution ticks into seconds use the formula

```
secs = TMR_ClockLong()/TMR_LONGTICKS;
```

Header File
'`Sturm/Kernel.h`'

See Also
`TMR_Clock`, `TMR_LONGTICKS` , Section 3.5 [Clock], page 16.

A.6 Rational

A.6.1 Arithmetic

Abs Function on Type_Rational

Absolute value of a rational: *R* `=` `Abs(`*S*`)`

S: `const Type_Rational &`

R: `Type_Rational`

The absolute value of *R*.

Header File
'`Sturm/Rational.h`'

See Also
`Sign` on `Type_Rational` , Chapter 5 [Rationals], page 25.

Inverse Function on Type_Rational

Invert rational: *R* `=` `Inverse(`*S*`)`

S: `const Type_Rational &`

S non-zero.

R: `Type_Rational`

The inverse of *S*, i.e. `1/S`.

Header File
'`Sturm/Rational.h`'

See Also
Chapter 5 [Rationals], page 25.

Sign Function on Type_Rational

Sign of a rational: s = `Sign(`R`)`

R: `const Type_Rational &`
s: `long` The sign of R.

Header File
'`Sturm/Rational.h`'

See Also
`Abs` on `Type_Rational` , Chapter 5 [Rationals], page 25.

+ Arithmetic operator on Type_Rational

Add rationals: R `=` S `+` T

S, T: `const Type_Rational &`
R: `Type_Rational`
The sum of S and T.

Header File
'`Sturm/Rational.h`'

See Also
Chapter 5 [Rationals], page 25.

+= Arithmetic operator on Type_Rational

Add and assign an integer to a rational: R `+=` I

R: `Type_Rational &`
I: `const Type_Integer &`

Description
The values of R and I are added and the result is stored in R.

Header File
'`Sturm/Rational.h`'

See Also
Chapter 5 [Rationals], page 25.

+= Arithmetic operator on Type_Rational

Add and assign a rational: R `+=` S

R: `Type_Rational &`
S: `const Type_Rational &`

Description
The values of R and S are added and the result is stored in R.

Header File
'Sturm/Rational.h'

See Also
Chapter 5 [Rationals], page 25.

/ Arithmetic operator on Type_Rational

Divide rationals: R `=` S `/` T

S, T: `const Type_Rational &`

T non-zero.

R: `Type_Rational`

The quotient of S and T.

Header File
'Sturm/Rational.h'

See Also
Chapter 5 [Rationals], page 25.

/= Arithmetic operator on Type_Rational

Divide a rational by an integer and assign: R `/=` I

R: `Type_Rational &`

I: `const Type_Integer &`

Description
The values of R is divided by the value of I and the result is stored in R.

Header File
'Sturm/Rational.h'

See Also
Chapter 5 [Rationals], page 25.

/= Arithmetic operator on Type_Rational

Divide and assign rationals: R `/=` S

R: `Type_Rational &`

S: `const Type_Rational &`

Description
The values of R is divided by the value of S and the result is stored in R.

Header File
'Sturm/Rational.h'

See Also
Chapter 5 [Rationals], page 25.

- Arithmetic operator on Type_Rational

Negate rational: R `= -`S

S: `const Type_Rational &`

R: `Type_Rational`

The negative of S.

Header File
'`Sturm/Rational.h`'

See Also
Chapter 5 [Rationals], page 25.

* Arithmetic operator on Type_Rational

Multiply rationals: R `=` S `*` T

S, T: `const Type_Rational &`

R: `Type_Rational &`

The product of S and T.

Header File
'`Sturm/Rational.h`'

See Also
Chapter 5 [Rationals], page 25.

*= Arithmetic operator on Type_Rational

Multiply and assign an integer to a rational: R `*=` I

R: `Type_Rational &`

I: `const Type_Integer &`

Description
The values of R and I are multiplied and the result is stored in R.

Header File
'`Sturm/Rational.h`'

See Also
Chapter 5 [Rationals], page 25.

*= Arithmetic operator on Type_Rational

Multiply and assign a rational: R `*=` S

R: `Type_Rational &`
S: `const Type_Rational &`

Description
The values of R and S are multiplied and the result is stored in R.

Header File
'`Sturm/Rational.h`'

See Also
Chapter 5 [Rationals], page 25.

- Arithmetic operator on Type_Rational

Subtract rationals: R `=` S `-` T

S, T: `const Type_Rational &`
R: `Type_Rational`
The difference of S and T.

Header File
'`Sturm/Rational.h`'

See Also
Chapter 5 [Rationals], page 25.

-= Arithmetic operator on Type_Rational

Subtract an integer from a rational and assign: R `-=` I

R: `Type_Rational &`
I: `const Type_Integer &`

Description
The value of I is subtracted from the value of R and the result is stored in R.

Header File
'`Sturm/Rational.h`'

See Also
Chapter 5 [Rationals], page 25.

-= Arithmetic operator on Type_Rational

Subtract and assign a rational: R `-=` S

R: `Type_Rational &`
S: `const Type_Rational &`

Description
The value of S is subtracted from the value of R and the result is stored in R.

Header File
'Sturm/Rational.h'

See Also
Chapter 5 [Rationals], page 25.

A.6.2 Assignment

= Operator on Type_Rational

Assign an integer to a rational: R `=` I

R: `Type_Rational &`

I: `const Type_Integer &`

Description
R is set to the value of I, i.e. the numerator is I, the denominator is 1.

Header File
'Sturm/Rational.h'

See Also
Chapter 5 [Rationals], page 25.

= Operator on Type_Rational

Assign an rational to a rational: R `=` S

R: `Type_Rational &`

S: `const Type_Rational &`

Description
R is set to the value of S.

Header File
'Sturm/Rational.h'

See Also
Chapter 5 [Rationals], page 25.

= Operator on Type_Rational

Assign a C string to a rational: R `=` s

R: `Type_Rational &`

s: `char *` A null-terminated string.

Description
R is set to the rational number given in s. The format of the character representation consists of

- the character representation of the numerator (an integer), optionally followed by
- a slash (‘/’) followed by the character representation of the denominator (an integer).

Leading spaces and spaces between the integers and the slash are ignored; the denominator must be non-zero.

Header File
‘Sturm/Rational.h’

See Also
`=` on `Type_Integer` and `char *` , Chapter 5 [Rationals], page 25.

A.6.3 Constructors

Initialize Function on Type_Rational

Initialize a rational: `Initialize(`*R*`)`

R: `Type_Rational *`

A pointer to the rational to be initialized.

Description
The rational is set to a state identical to the one set by the `void` constructor. This means that the initialized rational must not be used in computations but may only be assigned a value.

Header File
‘Sturm/Rational.h’

See Also
`Type_Rational(void)` , Chapter 5 [Rationals], page 25.

OneRational Function

The rational representation of 1: *R* `= OneRational()`

R: `Type_Rational`

The rational representation of 1.

Header File
‘Sturm/Rational.h’

See Also
`ZeroRational` , Chapter 5 [Rationals], page 25.

Type_Rational Constructor on Type_Rational

Initialize a rational from integers: `Type_Rational` *R*`(`*N*`,` *D*`,` *f*`)`

N, D: `const Type_Integer &`

The numerator and the denominator, D nonzero.

f: `Type_Bool = FALSE`

A flag specifying whether the numerator and the denominator are normalized. If this argument is omitted, its default value is `FALSE`.

Description

If f is `TRUE`, the numerator and the denominator of R are set to N and D, respectively. If f is `FALSE`, the numerator and the denominator of R are set to normalized values of N and D such that the numerator and the denominator do not have a common factor and the sign of the denominator is positive.

Header File

'`Sturm/Rational.h`'

See Also

`Numerator` on `Type_Rational`, `Denominator` on `Type_Rational` , Chapter 5 [Rationals], page 25.

Type_Rational — Constructor on Type_Rational

Initialize a rational from an integer: `Type_Rational` $R(N)$

N: `const Type_Integer &`

The numerator.

Description

R is set to the value of N.

Header File

'`Sturm/Rational.h`'

See Also

`Numerator` on `Type_Rational` , Chapter 5 [Rationals], page 25.

Type_Rational — Constructor on Type_Rational

Initialize a rational from a string: `Type_Rational` $R(s)$

s: `char *`

Description

R is set to the value given in s, where s is a C++ null-terminated string containing the character representation of a rational in the form

```
INTEGER [/ INTEGER]
```

i.e. the character representation of the numerator followed by an optional slash ('`/`') followed by the character representation of the denominator. Blanks between the integers and the slash are ignored; the denominator must be non-zero.

Header File
'Sturm/Rational.h'

See Also
Chapter 5 [Rationals], page 25.

Type_Rational
Constructor on Type_Rational

Initialize an rational: `Type_Rational` *R*

Description
Default constructor for an object of type `Type_Rational`.

Warnings
R is *not* set to any meaningful value (such as 0)! You must assign a value to the variable before using it.

Header File
'Sturm/Rational.h'

See Also
`Initialize` on `Type_Rational` , Chapter 5 [Rationals], page 25.

ZeroRational
Function

The rational representation of 0: *R* `= ZeroRational()`

R: `Type_Rational`

The rational representation of 0.

Header File
'Sturm/Rational.h'

See Also
`OneRational` , Chapter 5 [Rationals], page 25.

A.6.4 IO

Read
Member function on Type_Rational

Read a rational: `Type_Rational::Read(`*is*`,` *R*`)`

is: `istream &`

A C++ input stream.

R: `Type_Rational &`

The rational to be read.

Description
R is set to a rational number read from the input stream. The input format consists of

- the character representation of the numerator (an integer), optionally followed by
- a slash (‘/’) followed by the character representation of the denominator (an integer).

Leading spaces and spaces between the integers and the slash are ignored; the denominator must be non-zero.

Remarks
This function is for internal use only.

Header File
‘Sturm/Rational.h’

See Also
`>>` on `Type_Integer`, `>>` on `Type_Rational`, `Write` on `Type_Rational` , Chapter 5 [Rationals], page 25.

Write Member function on Type_Rational

Write a rational: `Type_Rational::Write(`*os*`,` *R*`)`

os: `ostream &`
: A C++ output stream.

R: `const Type_Rational &`
: The rational to be written.

Description
The character representation of *R* is written to the output stream. The output format consists of

- the character representation of the numerator (an integer), optionally followed by
- a slash (‘/’) followed by the character representation of the denominator (an integer) if it is not 1.

Remarks
This function is for internal use only.

Header File
‘Sturm/Rational.h’

See Also
`<<` on `Type_Integer`, `<<` on `Type_Rational`, `Read` on `Type_Rational` , Chapter 5 [Rationals], page 25.

>> I/O operator on Type_Rational

Read a rational: *is* `>>` *R*

is: `istream &`
: A C++ input stream.

R: `Type_Rational &`

The rational to be read.

Description
R is set to a rational number read from the input stream. The input format consists of

- the character representation of the numerator (an integer), optionally followed by
- a slash ('`/`') followed by the character representation of the denominator (an integer).

Leading spaces and spaces between the integers and the slash are ignored; the denominator must be non-zero.

Header File
'`Sturm/Rational.h`'

See Also
`>>` on `Type_Integer`, `<<` on `Type_Rational` , Chapter 5 [Rationals], page 25.

`<<` I/O operator on Type_Rational

Write a rational: *os* `<<` *R*

os: `ostream &`

A C++ output stream.

R: `const Type_Rational &`

The rational to be written.

Description
The character representation of *R* is written to the output stream. The output format consists of

- the character representation of the numerator (an integer), optionally followed by
- a slash ('`/`') followed by the character representation of the denominator (an integer) if it is not 1.

Remarks
This function is for internal use only.

Header File
'`Sturm/Rational.h`'

See Also
`<<` on `Type_Integer`, `>>` on `Type_Rational` , Chapter 5 [Rationals], page 25.

A.6.5 Predicates

IsNegative Predicate on Type_Rational

Compare a rational with zero (less): f = `IsNegative(`R`)`

R: `const Type_Rational &`

f: `Type_Bool`

`TRUE` if R is less than zero; `FALSE` otherwise.

Header File

'`Sturm/Rational.h`'

See Also

Chapter 5 [Rationals], page 25.

IsNonNegative Predicate on Type_Rational

Compare a rational with zero (not less): f = `IsNonNegative(`R`)`

R: `const Type_Rational &`

f: `Type_Bool`

`TRUE` if R is equal to or greater than zero; `FALSE` otherwise.

Header File

'`Sturm/Rational.h`'

See Also

Chapter 5 [Rationals], page 25.

IsNonPositive Predicate on Type_Rational

Compare a rational with zero (not greater): f = `IsNonPositive(`R`)`

R: `const Type_Rational &`

f: `Type_Bool`

`TRUE` if R is less than or equal to zero; `FALSE` otherwise.

Header File

'`Sturm/Rational.h`'

See Also

Chapter 5 [Rationals], page 25.

IsNonZero Predicate on Type_Rational

Compare a rational with zero (not equal): f = `IsNonZero(`R`)`

r: `const Type_Rational &`

f: `Type_Bool`

`TRUE` if R is not equal to zero; `FALSE` otherwise.

Header File

'`Sturm/Rational.h`'

See Also
Chapter 5 [Rationals], page 25.

IsOne
Predicate on Type_Rational

Compare a rational with one (equal): f = `IsOne(`R`)`

R: `const Type_Rational &`

f: `Type_Bool`

`TRUE` if R is equal to one; `FALSE` otherwise.

Header File
'`Sturm/Rational.h`'

See Also
Chapter 5 [Rationals], page 25.

IsPositive
Predicate on Type_Rational

Compare a rational with zero (greater): f = `IsPositive(`R`)`

R: `const Type_Rational &`

f: `Type_Bool`

`TRUE` if R is greater than zero; `FALSE` otherwise.

Header File
'`Sturm/Rational.h`'

See Also
Chapter 5 [Rationals], page 25.

IsZero
Predicate on Type_Rational

Compare a rational with zero (equal): f = `IsZero(`R`)`

R: `const Type_Rational &`

f: `Type_Bool`

`TRUE` if R is equal to zero; `FALSE` otherwise.

Header File
'`Sturm/Rational.h`'

See Also
Chapter 5 [Rationals], page 25.

==
Comparison operator on Type_Rational

Compare rationals (equal): f = R `==` S

R, S: `const Type_Rational &`

f: `Type_Bool`

`TRUE` if *R* is equal to *S*; `FALSE` otherwise.

Header File
'`Sturm/Rational.h`'

See Also
`!=` on `Type_Rational`, `<=` on `Type_Rational`, `>=` on `Type_Rational`, `<` on `Type_Rational`, `>` on `Type_Rational` , Chapter 5 [Rationals], page 25.

`>` — Comparison operator on Type_Rational

Compare rationals (greater): f `=` R `>` S

R, *S*: `const Type_Rational &`

f: `Type_Bool`

`TRUE` if *R* is greater than *S*; `FALSE` otherwise.

Header File
'`Sturm/Rational.h`'

See Also
`==` on `Type_Rational`, `!=` on `Type_Rational`, `<=` on `Type_Rational`, `>=` on `Type_Rational`, `<` on `Type_Rational` , Chapter 5 [Rationals], page 25.

`>=` — Comparison operator on Type_Rational

Compare rationals (greater or equal): f `=` R `>=` S

R, *S*: `const Type_Rational &`

f: `Type_Bool`

`TRUE` if *R* is greater than or equal to *S*; `FALSE` otherwise.

Header File
'`Sturm/Rational.h`'

See Also
`==` on `Type_Rational`, `!=` on `Type_Rational`, `<=` on `Type_Rational`, `<` on `Type_Rational`, `>` on `Type_Rational` , Chapter 5 [Rationals], page 25.

`<` — Comparison operator on Type_Rational

Compare rationals (less): f `=` R `<` S

R, *S*: `const Type_Rational &`

f: `Type_Bool`

`TRUE` if *R* is less than *S*; `FALSE` otherwise.

Header File
'`Sturm/Rational.h`'

See Also
`==` on `Type_Rational`, `!=` on `Type_Rational`, `<=` on `Type_Rational`, `>=` on `Type_Rational`, `>` on `Type_Rational` , Chapter 5 [Rationals], page 25.

<= Comparison operator on Type_Rational

Compare rationals (less or equal): f `=` R `<=` S

R, S: `const Type_Rational &`

f: `Type_Bool`

`TRUE` if R is less than or equal to S; `FALSE` otherwise.

Header File
'`Sturm/Rational.h`'

See Also
`==` on `Type_Rational`, `!=` on `Type_Rational`, `>=` on `Type_Rational`, `<` on `Type_Rational`, `>` on `Type_Rational` , Chapter 5 [Rationals], page 25.

!= Comparison operator on Type_Rational

Compare rationals (not equal): f `=` R `!=` S

R, S: `const Type_Rational &`

f: `Type_Bool`

`TRUE` if R and S are not equal; `FALSE` otherwise.

Header File
'`Sturm/Rational.h`'

See Also
`==` on `Type_Rational`, `<=` on `Type_Rational`, `>=` on `Type_Rational`, `<` on `Type_Rational`, `>` on `Type_Rational` , Chapter 5 [Rationals], page 25.

A.6.6 Selectors

Denominator Function on Type_Rational

Get the denominator of a rational: D `= Denominator(`R`)`

R: `const Type_Rational &`

D: `Type_Integer`

The denominator of R.

Header File
'`Sturm/Rational.h`'

See Also
`Numerator` on `Type_Rational` , Chapter 5 [Rationals], page 25.

Numerator Function on Type_Rational

Get the numerator of a rational: N `= Numerator(`R`)`

R: `const Type_Rational &`

N: `Type_Integer`

The numerator of R.

Header File

'`Sturm/Rational.h`'

See Also

`Denominator` on `Type_Rational` , Chapter 5 [Rationals], page 25.

Concept Index

I

K

L

M

N

O

P

R

S

T

W

Function and Type Index

Table of Contents