C/C++-Modules in Singular

Singular - A Computer Algebra System for Polynomial Computations

Technical Report

for SINGULAR version 3-1-6, Dec 2012 and above

SINGULAR is created and its development is directed and coordinated by
W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann

Hans Schönemann

Fachbereich Mathematik
Zentrum für Computeralgebra
Universität Kaiserslautern
D-67653 Kaiserslautern

Short Contents

1 Preface

Dynamic modules provide the possibility to extend SINGULAR dynamically on systems supporting dynamic loading.

Currently, this works on linux ELF systems, on FreeBSD, Solaris and Mac OsX (for dynamically linked binaries of SINGULAR).

The following examples show how to integerate C++ procedures as ian equivalent to SINGULAR library procedures and how to add new data types.

Alternativly, C++-modules may be linked in at link time.

2 Procecedures in a dynamic module

2.1 hello world

The standard example in a minimal setting: no help, no checking of the arguments.

The module only contains the work horse (`hello`) and the required initialization routine of the module(`mod_init`). If the module should be linked statically, the initialization routine must be named `<modle_name>_mod_init`.

```
#include "Singular/mod2.h" // general settings/macros
#include "kernel/febase.h"  // for Print,PrintS, WerrorS
#include "Singular/ipid.h" // for SModulFunctions, leftv

BOOLEAN hello(leftv result, leftv arg)
{
  result->rtyp=NONE;  // set the result type
  PrintS("hello world\n");
  return FALSE; // return FALSE: no error
}

extern "C" int mod_init(SModulFunctions* psModulFunctions)
{
  // this is the initialization routine of the module
  // adding the routine hello:
  psModulFunctions->iiAddCproc(
    (currPack->libname? currPack->libname: ""),// the library name,
            // rely on the loader to set it in currPack->libname
    "hello",// the name for the singular interpreter
    FALSE,  // should enter the global name space
    hello); // the C/C++ routine
  return 1;
}
```

Remark: SINGULAR 4 provides the same mechaism, but requires other include files. The example above starts for SINGULAR 4:

```
#include "kernel/mod2.h" // general settings/macros
#include"kernel/febase.h"  // for Print, WerrorS
#include"Singular/ipid.h" // for SModulFunctions, leftv
....
```

2.2 Parameters and result

The data type in the SINGULAR interpreter is `leftv` (resp. `sleftv`). This type provides the method `Typ()` which describes how the generic result of `Data()` must be interpreted. The generic access routines `Data()` (and `CopyD()`) return `void*`.

The access routine `Data()` gives a (`void *`) pointer to the argument. After returning the interpreter will clean up the argument list therefore this pointer should be considered read-only. The alternative is `CopyD()` which coies the argument and the routine has to do the clean up.

All arguments are linked via the `next` entry in `leftv`.

Returning results is simpler: put the type id into `rtyp` (or NONE for a void routine) and the data pointer into `data`.

Normal output should use `PrintS(const char *)` resp. `Print(...)` (printf systax).
To report an error, use `WerrorS(const char *)` resp. `Werror(...)` for a message and return `TRUE`.

2.3 Integers

For the SINGULAR type `int Typ()` returns `INT_CMD` and `Data()` should be cast to the C type `int`.

```
#include "Singular/mod2.h" // general settings/macros
#include"kernel/febase.h"  // for Print,PrintS, WerrorS
#include"Singular/ipid.h" // for SModulFunctions, leftv

BOOLEAN add (leftv result, leftv arg)
{
  // check the arguments
  if ((arg==NULL)
  ||(arg->next==NULL) // we have not at least 2 arguments
  ||(arg->Typ()!=INT_CMD)
  ||(arg->next->Typ()!=INT_CMD)) // or they are not int
  {
    WerrorS("syntax: add(<int>,<int>)"); // output an error message
    return TRUE; // return TRUE: error
  }
  // now we have at least 2 arguments of type int
  // Data() give read-only access to them
  int a=(int)(long)arg->Data();
  int b=(int)(long)arg->next->Data();
  result->rtyp=INT_CMD; // set the result type
  result->data=(char*)(long)(a+b);// set the result data
  return FALSE;
}
extern "C" int mod_init(SModulFunctions* psModulFunctions)
{
  // this is the initialization routine of the module
  // adding the routine add:
  psModulFunctions->iiAddCproc(
    (currPack->libname? currPack->libname: ""), "add",
    FALSE, // not static
    add);
  return 1;
}
```

2.4 Online help

```
#include "Singular/mod2.h" // general settings/macros
#include"kernel/febase.h"  // for Print, WerrorS
#include"Singular/ipid.h" // for SModulFunctions, leftv

BOOLEAN add (leftv result, leftv arg)
{
...
}
extern "C" int mod_init(SModulFunctions* psModulFunctions)
{
```

```
  // this is the initialization routine of the module
  // adding the routine add:
  psModulFunctions->iiAddCproc(
    (currPack->libname? currPack->libname: ""), "add",FALSE,add);
  const char* add_help="USAGE: add(int a, int b)\n"
                       "RETURN: sum of a and b";
  module_help_proc(
    (currPack->libname? currPack->libname: ""),// the library name,
    "add", // the name of the procedure
    add_help); // the help string
  module_help_main(
    (currPack->libname? currPack->libname: ""),// the library name,
    "a demo module"); // the help string for the module
  return 1;
}
```

2.5 Numbers

For the SINGULAR type `number Typ()` returns `NUMBER_CMD` and `Data()` should be cast to the SINGULARs C++ type `number`.

Numbers depend on the current base ring which is available a global variable `currRing` (resp. `currRingHdl`).

Important generic routines for numbers can be found in `kernel/numbers.h`. These routines (with the exception of nNeg and n*Inp*) create new objects which must be returned or destroyed with nDelete.

2.6 Polynomials

For the SINGULAR type `poly Typ()` returns `POLY_CMD` and `Data()` should be cast to the SINGULARs C++ type `poly`.

Polynomials are also ring dependent, the remarks from numbers apply also here.

The general polynomial routines destroy their input. Exceptions: pHead, pCopy, pp_*.

2.7 Ideals

For the SINGULAR type `poly Typ()` returns `IDEAL_CMD` and `Data()` should be cast to the SINGULARs C++ type `ideal`.

Ideals consist of polynomials and are also ring dependent, the remarks from numbers apply also here.

2.8 Matrices

For the SINGULAR type `matrix Typ()` returns `MATRIX_CMD` and `Data()` should be cast to the SINGULARs C++ type `matrix`.

Matrices consist of polynomials and are also ring dependent, the remarks from numbers apply also here.

2.9 Other types

Typ()	SINGULAR type	C type	ring dep.
BIGINT_CMD	bigint	number	no
BIGINTMAT_CMD	bigintmat	bigintmat*	no
INTMAT_CMD	intmat	intvec*	no
INTVEC_CMD	intvec	intvec*	no
LINK_CMD	link	si_link	no
LIST_CMD	list	lists	yes/no
MAP_CMD	map	map/ideal	yes
MODULE_CMD	module	ideal	yes
PACKAGE_CMD	package	package	no
RING_CMD	ring	ring	no
STRING_CMD	string	char *	no
VECTOR_CMD	vector	poly	yes

3 New data types in a dynamic module

The SINGULAR data type `blackbox` allows to extend the interpreter with new data types.

The `int` data type uses the place of the pointer for the actual data, but here we will create the new type `demo` which is a dynamically allocated `int` together with some basic arithmetic. The initialisation, copy and destroy methods prints the address they work with to show the handling of dynamically allocated objects.

```
#include "Singular/mod2.h" // general settings/macros
#include"kernel/febase.h"  // for Print, WerrorS
#include"Singular/ipid.h" // for SModulFunctions, leftv
#include"Singular/blackbox.h" // for blackbox

static int demoID; // our typ ID
void* demo_Init(blackbox *b)
{
  void *d=omAlloc(sizeof(int));
  Print("create at %lx\n",(unsigned long)d);
  return (void*)d;
}

void demo_destroy(blackbox *b, void *d)
{
  if(d!=NULL)
  {
    Print("destroy %x at %lx\n",*((int*)d),(unsigned long)d);
  }
}
void* demo_Copy(blackbox *b, void *d)
{
  int *dd=(int*)omAlloc(sizeof(int));
  *dd=*((int*)d);
  return dd;
}
char* demo_String(blackbox *b, void *d)
{
  if (d==NULL) return omStrDup("invalid object");
  else
  {
    char buf[20];
    sprintf(buf,"%d",*((int*)d));
    return omStrDup(buf);
  }
}

void demo_Print(blackbox *b, void *d)
{
  Print("%d at %lx",*(int*)d,(unsigned long)d);
  // \n will be added by interpreter
}
BOOLEAN demo_Assign(leftv l, leftv r)
{
  blackbox *b=NULL; //currently unused in demo_Copy
```

```
  if (r->Typ()==l->Typ()) // assignment of same type
  {
    if (l->rtyp==IDHDL) // assign to variable
    {
      omFree(IDDATA((idhdl)l->data));
      IDDATA((idhdl)l->data) = (char*)demo_Copy(b,r->Data());
    }
    else  // assign to a part of a structure
    {
      leftv ll=l->LData();
      if (ll==NULL)
      {
        return TRUE; // out of array bounds or similiar
      }
      omFree(ll->data);
      ll->data = demo_Copy(b,r->Data());
    }
  }
  else if (r->Typ()==INT_CMD)
  {
    if (l->rtyp==IDHDL) // assign to variable
    {
      int *dd=(int*)(IDDATA((idhdl)l->data));
      *dd=(int)(long)r->Data();
    }
    else  // assign to a part of a structure
    {
      leftv ll=l->LData();
      if (ll==NULL)
      {
        return TRUE; // out of array bounds or similiar
      }
      int *dd=(int*)(ll->data);
      *dd=(int)(long)r->Data();
    }
  }
  else
  {
    Werror("assign Type(%d) = Type(%d) not implemented",l->Typ(),r->Typ());
    return TRUE;
  }
  return FALSE;
}

// operations with 2 arguments
BOOLEAN demo_Op2(int op,leftv res, leftv r1,leftv r2)
{
  blackbox *b=NULL; // currently unused
  int *dd=(int*)demo_Init(b);
  // collecting the argumnts in i1,i2
  // accecpt the types int and demo
  int i1,i2;
  if (r1->Typ()==INT_CMD) i1=(int)(long)r1->Data();
```

```
  else if (r1->Typ()==demoID) i1=*((int*)r1->Data());
  else return TRUE;
  if (r2->Typ()==INT_CMD) i2=(int)(long)r2->Data();
  else if (r2->Typ()==demoID) i2=*((int*)r2->Data());
  else return TRUE;
  Print("op:%c, arg1=%d, arg2=%d\n",op,i1,i2);
  switch(op)
  {
    case '+':*dd=i1+i2; break;
    case '-':*dd=i1-i2; break;
    case '*':*dd=i1*i2; break;
    case '/':*dd=i1/i2; break;
    case '%':*dd=i1%i2; break;
    default:
      omFree(dd);// clean up dynamic objects
      // use the default routine for error handling
      return blackboxDefaultOp2(op,res,r1,r2);
  }
  // regular return with the new data in res:
  res->data=dd;
  res->rtyp=demoID;
  return FALSE;
}

extern "C" int mod_init()
{
  blackbox *b=(blackbox*)omAlloc0(sizeof(blackbox));
  // a method to create new objects:
  b->blackbox_Init=demo_Init;
  // and to destroy it:
  b->blackbox_destroy=demo_destroy;
  // copy objects of this type:
  b->blackbox_Copy=demo_Copy;
  // converting to string (also used for printing, if Print is not defined)
  b->blackbox_String=demo_String;
  b->blackbox_Print=demo_Print;
  // assigning something to object of this type:
  b->blackbox_Assign=demo_Assign;
  // implement some operations with 2 arguments:
  b->blackbox_Op2=demo_Op2;
  // and finally, reigiter it as type demo
  demoID=setBlackboxStuff(b,"demo");
  Print("created new type demo with id %d\n",demoID);
  return 1;
}
```

4 Building a dynamic module

A dynamic module is very similiar to a shared library. All used external symbols of the module must be in a shared library or (in contrast to real shared library) in the SINGULAR main executable.

Building demo.so on a ELF system (linux, FreeBSD, Solaris):

```
g++ -I<singular_top_dir> \
   -O -c demo.cc -DOM_NDEBUG -DNDEBUG -fPIC -DPIC -o demo.o
g++ -shared -o demo.so demo.o
```

Building demo.so on Mac OsX:

```
g++ -I<singular_top_dir> \
   -O -c demo.cc -DOM_NDEBUG -DNDEBUG -fPIC -DPIC -o demo.o
libtool -dynamic -twolevel_namespace -weak_reference_mismatches weak \
    -undefined dynamic_lookup -o demo.so demo.o
```

Remark: SINGULAR 4 provides the same mechanism, but requires currently more directories for include files. In th example scripts above change
`-I<singular_top_dir>` to
`-I<singular_top_dir> -I<singular_top_dir>/libpolys -I<build_dir> -I<build_dir>/libpolys -I<singular_top_dir>/factory/include`.

5 Dynamic modules in Singular

5.1 Global procedures

`LIB` loads procedures from a module into the global name space `Top`. Other objects from the module will only appaer in the name space of the module, which is constructed be captilizing the first letter of the module name and converting all others to lower case.

```
LIB"demo.so"; // loading into the global name space
↦ // ** loaded demo.so
hello();
↦ hello world
listvar(package);
↦ // Demo                    [0]  package Demo (C,demo.so)
↦ // Standard                [0]  package Standard (S,standard.lib)
↦ // Top                     [0]  package Top (N)
listvar(Demo);
↦ // Demo                    [0]  package Demo (C,demo.so)
↦ // ::info                 [0]  string a demo module
↦ // ::add_help             [0]  string USAGE: add(int a, in..., 47 char(s)
↦ // ::add                  [0]  proc from demo.so
↦ // ::hello                [0]  proc from demo.so
help Demo;
↦ a demo module
help Demo::add;
↦ help for add from package Demo
↦ USAGE: add(int a, int b)
↦ RETURN: sum of a and b
add(2,3);
↦ 5
add(2,3,4);
↦ 5
add(2,"bla");
↦   ? syntax: add(<int>,<int>)
↦   ? error occurred in or before STDIN line 6: ‘Demo::add(2,"bla");‘
```

5.2 Local procedures

`load` loads a module only into its own name space, which is constructed be captilizing the first letter of the module name and converting all others to lower case.

(Remark: `load("...","with");` is equivalent to `LIB "...."`; and `LIB("...");`.)

```
load("demo.so");
↦ // ** loaded demo.so
listvar(Demo);
↦ // Demo                    [0]  package Demo (C,demo.so)
↦ // ::info                 [0]  string a demo module
↦ // ::add_help             [0]  string USAGE: add(int a, in..., 47 char(s)
↦ // ::add                  [0]  proc from demo.so
↦ // ::hello                [0]  proc from demo.so
add(2,3);
↦    ? ‘add(2)‘ is undefined
↦    ? error occurred in or before STDIN line 3: ‘add(2,3);‘
```

```
Demo::add(2,3);
↦ 5
help Demo;
help Demo::hello;
↦ help for hello from package Demo
↦ ‘hello_help‘ not found in package Demo
string Demo::hello_help="a hello world example"; // adding the missing help
help Demo::hello;
↦ help for hello from package Demo
↦ a hello world example
help Demo::add;
↦ help for add from package Demo
↦ USAGE: add(int a, int b)
↦ RETURN: sum of a and b
```

5.3 Data types

The use of the new data type from the module `bb_demo.so`:

```
LIB"bb_demo.so";
↦ created new type demo with id 522
↦ // ** loaded bb_demo.so
demo i;
↦ create at 7f3224df06b8
i; // still undefined
↦ 618596032 at 7f3224df06b8
i=3; i;
↦ destroy 24df06c0 at 7f3224df06a8
↦ 3 at 7f3224df06b8
demo j=i; j;
↦ create at 7f3224df06d8
↦ destroy 3 at 7f3224df06c0
↦ 3 at 7f3224df06d8
i+j;
↦ create at 7f3224df06e0
↦ op:+, arg1=3, arg2=3
↦ destroy 3 at 7f3224df06c8
↦ 6 at 7f3224df06e0
j=i*2;
↦ create at 7f3224df06e8
↦ op:*, arg1=3, arg2=2
↦ destroy 6 at 7f3224df06e8
print(i);
↦    ? ’blackbox_Op1’ of type demo(522) for op print(479) not implemented
↦    ? error occurred in or before STDIN line 8: ‘print(i);‘
i mod 2; // mapped to %
↦ create at 7f3224df06f0
↦ op:%, arg1=3, arg2=2
↦ destroy 6 at 7f3224df06e0
↦ 1 at 7f3224df06f0
string(j);
↦ destroy 1 at 7f3224df06f0
↦ 6
listvar();
```

```
↦ // j                                  [0]  demo
↦ // i                                  [0]  demo
string(i);
↦ 3
```

6 Static modules in Singular

All the features for dynamic modules apply also to statically linked modules, additionally the module must be linked in and know to the Singular interpreter:

Add module name to the list of builtin modules, for example `staticdemo`

```
#define SI_FOREACH_BUILTIN(add)\
  add(staticdemo)\
  SI_BUILTIN_PYOBJECT(add)
```

in libpolys/polys/mod_raw.h (Singular 4-0) resp. kernel/mod_raw.h (Singular 3-1-6) and provide `int staticdemo_mod_init(SModulFunctions* p)` instead of `int mod_init(SModulFunctions* p)` in your library.

```
int staticdemo_mod_init(SModulFunctions*){ PrintS("init of staticdemo\n"); }
```

Finally add your library to the libraries to link with: (`LIBS` and `LIBSG` in Singular/Makefile (Singular 3-1-6).

The interface in the interpreter is identical to that of dynamic modules

```
LIB"staticdemo.so";
↦ init of staticdemo
↦ // ** loaded (builtin) staticdemo.so
listvar(package);
↦ // Staticdemo  [0]  package Staticdemo (C,staticdemo.so)
↦ // Standard    [0]  package Standard (S,standard.lib)
↦ // Top         [0]  package Top (N)
```

7 Index

Table of Contents