1. Instantiation

1.1 location descriptor of components:

class ctl::location

non-trivial Ctors:
location(const char* str) &&
location(const std::string& str)
str: ssh like syntax:
[user@][host:][path/][exec] [options]
options:
-d <path>: set working directory absolut or relative to location of target component
-e '['<args>']': run test/init routine with args if any connected
-f <log-file[:themes]>: logging specification (only for debug compilation active)
-g <debugger>: invoke component by debugger (default=gdb, see also config)
-I: write list of connected functions and classes to stdout and terminate
-l <linkage-type> [flags]: linkage type one of:
local linkage-types:
lib
thread: starts new thread
file[:] <mode>: open file with mode (see fopen), operators <</>> as file-io
serial: used internally
remote linkage-types:
tcp: starts new process using tcp/ip
pipe: starts new process using linux pipes
mpi: starts new processes (see also config)
pvm:starts new processes (see also config)
dmn: connect to a running ctl-dmn process
logid: reference a member of group by logical id
pipes of channels:
chains: e.g. dmn | thread or thread | tcp: messages are passed through the chain
bridges: e.g. tcp > pipe > tcp or pipe < dmn: built direct socket connection (begin : end) either by connect/accept (>) o by accept/connect (<)
-n: decrease schedule-priority of component instance (effects only with remote linkage)
-s <remote_shell> [port]: set remote shell and used port (supported ssh, rsh), see also config
-T <milli-sec>: set time-out for first handshake while pipe and tcp link creation
-v: be (a little) verbose (see also flag -f)
-V <env-file>: read env-file instead of ctl.env
-x [terminal command]: start component in own terminal (default is xterm -e, see also config)
-X var1=value1 [, var2=value2]*: export variable to component

1.2 instantiate/owning reference/stream to one component instance

class ctl::link

non-trivial Ctors:
link(const location& loc) &&
link(const char* loc) &&
link(const std::string& loc): instantiate component given by loc // blocking
link(const char* filename, const char* mod) &&
link(const std::string& filename, const std::string& mod): open file <filename> for reading <mod>='r' or writing mod='w'

Dtor: a link is owner of the instantiated (and referenced) component //non blocking
template<typename T> ctl::oStream& operator<<(ctl::oStream&, const T&) // T serialisable && //non blocking
template<typename T> ctl::iStream& operator>>(ctl::iStream&, T&) // T serialisable: send/receive an object of type T to/from this link //blocking
bool operator!() const: return true iff initialised and valid //non blocking

1.3 instantiate/owning reference a connected group of remote components

class ctl::group

non-trivial Ctors
template<class locationContainer> group(const locationContainer&, ctl::cannelT=ctl::undefined) // locationContainer has size(), begin(), end() //blocking

bool run(): evaluate all incoming function calls until termination determined //blocking
link operator()[size_t logid] const: return link to the logid'th owned component //non blocking
size_t logId() const: return logical id of local component of this group
size_t size() const: return number of components of this group
Dtor: a group is owner of the instantiated (and referenced) components //blocking
bool operator!() const: return true iff initialised and valid //non blocking

template<typename X> X group::operator <op> (const X& x)
op in { +(sum) , *(product) , <(min) , >(max) , |(or) , &(and) , ^(xor) }
return result of op applied to set of local x //blocking

1.4 result handle of CI-functions/objects calls

template<typename Y> class ctl::resul**t**

Each CI call returns instead of the value of type Y, an object of type ctl::result<Y>.
non-trivial Ctors: all are non-blocking
result(const Y&)
Dtor: the destructor waits until the result is available // blocking
bool operator!() const: return true if result is available // non blocking
operator Y() const: waits until the result is available and returns it, saying e.g. Y y = ci.getY() means calling this blocking operator (and the Dtor) // blocking

1.5 global ctl::functions:

template<typename C> C&
ctl::createStatic(<ctor-args>)

create object by C(<ctor-args>), this object will be owned/destroyed by the ctl::environment

void <ci-scope>::use(const char* loc) &&
void <ci-scope>::use(const std::string& loc) &&
void <ci-scope>::use(const link& loc)
(ci-scope either CI-namespace, CI-class or CI-template),
let loc be default location for ci-scope
if “loc” starts with '+' , the rest string is used as modifier for default location
(set before or later)
otherwise overwriting earlier calls of use

void <ci-scope>::use()
unset default location

void <ci-scope>::use(link& loc)
if !loc get else set default location

template<class Impl> void <ci-class>::use(Impl* =0) &&
template<class Impl, class wrapper> void <ci-

class>::use(Impl* =0 ,wrapper* =0)
let Impl be the local default implementation of ci-class (using wrapper as connect wrapper)

commons of CI classes:
non-trivial Ctors // non blocking
Ctor call with local implementation // non blocking
bool operator!() const: return true iff initialised and valid // non blocking

serialisation of user defined types:
intrusive definition: CTL_Type(<class_name>, <abstraction>, <arg-list>, #arg-list)
</CTL_Template(<tmpl_name>, <tmpl-arg-list>, #tmpl-arg-list, <abstraction>, <arg-list>, #arg-list): abstraction in { string, array, vectorArray, listArray, setArray, mapArray,queueArray, tupel, reference, empty
extrusive definition:
CTL_setType/CTL_setTemplate
ctl.env: location map/environment variables/CTL_Monitor
const char* ctl::getcwd(): get logical working directory
link getClient(): get calling client inside CI implementation
Configuring the ctl: default debugger, default remote shell and port, MPI, PVM,...

2. The Component Interface (CI)

CTL_<macros>
#define CTL_Library <lib-name>
#define CTL_SubLibrary <lib-name>
#define CTL_SubSubLibrary <lib-name>
#define CTL_SubSubSubLibrary <lib-name>
name a (nested) CI-namespace

#include CTL_LibBegin
#include CTL_LibEnd:
open/close (nested) CI-namespace

#define CTL_Function<id> <y, f-name, (arg-list), #arg-list>
#define CTL_FunctionTmpl<id>: <y, f-name, (arg-list), #arg-list, (t-param-list), #t-param-list>
declare a function/template inside namespace

CTL_ClassFwd(<class-name>)
CTL_TemplateFwd(<tmplname, (t-param-list), #t-param-list>)
forward declaration of a CI-class/template

#define CTL_Class <classname>
name a class #define

#define CTL_ClassTmpl<tmplname, (t-param-list), #t-param-list>:
name a CI-class-template

#define CTL_Extends
inherited classes

#include CTL_ClassBegin,
#include CTL_ClassEnd
open/close a class/template scope

#define CTL_Constructor<id>(<list>), #list:
declare a constructor

#define CTL_Method<id>(<list>)[const], #list
declare a [const] method

#define CTL_StaticMethod<id>(<list>), #list
declare a static method

#define CTL_<functor><id>Throws(<list>), #list: add throw list to function/template, ctor, method. static method

(tmpl, (<list>), #list): macro to hide ',' in template-arg-list

Abstract Types:
bool, char≡int1, int2, int4, int8, unsigned char ≡ uint1, uint2, uint4, uint8, float≡real4, double≡real8

array<T>≡array<T, _>: linear container of T's of variable size
array<T, dim1, dim2,...> where dim1, dim2,... > -2: multidimensional container, dim<id>≡-1≡_ then dimension<id> is variable
tupel<T0, T1, …, Tmax>: aggregation of T<id>'s
cstring<T>: zero terminated container of T's
string=cstring<char>, wstring=cstring<wchar>
reference<T>: reference to T (includes typename + avoids copies in stream)
any: like polymorph reference, see ctl::any
const modifier, not const ==> per reference returning void && all args const && no explicite exception ==> one way call

3. Connecting Interface:Implementation

#define CTL_Connect/F/C/P: connect CI to C++/Fortran/C/Pascal implementation

#define CTL_nMT
helpfull iff implementation is not multithread save
void CTL_connect()
{
void ctl::connect<CI, Impl, [wrapper]>() &&
void CI::connect<[wrapper]>();
//implement all Ctors/methods/static methods of CI by Impl [using wrapper for overload resolution]
(CI_tmpl are connected by its instantiations)

ctl::connectR<CI, Impl, [wrapper]>();
//connect all Ctors/methods/static methods of CI + all methods/static methods of its base classes [using wrapper for overload resolution] by Impl

void ci-library::connect<mid>(impl-functor)
void ci-library::connect<mid, t-param>(impl-functor)
implement the function/-template of ci-library with id mid by impl-functor
}

connect wrapper
struct <wrapper>
{
CTL_Constructor(cid, (arg-list), #arglist)
CTL_Method(mid, y, (arg-list) [const], #arglist)
CTL_StaticMethod(mid, y, (arg-list), #arglist)
};

ctl.env and ./ctl.env
Defining variables inside the component, e.g.
CTL_Monitor = -x -l thread
<ci-scope> → "location-str"
include <filename>

configuration of libctl
-DCTL_MPI support MPI linkage
-DCTL_PVM support PVM linkage
-DCTL_LocatorCI <location>
-DCTL_Shell remote shell in {ssh[default] rsh}
-DCTL_ShellPort [ssh=22, rsh=414]
-DCTL_DBG [gdb]
-DCTL_Terminal[xterm]

4. Examples