SafeTL — C++ Safe Template Library

with auxiliary components,
version 2.09 (11.12.2008)

Smart Pointers:
    Active Smart Pointer (Value and Copy Semantics) and
    Passive Smart Pointer (Pointer or Reference Semantics),
Containers (Value and Copy Semantics):
    Ordinary One-Dimensional Array with Bounds Checking,
    Blocked One-Dimensional Array with Bounds Checking and
        Capability to Graw without Reallocating,
    Rectangular Many-Dimensional Array with Bounds Checking,
    Doubly-Linked List,
Debug, Output, and Assertion Checking Utilities.

User's Guide

I. B. Smirnov

The generation date of this version of this document: December 11, 2008.

Contents

1 Introduction

This is a description of some utility packages which are proved to be very useful for program development in `C++`.

The most interesing components are co-called active and passive pointers, which represent an original programming methodology. The Active smart pointer owns the object, clones it at copying itself and deletes it at the deletion of self. This type of pointers is sometimes mentioned in the literature as cloning pointers or deeply cloning pointers. The Passive smart pointer is completely opposite to the Active one. The Passive pointer does not clone or delete the object, but it allows the object to be deleted by the other means and if this happens, the Passive pointer to it is annuled automatically, thus preventing a dangerous effect known as dangling pointers.

The following collection of arrays and lists of various types is adapted to work with these pointers.

Dynamic Linear Arrays represent independently made one-dimensional arrays with variable size. In essence Dynamic Linear Arrays are simply another variant of what is today widely known as "vectors" (or more exactly, the "vectors" is an erratic notation of that which we always knew as one-dimensional arrays with variable size). There are differences between the Dynamic Linear Arrays and the vectors, some of them are noticeable, but still not "fundamental".

There is a different type whose notation does not have suffix "Linear": Dynamic Arrays. They are multi-dimensional arrays like what is typically defined by regular way as [...][...][...]..., that is these are rectangle multi-dimensional arrays. They are quite equivalent to what is known in Fortran-77 as multi-dimensional arrays. We should remark that C++ gives possibility to construct multi-dimensional arrays from Dynamic Linear Arrays. Indeed one can put one Dynamic Linear Array as element of another Dynamic Linear Array, that as element of third etc. But then the dimension of each such array could be unique and may differ from the array kept at neighboring array element. This is good scheem, but it is different concept. Dynamic Array is strictly rectangular. But internally Dynamic Arrays are anyway implemented through Dynamic Linear Arrays. Each Dynamic Array has inside one big Dynamic Linear Array and provides interface to deal with multiple indexes, as well as fast filling as a whole. I do not know whether STL has an analogue.

The both arrays check boundaries at indexing, but these checks can be switched off if the speed is more important than reliability.

"Block array" is equivalent to Dynamic Linear Array, but it is allocated by blocks and can grow without copying of content between chunks of free memory. Instead, a new chuch is reserved in addition and as continuation for the content of the previously reserved ones. Consequently the elements are allocated not completely sequentially, and cannot be accessed as that of built-in arrays. Internally Block Array is implemented through Dynamic Linear Arrays. I do not know whether STL has an analogue.

An Abstract List is a simple and reliable implementation of what was always known as is: lists, namely doubly linked lists. For each new element a uniqure chunk of memory is reserved.

The author does not include to this library other types of containers, although he has written and eventually used at least several others. But they are still rarely used or can be easily built from these which supplied or they are not generic enough or they might have good equivalents in STL, or all together. For example, "maps" from STL is a good variant for consideration, when one needs fast random access via generic non-numerical or non-sequencial indexes.

Special output utilities allow to make output listings with indentation. Indentation simplifies greatly reading and understanding of long listings. For example, when printing a dump of an object, it is reasonable to print its class type and then increase indentation by, say, two blanks. After all object members are printed, the length of indentation should be restored to previous value. This allows to distinguish visually all object internals from externals.

Debuging utilities allow to detect errors in the form of special assertion checks and print detailed protocols with detalization of condition, the values of interesting variables, the name of current function and the stack of previously called functions. All taken together frequently allow to guess about the reason of the error without additional run of the program under debugger.

The author admits that there may be a lot of typos and missings in this manual. It is assumed in development stage. Currently this manual represents some parts simply extracted from the manual of HEED++ and slightly corrected.

The full SAFETL package can be used according to GNU Lesser General Public License version 2.1, but some components can be used more freely according to licenses printed in the beginning of the corresponding files.

The program is provided "as is" without express or implied warranty.

2 Dependencies on other software

There is no obvious dependencies on non-standard software. The packages are exploited mainly at Linux of various versions. But recently they have also been installed at Windows with Microsoft Visual Studio C++ and they seem to work perfectly in this environment as well.

The commands for compilation at UNIX are written for (t)csh command interpreter, "make" command and g++ compiler.

The commands for compilation at Windows are written for `CMD.EXE` command interpreter and mentioned Microsoft Visual Studio batch compiler.

3 Structure and directories

The main directory is called "wcpplib", which was deemed to mean "the working C++ library". In it five directories reside:

- `stream` – handling streams and indentation for output listings.
- `util` – miscellaneous utility classes and macro-definitions, in particular, debug facilities and facilities for emulation of some operators of the new C++ standard for old compilers.
- `safetl` – SafeTL or safe template library: special smart pointers (see appendix in [1], [2], and section 9.2 farther in this guide), dynamic arrays and lists;
- `install` – installation utilities, currently with a single program "`concom`" which simplifies interface with make-related staff at UNUX. All "`build.b`" scripts use this program. Therefore it should be compiled first prior to wcpplib by a regular script (which is trivial and it is supplied) and put into directory registered in the `PATH` environment variable. Of course, `concom` does not depend on the rest of wcpplib, but wcpplib at UNUX is compiled with the help of the executable `concom`.
- `doc` – GNU LGPL license (covering a part of this software), small installation summary, file `version.txt` determining the version number, and file `date_of_safetl_arch_creation.txt` determining the date of creation of this package, that is the date when the source files were put into archive by running corresponding script (they, of course, are created earlier themselves). These version number and date should (reasonably) correspond to what you see at the title page of this manual.

The following important files reside in `wcpplib` directory:

- `buildhd.b` – building (compiling and making the libraries) for the total `wcpplib` for UNIX.
- `clearbin.b` – removing all binary libraries, compiled files ("object files") and template database (for those systems where it exists) for UNIX

- `build_vs.bat` – building in Windows (compiling and making the libraries for the part of `wcpplib`, which was ported to Windows.
- `clearbin_vs.bat` – removing all binary libraries, compiled files ("object files") and template database (for those systems where it exists) in Windows.

4 File types and extensions

The following rules are used for naming files.

- `*.c` – C++ source code (this extension is usually reserved for C-language in other programs; since the author does not use the plain C language in this program, mistakes are impossible; shorter names are more convenient for handling disk files).
- `*.ic` – inlined C++ code.
- `*.h` – header files for C++ code.
- `*.o` – compiled source files ("object files") for UNIX.
- `*.obj` – compiled source files ("object files") for Windows. Windows also generates additional service files `*.ilk` and `*.pdb`.
- `buildst.b` – scripts for compilation of source files in a given directory or its subdirectories and building the libraries for UNIX.
- `buildst_vs.bat` – scripts fot compilation of source files in a given directory or its subdirectories and building the libraries for Windows Microsoft Visual Studio C++.
- `lib*.a` – static library of "object files" for UNIX.
- `lib*.lib` – static library of "object files" for Windows.
- `lib*.so` – dynamic libraries for UNIX. Today only one such library is presented (because it is necessary). This is `libRootMGHist.so` loaded in ROOT in order to make it able to read GHist-created histograms.
- `AnyName_ts.c` – test program for testing the class or the package indicated by `AnyName`.
- `AnyName_ts.b` – script for compilation and linking the test program for UNIX.
- `AnyName_vs_ts.bat` – script for compilation and linking the test program for Windows.
- `AnyName_ts.e` – executable file of the test program in UNIX.
- `AnyName_vs_ts.exe` – executable file of the test program in Windows.
- `AnyName_ts.r` – script for running the test program that produces very long output listing. Only for UNIX. The sole mission of these scripts is to delete old file with listing, to run the executable file and to redirect new output to this file.
- `AnyName_ts.out` – output listings of test programs, which are generated by *.r scripts.
- `AnyName_ts.eout` – output listings of test programs supplied by the author in order to allow the user to compare output at his computer with it. Use command
  `diff AnyName_ts.eout AnyName_ts.out`
  in UNIX or
  `FC AnyName_ts.eout AnyName_ts.out`
  in Windows.

- `*.mktar` – scripts to make tar-archives. The archives supplied were made with these scripts. As the reader could see from the text of these files, before archiving these scripts are copied into the header archived directory and then they are archived as ordinary files for the purpose of investigation of various archiving missings and bugs.

5 Installation

We will consider in this section installation at UNIX-like systems.
It is meant installation of source code and its compilation, link and running.
The installations sequense consists approximately of the following:

1. Copy from my internet page `http://cern.ch/ismirnov/heed/` and install in your working directory a file `safetl.tar.gz`. (unless you have done that already).

2. Unpack it with `gunzip filename` and `tar -xvf filename` commands. You will see a directory tree rooted from `wcpplib` .

3. Enter to subdirectory `wcpplib/install` and run stript `concom.b` . This will compile and link a little program `concom.c`, which is used for simple generation of makefiles. The executable version should now appear in the same directory with name `wcpplib/install/concom.e` . But it is not mandatory that it resides there. It can be placed in any directory included in your `PATH` environment variable and have name `concom` without extension. So, after you find an appropriate place for it, you please rename it accordingly. Also make sure that it is seen by your shell. You can check this by command "`which concom` " . It sometimes happens that shell does not see new commands which appear after its launching. If this is the case, you need either to re-run the shell or to use shell built-in command `rehash`.

4. Look in chapter 6, and take a short look in chapter 7 of this manual.

5. Take file `wcpplib/install/safetl_setup.b`, put it or its content in convenient place and modify it accordingly to your preferences.

6. Run it by "source".

7. Change to some small and "trivial" enough subdirectory of wcpplib, for example to `wcpplib/stream` and run build script (`buildst.b` or `build.b`). It should compile the source files and put them into library file. In some rare cases instead of "archiving" a library such scripts simply copy the "object" files to upper directory, from which they are took later on at linking other programs.

   Look what `build*.b` says and depending on that you might need to edit the content of `wcpplib/install/safetl_setup.b` additionally, then you are advised to clean binaries by `rm *.o` (because the "mechanics" of `build*.b` does not usually detect the changes of environment variables) and to run `build.b` again.

   Also be aware that I do not normally check the functioning of all these compilation and linking scripts in all mossible modes and as if they are installed on another's computer, clean of anything pre-installed. Because of huge number of various possible combinations of environment variables, scripts, files to compile, test programs to link, directories, computer environments, operating systems, compilers, librariries it is above my possibilities to test every possible way of work. I mainly supply the packages as they are currently (or may be with some delay) work at the computers at which I am running them for my purposes. Therefore you might find in scripts and these instructions any my problems as well, starting from missing some files in supplied archives (about which you are kindly invited to inform me) and to any problems in scripts resulted to compilation or linking errors (about which you are also invited to inform me, if you have constructive information, but not just that "it does not work").

8. When "a sample" subdirectory is compiled successfully, change to root directory of wcpplib, that is to `.../wcpplib`, and run `buildst.b`. Again you might need to adjust your environment and run `wcpplib/clearbin.b` script in order to clean before the next attempt of full rebuild.

9. When you finish this, you may test if this stuff links and works at your computer. For this change to `wcpplib/stream`,
   run `prstream_ts.b`,
   run `prstream_ts.e`,
   run `prstream_ts.e > ttt.txt`,
   run `diff prstream_ts.eout ttt.txt`.
   If you see difference, something could be wrong. For this trivial test it is unlikely that any difference could be detected. For some much more complicated tests diff sometimes detects differences, for example when I change the code and forget to rewrite *.eout files, or when a raw machine addresses are output — they are naturally to be different in different computer. Therefore study and interpret the output of diff according to common sence.

   Much more complicated tests could be done by changing to `wcpplib/safetl`, and running `run_all_tests_ex.b`. This will try many tests, that is link many tests programs, run each and and for each run diff to compare results. In the contrast to the procedure described above this script does all the procedure automatically. One of the programs is run by it many times in different modes. During these trials its error-detection system triggers various errors, which should coincide with what is expected. You have to analyse the results which are output to your screen. It is usually clear whether everything is OK or not OK. If diff detects difference, its output is quite obvious. The tests themselves takes a few seconds of computer time.

6 Environment variables

Many scripts used for compilation and linking require some environment variables to be set. In this chapter we will describe them.

6.1 UNIX

For compilation and link of `wcpplib` you will need the following UNIX environment variables installed:

- `CPPMAKEOPT` – is read in program concom (using a C++ function `getenv(char* name_of_var) )` and is supplied to cpp (C preprocessor) in order to instruct it to output a rule suitable for make. Recommended values:

```
setenv CPPMAKEOPT "-MM"     # linux
setenv CPPMAKEOPT "-xM"     # Solaris
```

  If this variable is not defined, `concom` will use `-M`.

- `CPPCOMPILE` – compilation command
  `CPPCOMPILED` – command for compilation with debug information
  `CPPLINK` – compilation and link command
  `CPPLINKD` – command for compilation and link with debug information
  Normally used in `build.b` scripts as the first argument of call of `concom`.

  Recommended values (for a few typical systems):

– Scientific Linux, version 4.1:

```
setenv CPPCOMPILE "g++ -c  "
setenv CPPLINK    "g++     "
setenv CPPCOMPILED "g++ -g -c  "
setenv CPPLINKD    "g++ -g  "
```

– Linux, RedHat version 8.0:

```
setenv CPPCOMPILE "g++ -DCOPY_RETURNS_COMMON_BASE -c  "
setenv CPPLINK    "g++ -DCOPY_RETURNS_COMMON_BASE   "
setenv CPPCOMPILED "g++ -DCOPY_RETURNS_COMMON_BASE -g -c  "
setenv CPPLINKD    "g++ -DCOPY_RETURNS_COMMON_BASE -g  "
```

(Remark: this makes all classes handled by active pointers derived from a special common base class. This is needed in order to bypass a missing feature in older GNU compilers. The test program `wcpplib/safetl/ActivePtrT_ts.b` has to be compiled without `-DCOPY_RETURNS_COMMON_BASE`. This program is concieved as trivial example, which does not need derivation from common base and for clarity it does not contain macros that normally allow to switch between this derivation and non of it.)

– Linux, RedHat older versions:

```
setenv CPPCOMPILE "g++  -fhuge-objects -c  "
setenv CPPLINK    "g++ -fhuge-objects  "
setenv CPPCOMPILED "g++ -fhuge-objects -g -c  "
setenv CPPLINKD    "g++ -fhuge-objects -g  "
```

– Solaris and most of the others:

```
setenv CPPCOMPILE "CC -c"
setenv CPPCOMPILED "CC -g -c"
setenv CPPLINK  "CC "
setenv CPPLINKD  "CC -g"
```

- `ARCOMMAND` – command for making libraries of wcpplib and heed (archives of object files). Recommended value:

```
setenv ARCOMMAND "ar -rs"
```

- `WCPPLIB` – location of the root directory of wcpplib for searching include files. The include files in wcpplib are included with prefix `wcpplib/subdirectory/...` like the following: “`#include "wcpplib/safetl/AbsArr.h"`”. This allows to avoid mess between the local files and possible system files with the same names (although no such coincidences are ever noticed). This variable is used in `build.b` scripts as argument of `concom` command, which passes it to preprocessor, `make`, and compiler. The example of initializstion:

```
setenv WCPPLIB $HOME
```

(if subdirectory wcpplib is located right in `HOME`)

- `WCPPLIBOBJ` - location of libraries and some object files of wcpplib. Used at link of executable programs. For example:

```
setenv WCPPLIBOBJ $WCPPLIB/wcpplib
```

6.2 For Windows and Visual Studio

The bat-files for Windows were not updated from previous version, so they might need some manual intervention of the user. Below I repeat the old text, which was also not updated.

As Windows users know, the Windows environment variables can be initialized through screen buttons (in backward translation from Russian)
`Start->System->Additional->Enviroment variables->Enviroment variables for user`
or in interpreter by command
`set var_name=var_value`.
In the following examples the second form are printed. As already was mentioned, it is assumed that the Visual Studio C++ environments are also set in the working command interpreter. This can be done by running the command shell from the Tools menu of the Visual Studio window. The examples are tested under Visual Studio C++ version 8 in Windows XP.

- `WCPPLIB` – location of the root directory of wcpplib for searching include files. The include files in wcpplib are included with prefix `wcpplib/subdirectory/...` like the following: "`#include "wcpplib/safetl/AbsArr.h"`". This allows to avoid mess between the local files and possible system files with the same names (although no such coincidences are ever noticed).

  The example of initialization:

  ```
  set WCPPLIB=''C:\Documents and Settings\user name\working directory name''
  ```

- `WCPPLIBOBJ` - location of libraries and some object files of wcpplib. Used at link of executable programs. For example:

  ```
  set WCPPLIB=''C:\Documents and Settings\user name\working directory name\wcpplib''
  ```

- `VSCPPCOMPOPT` – additional compiler options. This variable was initially included in order to switch between the linked libraries. If your main program is being debugged, you might want to include debug information in your libraries as well. If you compile "release" version of main program, you might want to exclude the debug info from the libraries either. Anyway, my experience (incompleted so far) shows that if one use different types of debug information, the program will not be linked. Moreover, the difference in speed of calculations in debug and release configurations is huge (unlike Linux). Therefore I have included this variable in every compilation and linking script. This allows to change the debug information simultaneously everywhere (except the main Visual Studio Environment, in which the user should edit `Tools->Progect pages` system of menu manually). Example of this variable:

  ```
  set VSCPPCOMPOPT=/MTd
  ```

  for debug and

  ```
  set VSCPPCOMPOPT=/MT
  ```

  for release configuration.

7 Macro variables

There are some marco variables (macros) which may be changed in order to adapt wcpplib for particular system, compiler and envorinment. At contemporary systems there may be no necessity to change them. Some of them were not used during long time. But we will describe all of them here for completness.

7.1 Emulation of new C++ standard at old systems

The file `wcpplib/util/emul_new_stand.h`:

- `S_ENWSD` – if defined, the programs will be switched to emulation of cast operators: instead of `static_cast` as well as `dynamic_cast` it will be direct type transformations: `((type)(name))`.
- `S_NO_TYPEID` – if defined, there will be no dynamic type identification and therefore no `dynamic_cast`. Instead of `dynamic_cast` `static_cast` will be inserted. It works well sometimes, but it depends on program. For some programs it cannot work categorically.
- `S_EEXCEPT` – Sign to Emulate Exceptions. If defined, they are emulated, but this happens in a package, which is not supplied.
- `S_EMUTABLE` – sign to emulate directive "mutable". If defined, they are emilated (in theory it is not good style, but some old compilers were unable to understand them, so I needed to apply this somewhere).

This file also issues directive `using namespace std`, which appears to necessary at newest systems.

7.2 Strings

The development of wcpplib was started before the standard library was firmly established, and when there were many other options and it was not clear which of them will predominate. Wcpplib does not depend on sophisticated operations with strings. The strings are used mainly for notation of various objects and files. The typical operations are initialization from characters array, convertion to characters array, initialization from integer number, and addition of strings. Such simple operations were expected to be supported by any sensible library. Therefore wcpplib was based on references to a symbolic name `String`, which was expected to be redefined somehow to a specific package in file `wcpplib/util/String.h`. Unfortunalety, STL, which dominates now, differs from sensible libraries by syntax of convertion from string to character array (notorious `.c_str()`). Therefore when strings are disappeared from CLHEP, I needed to edit many files and I included switches like this:

```
#ifdef USE_STLSTRING
  ofstream file(file_name.c_str());
#else
  ofstream file(file_name);
#endif
```

Such things still remain in many files. `USE_STLSTRING` is a macro defined in file `wcpplib/util/String.h` and meaning that std::string is invoked at the place of `String`:

```
typedef std::string String;
```

In this file also useful conversion fron `long` to `string` is defined:

```
inline String long_to_String(long n)
{ std::ostrstream s; s << n <<'\0'; return String(s.str()); }
```

7.3 Smart pointers

All the programs are based an a system of two ortogonal (in some sense) smart pointers, which are quite complicated technically (ideas are described in [1, 2] and in section 9.2). They are defined in file `wcpplib/safetl/AbsPtr.h`. There are many macro variables and their descriptions. One variable which might need adjustment is `COPY_RETURNS_COMMON_BASE`. If it is defined, it permits to bypass a problem in old GNU compilers. When it is defined, it leads to defition of some other macro variables and `class CommonBase`, from which all classes pointed by `ActivePtr` have to be derived. This all is made automatically. So at GNU gcc compilers with versions 3.2 and higher this variable can be undefined (that is "define" is commented). At older compilers it perhaps have to be defined. If it has to be defined, but it is not, the compiler should say:

```
sorry, not implemented:
adjusting pointers for covariant returns
```

The file `wcpplib/safetl/AbsPtr.h` contains much more macro-variables, but all they are assumed to be adjusted to the optimal values in the supplied version. In case of doubts see comments around them in this file.

7.4 Templated arrays

Some basic information is given in Introduction to this manual. Below are some details.

The linear array (which is frequntly called "vector" by C++-programmers nowadays) is called here `DynLinArr` (dynamic linear array) and is defined in `wcpplib/safetl/AbsArr.h` (and, of course, in `....c`, the name `AbsArr` is translated as an abstract array). The violation of boundaries by the array index is checked at each access, if the macro `ALR_CHECK_BOUND` is defined. Of course, the check takes some time at execution, but it appears that in real physical Monte Carlo simulations the slowing down of calculations is barely noticed. Also note that at Linux it is not possible to switch this macro on in one files and off in other (at least the author was not able to do this): it will appear either switched on or off in the whole program. Therefore this variable is just permanently defined just in `wcpplib/safetl/AbsArr.h` and hence the boundaries are always checked.

The program uses also many-dimensional array `DynArr`, defined in the same files. It can have as many dimensions as necessary. Index by each dimension can run from 0 to some fixed for this dimension boundary. There is another macro name assotiated with it: `ALR_CHECK_EACH_BOUND`. Like `ALR_CHECK_BOUND` it is set permanently in `wcpplib/safetl/AbsArr.h`. But user can change the both. Actually `DynArr` is inplemented through `DynLinArr`. If `ALR_CHECK_EACH_BOUND` is not set, only the total boundaries are checked. This guarantees only the the element will be inside `DynArr`, but not necessary at the correct place: the boundary of particular index can be violated. If `ALR_CHECK_EACH_BOUND` is set, each index is checked separately, and the total boundaries are not checked.

Both `DynLinArr` and `DynArr` have functions (`acu()`) for access without check at all (independently on setting of described macros). Use them with caution.

The program uses another one-dimensional array that rises without copying the content (it can allocate elements in separated blocks, many elements per block, new elements can be put in another new block), see files `wcpplib/safetl/BlkArr.h(.c)` (translated as block array).

The program uses doubly linked lists, see files `wcpplib/safetl/AbsList.h(.c)` (abstract list). All arrays use `DynLinArr` internally as a driver and are checked, if the linear array is checked. The lists are checked always.

7.5 Streams

Two default stream notations are used thorought the program: `mcout` and `mcerr`. (These notations actually mean "my cout" and "my cerr".) The both are macro-variables defined in file `wcpplib/stream/prstream.h`. Currently the both are substituted by `cout`.

7.6 Run-time error detection

This package is designed for checking assertions and printing clear diagnostic in the case of error. It is defined in file `wcpplib/util/FunNameStack.h`. It can be partially switched on or off (the latter is provided in order to speed up the calculations, but no noticable effect can usually be detected in real programs, in which various kinds of calculations are mixed).

- FUNNAMESTACK – writting the function name in internal stack, when the macro call `mfunname("function name in any readable form")` is invoked. Does not make effect for permanent variant of this call `mfunnamep("function name in any readable form")`. (the permanent variant is usually used in the source text if there are assertions right in this function; the assertion requires either of them for successful compilation, when DO_CHECKS is defined, see below). Such calls are usually placed right after the title of the function. The phycical copying of string does not occur. Only the address of a static string is memorized in the stack. Thus there is not noticable effect for calculation speed. The stack is usually printed after triggering error by an assertion statement. Thus the user can instantly see the name of the function in which error is detected and all the functions through which this function was called, which is a precious feature. If the function was compiled with `-g` option, the program will usually print also the file name and the line number. The program prints also the condition that triggers error, the variable names involved and their values. So in most of the cases there is no need to run the program second time under debugger, to reproduce the situation, and to recover the information from the debugger. After these printing the program normally flushes the streams and stops (terminates). The program can also throw an exception, if the value of a global variable `int s_throw_exception_in_spexit` is equal to one. The assertions can also be switched on or off for speeding up with the help of the next macro.

  At multithreading environment the keeping of stack with function names is greatly complicated. In the case of multithreading the package needs to keep separated function name stack for each thread. For this purpose the package is equipped with the methods to identify the current thread at the each call of any of its functions. At the output it prints the name stack of that thread, in which the error is detected. These possibilities are switched on by macro USE_BOOST_MULTITHREADING.

  Another complicated issue is the time when all these stacks must be initialized. The general rule is that stack is initialized at the moment, when any of functions detects the new thread. This is done through a singleton model known in literature as "GoF" approach. Correspondingly the header list of stacks is never deleted, because it is impossible to determine the moment, when all the registered functions are passed and no one will ever be called at the program termination. Therefore memory checkers could find some non-freed memory after the program termination, but this is inavoidable.

  The general advice: never use multithreading unless you have a lot of free time, which you urgently need to make busy.

  Also note that in the source of this package there are come controls for an original memory checker `clean_new`, but it is not supplied, and these switches should be ignored.

- DO_CHECKS – if defined, check the assertions of the type `check_econd(...)` (condition for triggering error with further termination), or `check_wcond(...)` (condition for issuing warning without program termination), or exclude them from the source text otherwise (this however does not make noticeable effect on speed).

7.7 Inline or not inline?

In some sub-directories the functions which can possibly be inline (if they are very small and simple and without loops), were declared by such a way, that they can be switched and appear as either inline or non-inline. First of all, the word `inline` was replaced in the source text by the word "`wl_inline`". This latter is a macro defined in the file `wcpplib/util/inlinec.h` as either empty macro or "`#define wl_inline inline`". The total file now loook by the following way:

```
//#define WCPPLIB_NO_INLINE
```

```
#ifdef WCPPLIB_NO_INLINE
#undef WCPPLIB_INLINE
#define wl_inline
#else
#define WCPPLIB_INLINE
#define wl_inline inline
#endif
```

So with the commented first line (as it is now) the empty macro `WCPPLIB_INLINE` is defined and the macro `wl_inline` means `inline`. The inlined functions are placed in files `*.ic`, which are not directly compiled. In the ends of the header files there are such sentences:

```
#ifdef WCPPLIB_INLINE
#include "wcpplib/subdirectory/filename.ic"
#endif
```

So in the case of inline, the files are added at the end of the header files. In source files there are opposite sentences:

```
#ifndef WCPPLIB_INLINE
#include "util/FunNameStack.ic"
#endif
```

and these "inlined" functions get non-inlined. In theory, the inlining leads to faster execution and more memory expense, where as non-inlining causes the opposite effect. But no considerable differenses were ever registered. So I am always working with inlining.

8 Compilation scripts

The program is too large to be re-compiled completely each time a single change is made. This would not be convenient if some program development is going on. Therefore the selective compilation of new, changed or affected files is actual.

In UNIX the selective compilation is usually made by the "`make`" programm, which interprete make-files. The latter describe file dependencies and commands to execute. However, the program HEED seems also to be too large for writting its make-files manually.

More or less automatic generation of make files is possible through the Autoconf-automake system. But using this system assumes getting skilled in about 360 pages of their non-trivial manuals, which is too large for author's busy schedule. In addition, its rules and scripts do not look transparent enough, and in general it seems to be designed rather for system-related works, than for pure calculations. For latter it seems to be somewhat redundant.

The acceptable alternative solution, which permits us to write very small and clear compilation scripts, is provided by the possibility to generate the dependence instructions automatically with `g++` compilers.

Unfortunately no such possibility is found for Windows Visual Studio compiler. There are some " `nmake`" utility. The author did not study it in detail so far, but it seems to have the same problems as UNIX' `make`. Therefore for Windows compiler only the scripts which perform the full compilation of each sub-package of `wcpplib`, as well as the script, which perform the full compilation of the total `wcpplib` (selected subpackages; this script simply calls each sub-package script one by one) are provided.

But in Linux (in which we can use of mentioned automatic generation of dependence instructions) it is possible to create a very simple program (currently 265 lines in C++ only, including reach comments!) working as a frontend of both the compiler and the "make' program and allowing one to write the compilation command for selective compilation for all necessary files (new, changed, or affected by new or changed ones) in a single line! Calling such a program one has just to provide as its arguments the compilation command with all its arguments and supply a list of source files (`*.c` files, `*.h` are not mentioned at all, which greatly simplifies the work, and each `*.c` is mentioned only once, which also simplifies everything; however, if one wants to compile a library, one need to mention equivalent object file two times more, see below, but this is tolerable complication). The program is called "`concom`", which is translated as "connect commands" (may be not very felicitous name, but it's lazy to change it in very many places, where it is already imprinted).

Each compilation script typically has call of this program. Another couple of lines of typical compilation script is the instruction for the ordinary make program to create library from the object files, of course, in the cases, when the library is necessary. And one else attribute of the compilation scripts is running the "make" itself. Let us consider a simple real example, but first we describe the syntax of `concom`.

The `concom` program takes the following arguments:

1 The name of compilation command (e.g. `$CPPCOMPILED`). Note that this name can also contain some compiler options, normally general for the full compiled system.

2–n where $n+1$ is the argument number (index) with one of the following extensions: `.c`, `.cpp`, `.CC`, `.f`, `.f77`, `.for`, `.F`. All the arguments from 2-nd to n-th are interpreted as an additional compilation options and are passed to preprocessor and compiler. Obviously the program simply scans arguments one by one and stops at the first argument with one of the listed extensions.

n+1... The source files to compile. All they should have one of the mentioned extensions. At least one source file should be given. But normally all source files to be compiled with identical options are given in one line.

The program `concom` processes the files one by one, for each it generates with the help of preprocessor the dependence instruction in `make` format, which is output to default stream. Its redirection to makefile is done manually in `build.b`, see the example below.

So, the typical building script, creating library libmatter.a and also writing in a certain directory a separated file `GasLib.o` (the file names are related to HEED) is given in the following real example:

```
#! /usr/bin/env tcsh
rm makefile
cat > makefile <<EOF1
#This makefile is created by build.b
EOF1
#wc particle_def.h eparticle.h lparticle.h particle_def.c eparticle.c lparticle.
c
cat >> makefile <<EOF2
$WCPPLIBOBJ/libmatter.a: AtomDef.o MoleculeDef.o MatterDef.o GasDef.o
        $ARCOMMAND $WCPPLIBOBJ/libmatter.a AtomDef.o MoleculeDef.o MatterDef.o G
asDef.o
EOF2
cat >> makefile <<EOF3
$WCPPLIBOBJ/GasLib.o: GasLib.o
        cp  GasLib.o $WCPPLIBOBJ/GasLib.o
EOF3

concom $CPPCOMPILED  -I$WCPPLIB  -DFUNNAMESTACK -DDO_CHECKS \
AtomDef.c MoleculeDef.c MatterDef.c GasDef.c GasLib.c >> makefile
if ($status != 0) then
    echo "Processing of source files by concom failed"
    exit
endif


make -r $WCPPLIBOBJ/libmatter.a
```

```
make -r $WCPPLIBOBJ/GasLib.o
```

These 27 lines are converted by `concom` into a 150-line makefile, starting as follows (we will not reproduce it completely):

```
#This makefile is created by build.b
/home/ismirnov/s06/wcpplib/libmatter.a: AtomDef.o MoleculeDef.o MatterDef.o GasDef.o
ar -rs /home/ismirnov/s06/wcpplib/libmatter.a AtomDef.o MoleculeDef.o MatterDef.o GasDef.o
/home/ismirnov/s06/wcpplib/GasLib.o: GasLib.o
cp  GasLib.o /home/ismirnov/s06/wcpplib/GasLib.o
AtomDef.o: AtomDef.c /home/ismirnov/s06/wcpplib/matter/AtomDef.h \
  /home/ismirnov/s06/wcpplib/util/String.h \
  /home/ismirnov/s06/wcpplib/safetl/AbsArr.h \
  /home/ismirnov/s06/wcpplib/stream/prstream.h \
  /home/ismirnov/s06/wcpplib/util/FunNameStack.h \
  /home/ismirnov/s06/wcpplib/stream/prstream.h \
  /home/ismirnov/s06/wcpplib/util/inlinec.h \
  /home/ismirnov/s06/wcpplib/util/FunNameStack.ic \
  /home/ismirnov/s06/wcpplib/util/emul_new_stand.h \
  /home/ismirnov/s06/wcpplib/util/String.h \
  /home/ismirnov/s06/wcpplib/safetl/AbsPtr.h \
  /home/ismirnov/s06/wcpplib/math/minmax.h \
  /home/ismirnov/s06/wcpplib/stream/prstream.h \
  /home/ismirnov/s06/wcpplib/util/FunNameStack.h \
  /home/ismirnov/s06/wcpplib/stream/definp.h \
  /home/ismirnov/s06/wcpplib/stream/prstream.h \
  /home/ismirnov/s06/wcpplib/util/String.h \
  /home/ismirnov/s06/wcpplib/util/FunNameStack.h \
  /home/ismirnov/s06/wcpplib/safetl/AbsArr.h \
  /home/ismirnov/s06/wcpplib/safetl/AbsList.h \
  /home/ismirnov/s06/wcpplib/clhep_units/WPhysicalConstants.h \
  /home/ismirnov/s06/wcpplib/clhep_units/PhysicalConstants.h \
  /home/ismirnov/s06/wcpplib/clhep_units/WSystemOfUnits.h \
  /home/ismirnov/s06/wcpplib/clhep_units/SystemOfUnits.h \
  /home/ismirnov/s06/wcpplib/stream/prstream.h \
  /home/ismirnov/s06/wcpplib/util/FunNameStack.h
g++ -g -c -I/home/ismirnov/s06 -DFUNNAMESTACK -DDO_CHECKS AtomDef.c

MoleculeDef.o: MoleculeDef.c \
...
```

Note that directly in the file `AtomDef.c` only the following include statements are given:

```
#include <iomanip.h>
#include "wcpplib/matter/AtomDef.h"
#include "wcpplib/clhep_units/WPhysicalConstants.h"
#include "wcpplib/stream/prstream.h"
#include "wcpplib/util/FunNameStack.h"
```

Comparing this list with the dependence statement from the makefile one has to conclude that not only the explicitly included files are checked, but also those “included from included”. Thus, all the hierarcy is checked, except the system files (like `<iomanip.h>`). The latter is perhaps unnecessary to check. If system is changed, it is wise to recompile everything.

Surely it is much easily to write 27 lines than to write 150 lines! Such a system is also much more reliably with respect to manual typing dependencies. If you miss some dependency, the system does not detect this error and you will eventually obtain wrong outdated program. If the makefile is gererated automatically, such an error is not possible. You can also change the program, adding new dependencies at any moments, and they will be immediately be taken into account.

Thus, the combination of the `concom` program and the `build.b` script allows user to change freely any of the source or included files or to add new included files into the source files. This system detects it since it remakes "makefile" each time the `build.b` command is run. But the system does not check the modification of `build.b` itself. It cannot also detect in general case the change of environment variables on which it could depend. If it or environments are modified, I usually remove all "object files" from relevant directory by "`rm *.o`", or run `wcpplib/clearbin.b` to clean everything and then run `build.b` again.

All the compilation of files in particular directories is done in UNIX by scripts `buildst.b`. The script `wcpplib/buildst.b` compiles all wcpp library (in the described sense, that is it creates makefiles and runs "`make`"). It simply jumps to each necessary sub-directory and run its local `buildst.b`.

We repeat that the program `concom` is placed in `wcpplib/install` directory. It should be compiled prior to `wcpplib` and `heed++` by a regular script `concom.b` and put manyally into a directory registered in `PATH` environment variable. Of course, `concom` does not depend on the rest of wcpplib. For Windows all this is not valid. Just run `wcpplib/buildst_vs.bat`, and that's all.

9 Miscellaneous

9.1 The quantities, indexes, and arrays

For the sake of clarity and consistency the author follows a rule that the numbers of something are started from `q` or `q_`. Most frequantly such variables are sizes of arrays or the number of valid elements in arrays (which are not always the same: the second can be smaller than the first). The character "q" is very convenient to use in order to distinguish the *quantities* from the *indexes*, the number of something from the something's number (at least for a non-native English speaker). The indexes and the something's numbers are usually started from characters `n,i,k,l,n,m`.

Then, if `q_something`, i.e. `q_point` or `q_particle`, is the quantity (the number) of points or particles, the user could expect to find near to its declaration the declaration of arrays of these items, named `point` or `particle` respectively. Note that both the name of array and the meaningfull part of the name of its size are in *singular* mode: `point` and `q_point`, not `points` and `q_points`. This is justified by the following considerations. When you address an element of such an array, you write, for example, `point[n]` or `particle[i]`. Having these inscriptions looking like `points[n]` or `particles[i]` would definitely lead to confusion, whether they address and mean a single point and a single particle or many of them allocated in these particular elements. So it is adopted that if the array name in plural, this means that its elements mean plural things themselves, that is they are usually arrays themselves. And much more frequent case of array with single elements is denoted by a single noun. Finally, in order to provide the notation of quantity to be as similar as possible to the notation of array and to be easily identified in the program as it, it is reasonable to leave it in the same singular mode, i.e. `q_point`.

All templated arrays, defined in wcpplib (in the directory `wcpplib/safetl`), use the standard C++ way of indexing, that is the first index is 0. For example, the size of the array `DynLinArr< double > ar` is obtained by `long q = ar.get_qel()` (get quantity of elements), or changed by `ar.put_qel(q)`. Then the allowed indexes are from `0` to `q-1`:
`double first_el = ar[0]`.

The array of this type can be an element of another array, which allows to build many-dimensional arrays with variable sizes (size by the next dimension can be initialized separately for each index by previous dimension and therefore it can be different).

If one wants to define an array with the constant in this sense sizes, that is a rectangular array, one should use another template class called `DynArr`. The number of dimensions of `DynArr` can be varied, but all dimensions are "fixed" along other indexes. The indexes and the sizes of `DynArr` are in general expressed by a variable of type `DynLinArr< long >`, rather than by a single `long`, as for `DynLinArr`. It is therefore not surprising that both in the interface and in the implementation of `DynArr` one can find many appearances of `DynLinArr< long >`. In particular, the normal fast access to elements of `DynArr` is provided by functions `.ac(const DynLinArr<long>& ind)`, rather than by `[][]....` The latter are also provided in the last version, but this is advised to be used only when sintax (of already existed or maximally flexible programs) assumes these brackets `[][]....` Note that in the case of `DynArr` this type of de-indexing works somewhat slowly due to some inavoidable problems in its implementation. For two and three-dimension cases special, more simple and very fast interface is also provided, e.g. `.ac(long i1, long i2, long i3)`. The first indexes are always 0.

In contrast with STL vectors, the both arrays, `DynLinArr` and `DynArr`, do not have "hidden" elements or unused memory reserved after the last elements. If the arrays have to be increased, a new memory block is allocated and the existing elements are copyed by assignment of each element. This is sometimes inconvenient by efficiency and due to loosing pointers addressed the array elements.

The array that can rise without copying content is `BlkArr`, the blocked array. The elements are stored in a sequence of blocks. It represents a trade off between the efficiency of rise, the efficiency of access by indexes, and the system memory expense. It is assumed that at the rise by one element each next block will have larger size than the previous one (twice as large). The user can also initialize the first block with reasonably large for his problem size. This allows to keep the total number of blocks reasonably small, while not occuping too much system memory. But the physical and logical sizes of this array are different values. The logical size is obtained or is put by similar `get_qel()` and `put_qel()` functions. The physical size is determined automatically except the initialization of the first block, which can be initialized by `void BlkArr::allocate_block(long fqel)`. The indexes are counted from zero as well.

At copying all arrays copy all elements by assignment. At deletion all elements are deleted as well.

There is also a list class `AbsList` which works by similar way: copying by assignment and deletion. We don't use interators, but iterate through the lists by a classic manner, like this:

```
class Element
{
  ...
};
ostream& operator<<(ostream& file, const Element& f) {...}
Element el1, el2;
AbsList< Element > a_list;

// filling the list

a_list.append(el1);  // copies el1 to list
a_list.prepend(el2); // copies el2 to list and puts it before el1

AbsListNode< Element >* pointer_to_node = NULL;
while( (pointer_to_node = a_list.get_next_node(pointer_to_node)) != NULL)
{
  // do something with element, for example flush it to stream
  mcout << pointer_to_node->el;
}
```

All arrays and the list can be elements of each other.

9.2 The pointers

The plain pointers are not used in the classes as class members. Otherwise they could create complete mess. When an object needs to refere at another object, I first decide whether that other object belongs to the first.

The notion "to belong", that is the "ownership", is usually coherent with the concept of object integrity, with an exclusive reference (one and only one referee per each refered object), and with the lifetime of refered object not exceeding the lifetime of the referee. The opposite notion, that is the "external" or "independent" object, is opposite in all these senses; it is assotiated with possibly many objects refering to that one and with the lifetime possibly exceeding the lifetime of referees and independent on them.

Frequently one else category of relations is undermined in the so-called "shared" smart pointers. Sometimes the term "smart pointer" is understood just as a notation of this particular and not very consistent category. For the first glance the "shared" smart pointer expresses a shared-ownership relation. Each object can have many pointers to it, or many "owners". The lifetime of such object could exceed the lifetime of a single pointer to it, but cannot exceed the lifetime of all pointers to it. By the opinion of the author this concept is a mixture of different cases and is rather confusing. For example, copying all the pointers to a certain object together to another place, which might naturally express the creation of another full header object with new instances of the owned part, would not actually trigger cloning the part. So the shared pointer does not indicate any form of "ownership", integrity, or independency (the integrity and independency are basic features by which we distinguish objects in our mind). At least all functionality of HEED can be achieved without "shared" pointers and they are not used here.

Of course, the particular case for expression of the owneship is putting the owned object as a class member of the owner. But this is not always possible since in order to do this we have to know precisely the type of the owned object and its size, which is not always the fact.

The problem of the variable size at fixed type is solved with the help of dynamic arrays.

The problem of a single object of given or derived type can be solved with the help of putting pointer in a class as a class member. But then we will have a terrible mess at initialization, access, copying, and deletion of the owner. The experience suggests that such structures can never be encoded and applied without errors. Therefore we use in these cases well debugged and well defended (fool-proved) active pointer or `ActivePtr`, a templated smart pointer.

The opposite case also need some defence and we use for it a passive pointer, `PassivePtr`, see `wcpplib/safetl/AbsPtr.h`.

The both pointers check the existance of the object at each access by `*`, `->`, and `.get()` and trigger an error if it does not exist. There is also function for dereferring with verification `.getver()`, which just returns `NULL`, if the object does not exist.

The active pointer clones the object at the copying the owner. This is implemented with some tricks, which are decribed in `wcpplib/safetl/AbsPtr.h`.

The passive pointer mainly intended to check that the object exists, and it also has many auxiliary functions, see `wcpplib/safetl/AbsPtr.h` as well.

The active pointer can refere only the objects which are separated entities in dynamic memory. The passive pointer can refere an object allocated by any method (including static and automatic objects or their parts). The object pointed by passive pointer can be removed (independantly on pointer), which normally triggers clearing the pointer and detecting error, if access through it is again requested.

The both pointers are "intrusive", since they assume the referred object is prepared to be pointed, that is it has some predefined components. It is unfortunate, but not too inconvenient.

Inside the function bodies the use of plain pointers is allowed. They are usually obtained by dereferring the smart pointers at the beginning of the functions and aimed for acceleration of access. In such cases it is, of course, assumed that the object does not disappear till the function end.

This concept was first described in the appendix of the preprint [1] (although with different notations of the types of the pointers). One reservation prompted by experience should be made. The implementation of the statement from [1]: "The addressed object should know the addresses of all protected (i.e. passive by the new notations) pointers to it and clear them at its deletion" could lead to too large expenses of computer time in the case of very many pointers referring one object. This can be avoided totally by the introduction of an intermediate reference counter object. Then all functionality (more precisely all significant functionality) is preserved after this re-design. This is how it is done now. More complete (but still not full) description of this methodology with a lot of references to related works appeared in SIGPLAN notices [2].

9.3 New and delete

The operator `new` is used in the application programs only at the first initialization of the object right before it is passed for control of active pointer.

The operator `delete` is practically never used (except for destructors of active pointers, arrays and list). All allocated memory is freed automatically with the help of the active pointers, templeted arrays and lists.

9.4 Output streams

Looking in the program a user can wonder what sentences like this mean:

```
ostream & operator << (ostream & file, const AtomMixDef& f)
{
  mfunname("ostream & operator << (ostream & file, const AtomMixDef& f)");
  Ifile<<"AtomMixDef\n";
  indn.n+=2;
  Ifile<<"Z_mean()="<<setw(3)<<f.Z_mean()
       <<" A_mean()/(gram/mole)="<<f.A_mean()/(gram/mole)<<'\n';
  ...
  Iprintn(file, f.qatom());
  long n;
  indn.n+=2;
  for( n=0; n<f.qatom(); n++)
  {
    Ifile<<"n="<<n<<" atom(n)->notation="<<f.atom(n)->notation()<<'\n';
    indn.n+=2;
    Ifile<<" weight_quan(n)="<<f.weight_quan(n)
         <<" weight_mass(n)="<<f.weight_mass(n)<<'\n';
    indn.n-=2;
  }
  indn.n-=2;
  indn.n-=2;
  return file;
}
```

Here this will be explained briefly. But first let us look what in gives in output stream:

```
AtomMixDef
  Z_mean()=12.2857 A_mean()/(gram/mole)=26.8245
  ...
  f.qatom()=3
    n=0 atom(n)->notation=Ar
       weight_quan(n)=0.642857 weight_mass(n)=0.957366
    n=1 atom(n)->notation=C
       weight_quan(n)=0.0714286 weight_mass(n)=0.031983
    n=2 atom(n)->notation=H
       weight_quan(n)=0.285714 weight_mass(n)=0.0106512
continuation would be here
```

First of all we see that there is a very convenient indentation, which distinguishes the beginning of objects and internal arrays or lists. `AtomMixDef` is notation of a class. This notation is already printed with indentation depending on the number of previous indentations. The indentation is provided because of the first "`I`" in front of "`file`": `Ifile<<"AtomMixDef\n";` "`Ifile`" is a marco which is defined in the file `wcpplib/stream/prstream.h`:

```
#define Ifile file<<indn
```

Similar macro is defined for mcout:

```
#define Imcout mcout<<indn
```

The notation `indn` is an object of a special small class, defined in the same file:

```
class indentation
{public:
  int n;     // current number of blanks to print
  int s_not;
  indentation(void){ n=0; }
};

extern indentation indn;

inline ostream& operator<<(ostream& file, indentation& ind)
{
  int n;
  if(ind.s_not==1)
    ind.s_not=0;
  else
    for(n=0; n<ind.n; n++) file<<' ';
  return file;
}
```

Thus, the sencence `file<<indn` means just output of `indn.n` blanks. The sentence `Ifile<<"AtomMixDef\n"` means actually `file<<indn.n<<"AtomMixDef\n"`, and it is not surprising that it provides indentation. The similar effect is provided by `Imcout<<"AtomMixDef\n"`. After printing the class notation one can want to increase the indentation by, for example, 2 blanks in order to express that the following is internals of this class. This is done by command `indn.n+=2` in the listing above.

In the sentence like `Ifile<<"Z_mean()="<<setw(3)<<f.Z_mean()` the notation of the variable `Z_mean()` is repeated. This is rather boring. To make such commands shorter we might define another macro like

```
#define Iprint( file , name ) \
file<<indn<<#name<<"="<<noindent<<name<<yesindent;
```

This macro is actually defined in the same file `wcpplib/stream/prstream.h` The functions

```
ostream& noindent(ostream& f);
ostream& yesindent(ostream& f);
```

are defined in the file `wcpplib/stream/prstream.c`:

```
indentation indn;

ostream& noindent(ostream& f)
{
  indn.s_not=1;
  return f;
}
ostream& yesindent(ostream& f)
```

```
{
  indn.s_not=0;
  return f;
}
```

Now we see, that this construction allows us to print the name of the object with indentation, to print the assignment character `=`, to print the value or a dump of the object *without* indentation of its first line (which otherwise would be just unnecessary blanks between `=` and the following content. The macro `Iprint` does not add the next line '`\n`' character at the end of the last line. It is good for printing the complicated objects, which usually do this themselves. For a single numbers it is not good. Therefore there is another macro `Iprintn`, which does this.

There are a number of other similar macros. For example, to print two numbers by one command in one line, we can use

```
#define Iprint2n( file , name1, name2 ) \
file<<indn<<#name1<<"="<<noindent<<name1 \
<<", "<<#name2<<"="<<noindent<<name2<<'\n'<<yesindent;
```

and so on.

9.5 Run-time error detection

In section 7.6 we already described briefly this package. Let us now consider it in more details.

The purpose of this package is detection of run-time logical errors and reporting about them. The term "run-time logical error" means here not things like segmentation fault but typically "a variable is out of bounds". There may be more complicated conditions with two, three variables etc. We have three main problems here. First, it needs to detect the error in the program by a way that does not reduce strongly its speed. Secondly, it needs to report about it by a way that minimizes the further debug effort. Information should be exhausting both about the place of this error and its essense. Thirdly, it needs to take consistent further actions, typically to terminate the program or to issue exception in the case of significant error, or simply to continue in the case of warning.

As example of importance of these tasks, let us consider the diagnistics recently occured in one of the author's programs:

```
 ***** ERROR in HBOOK2 : YMIN.GE.YMAX : ID=        4
 ***** ERROR in HPAK : Unknown histogram : ID=        4
```

Obviously, these messages are sent by the HBOOK package (to which the author's program is linked). We see that the error is initially detected in something called "HBOOK2". It is then not obvious, what is this. In many cases, such as the use of `C++` interface to the HBOOK package, when one does not deal directly with HBOOK functions, one may not remember that "HBOOK2" (as well as "HPAK") is the name of a subroutine. This nevertheless can be guessed. Then, this function can be called from thousand places in source code, and one does not see from this message from which this particular erroneous call is made. Next, this user-written function can itself be called from many different places and it is also difficult to determine fron which. The remark "`ID=        4` " indicates that this histogram has number 4 (the program was very small), but if this number is generated dinamically (as in the case of the `C++` interface), one have to analyse the program flow to determine the erroneous call. Next, the inscription `YMIN.GE.YMAX` indicates that either `YMIN` is wrong, or `YMAX`, or both, but one cannot determine this from this message. To find an error it needs either to analyse the source code or to add debug output before the call of HBOOK2. Why we do not insert a more detailed output directly to the functions like HBOOK2? Of course, we cannot modify library subroutines suh as HBOOK2, so here we have to deal with it as is, but we can at least inbuild more complete debug utilities in our new programs. Finally, the last line of the lising above indicates that another subroutine from HBOOK has detected, that the histogram was not initialized. None of these errors triggered the program termination, but this can be determined only from the continuation of the listing and from the other results. This is also inconvenient.

Let us now pass to C++. Let us assume that we need to guarantee or just to check that a variable does not exceed some limit. Then the ordinary way to express this in a program would be:

```
#ifdef DO_CHECKS   // \textsf{to bypass unnecessary checks after debug}
if( variable > limit )
{
  mcerr<<"error in function ...\n"
       <<"File is "<<__FILE__<<" , line number is "<<__LINE__<<'\n'
       <<" variable > limit, variable=" << variable << " limit = "<<limit
       <<'\n';
  abort();
}
#endif
```

In practice such a style is not satisfactory by two reasons:

- Such rudimentary checks are too long for printing and distracting at reading the prgram text. The essence of this expression is forbidding the case when “`variable > limit`” , which is not seen instantly in this code. The inbuild `assert()` statement looks shorter and more clear, but it does not print the values of the compared variables. In most of the cases this means that the programmer should either edit the program, include printing in its source, compile and link it, then run it and reproduce the same situation; or he should run the program under debugger and reproduce the same situation as well. If this situation appears after, say, a week of calculations, the situation gets not too funny.
- Printing the function name, the file name, and line number is not sufficient to localize the error in those cases when the function with error is called from a long sequense of other functions and the problem occurs in them.

The problem of the error localization is solved with the help of printing a stack of names of functions currently involved. The function name (or any other name) can be put in the stack by macro

```
mfunname("name to put in stack");
```

This macro is replaced by very simple text:

```
#define mfunnamep(string) static char* FunNameIIII=string;  \
FunNameWatch funnw(FunNameIIII)
```

The name `FunNameIIII` is choosen just as unusual name which can scarcely be met in programs. The initialization of object of type `FunNameWatch` triggers inclusion of the pointer to string into the stack. The destruction of such object triggers removal of the corresponding pointer to string from the stack. This does not take any significant time, since the character array itself is not copied. This does not work however in inlined functions. But we would not use this in very fast inlined functions anyway. For them more quick methods are applied. For many geometrical functions older and faster method is used, see sect. **??**.

So if you write a function

```
void my_function(void)
{
  mfunname("void my_function(void)");
  // at this moment the function name is remembered
  ...
}
```

the program will remember its name while this function is involved. The macro `mfunname` is switched on only if the macro `FUNNAMESTACK` is defined at compilation. The macro `mfunnamep` is permanent.

This stack is printed by macro `stackprt(stream)` which is defined in the file `FunNameStack.h` as

```
#define stackprt(stream)  stream<<FunNameStack::instance()
```

This means that the class `FunNameStack`, which is keeping the stack, is a "singleton" class. It is initialized in a single exemplar at the first call by a static function `instance()`

```
class FunNameStack
{
public:
  static FunNameStack& instance()
  {
    static FunNameStack inst;
    return inst;
  }
  ...
};
```

but it can be copied into another exemplar in order to pass the location of the error through the exception treatment:

```
class FunNameStack
{
public:
  ...
  FunNameStack( const FunNameStack& f);
  FunNameStack& operator=(const FunNameStack& f);
  ...
};
```

This class has two data members which can have it inactive, or make him printing names of functions at entring/exiting:

```
class FunNameStack
{
public:
  ...
  int s_act;          // 1 - active, 0 - not active
                      //( to switch without recompilation)
public:
  int s_print;        // works only is s_act=1;
                      // 0 - no print
                      // 1 - at each entry print the name of current function
                      // 2 - at each entry and each exit
                      //    print the name of current function
                      // 3 - at each entry print all stack
                      // 4 - at each entry and exit print all stack
  void set_parameters(int fs_act=1, int fs_print=0);
};
```

If user want to change the default parameters before invoking main, he can initialize a global object with any name of a special class:

```
class FunNameStack_Assist
{
public:
  FunNameStack_Assist(int fs_act=1, int fs_print=0)
```

```
  {
    FunNameStack::instance().set_parameters(fs_act=1, fs_print);
  }
};
```

With this system the problem of brief recording of error condition is solved like this:,

```
check_econd12( variable , > , limit , mcerr);
```

This means "checking error condition", the first number after "`econd`" means that there is one condition, and the second number means that both variables have to be printed in the case of error. This macro is defined in the file `FunNameStack.h`, and looks like this:

```
#define check_econd12(a, sign, b, stream) if(a  sign  b) \
{ funnw.ehdr(stream); stream<<'\n'<<#a<<#sign<<#b<<'\n'; \
stream<<#a<<'='<<(a)<<' '<<#b<<'='<<(b)<<'\n'; spexit(stream); }
```

The call `funnw.ehdr(stream)` prints "error notification" and the name of the current function. The macro `spexit(stream)` looks like

```
#define spexit(stream) {stackprt(stream); \
stream<<"File is "<<__FILE__<<" , line number is "<<__LINE__<<'\n'; \
spexit_action(stream); }
```

We see here `stackprt(stream)`, which outputs the stack of the functions currently involved. Then this macro further localizes the error by printing the file name and the line number. But both are available only of the program is compiled with `-g` option. Finnally, the function `spexit_action`, called at the end of `spexit`, is a very simple function, whose body is defined in the file `FunNameStack.c`:

```
void spexit_action(ostream& file)
{
  file<<"spexit_action: the streams will be now flushed\n";
  file.flush(); mcout.flush(); mcerr.flush();
  if(s_throw_exception_in_spexit != 1)
  {
    file<<"spexit_action: the abort function is called\n";
    abort();
  }
  else
  {
    file<<"spexit_action: an exception is now called\n";
    throw ExcFromSpexit();
  }
}
```

With these tools the error condition "`i < 0`" would be written in a program by

```
check_econd12( i , < , 0 , mcerr);
```

It is not possible to avoid these commas. In the case of `i=-1`, the program should write

```
myfun10(int i): ERROR:

i<0
i=-1 0=0
FunNameStack: s_init=1 qname=3
  0  void main(void)
  1  myfun3(int i)
  2  myfun10(int i)
File is test.c , line number is 39
spexit_action: the streams will be now flushed
spexit_action: the abort function is called
Abort (core dumped)
```

This looks nice except of the sentence "`0=0`", which is rather strange (although perfectly correct). Such things would be distracting in output listing, and experience suggests that it is reasonable to avoid them. Unfortunately, it is not easy to do in standard C++. The only feasible solution is to multiply the `check_econd` macros. For given case it needs to have macro

```
#define check_econd11(a, signb, stream) if(a  signb) \
{ stream<<"ERROR:\n"; stream<<'\n'<<#a<<#signb<<'\n'; \
stream<<#a<<'='<<(a)<<'\n'; spexit(stream); }
```

with one condition and one value to print. Its call will be written like this:

```
check_econd11( i , <  0 , mcerr);
```

and output will be

```
myfun10(int i): ERROR:

i<0
i=-1
FunNameStack: s_init=1 qname=3
  0  void main(void)
  1  myfun3(int i)
  2  myfun10(int i)
File is test.c , line number is 39
spexit_action: the streams will be now flushed
spexit_action: the abort function is called
Abort (core dumped)
```

The condition `i < 0 || i > 10` would be written as

```
check_econd21( i , <  0 || , > 10 , mcerr);
```

The condition `i < j1 || i > j2` (which requires printing `i`, `j1`, `j2`) would be written as

```
check_econd23( i , < , j1 , || ,  > j2 , mcerr);
```

The condition `i1 < j1 || i2 > j2` (which requires printing `i1`, `i2`, `j1`, `j2`) would be written as

```
check_econd23( i1 , < , j1 , || , i2 , > j2 , mcerr);
```

and so on. There are some more similar macros defined in the file `FunNameStack.h`. But there is no cover for all cases. In a generic case one can use generic macro

```
#define check_econd(condition, add, stream) if(condition) \
{ funnw.ehdr(stream); stream<<'\n'<<#condition<<'\n'; \
stream<<add; spexit(stream); }
```

which checks any condition and prints any string (`add`). The last condition would look like this:

```
check_econd( i1 < j1 || i2 > j2 ,
             "i1="<<i1<<" i2="<<i2<<" j1="<<j1<<" j2="<<j2 ,
             mcerr);
```

The classes and macros supporting these actions are described in subdirectory `wcpplib/util` in files `FunNameStack.*`, where `*` stands for `c`, `ic`, and `h`.

References

[1] I.B. Smirnov, LikelihoodLib — Fitting, Function Maximization, and Numerical Analysis. Preprint PNPI-2421, 2001.

[2] I. B. Smirnov, Raw Pointers in Application Classes of C++ Considered Harmful. ACM SIGPLAN Notices, Vol 42, No. 4, April 2007.