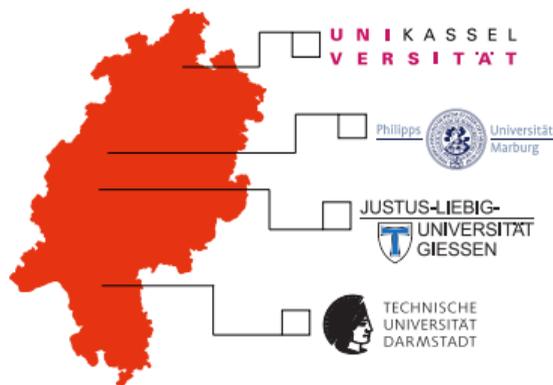


# Debugging & Totalview

Hessisches Kompetenzzentrum für Hochleistungsrechnen (HKHLR)

René Sitt / Tim Jammer

ProtHPC 2022-Q4: 2022.11.04



HKHLR is funded by the Hessian Ministry of Sciences and Arts



# In This Course

- ▶ General overview about debugging techniques and some common bug causes
- ▶ Overview about TotalView
- ▶ Usage of TotalView to debug *parallel* applications
- ▶ Technical background: Memory layout & Name mangling
- ▶ Live Demos



## "Example 1"

```
1 void foo () {
2     int * myPointer;
3
4     int i;
5
6     for (i=0;i<100;i++) {
7         myPointer[i]=i*i;
8     }
9
10    return myPointer;
11 }
```

## "Fix for Example 1"

```
1 void foo () {
2     int * myPointer =
3         (int*) malloc(sizeof(int)*100);
4     int i;
5
6     for (i=0;i<100;i++) {
7         myPointer[i]=i*i;
8     }
9
10    return myPointer;
11 };
```

FIX: Add a properly casted `malloc` to allocate sufficient memory!



## "Example 2"

```
1 void doScience () {
2     int * myPointer =
3     (int*) malloc(sizeof(int)*100);
4
5     ... // Do some work here
6
7
8     return;
9 }
10 int main(){
11     doScience();
12
13     ... // more work here
14 }
```

## "Fix for Example 2"

```
1 void doScience () {
2     int * myPointer =
3     (int*) malloc(sizeof(int)*100);
4
5     ... // Do some work here
6
7     free(myPointer);
8     return;
9 }
10 int main(){
11     doScience();
12
13     ... // more work here
14 }
```

FIX: Release memory using `free`!

## "Example 3"

```

1 int getPowersOf2_64Bit() {
2     int po2[64];
3
4     po2[0]=1;
5     for (int i=1;i<64;i++)
6         po2[i]=po2[i-1]*2;
7     return po2;
8 }
9 int main(){
10    int powersOf2[64];
11    powersOf2 = getPowersOf2_64Bit();
12
13    ... // more work here
14
15
16    return 0;
17 }

```

## "Fix for Example 3"

```

1 int getPowersOf2_64Bit() {
2     int po2 =
3         (int*) malloc(sizeof(int)*64);
4     po2[0]=1;
5     for (int i=1;i<64;i++)
6         po2[i]=po2[i-1]*2;
7     return po2;
8 }
9 int main(){
10    int * powersOf2;
11    powersOf2 = getPowersOf2_64Bit();
12
13    ... // more work here
14
15    free(powersOf2);
16    return 0;
17 }

```

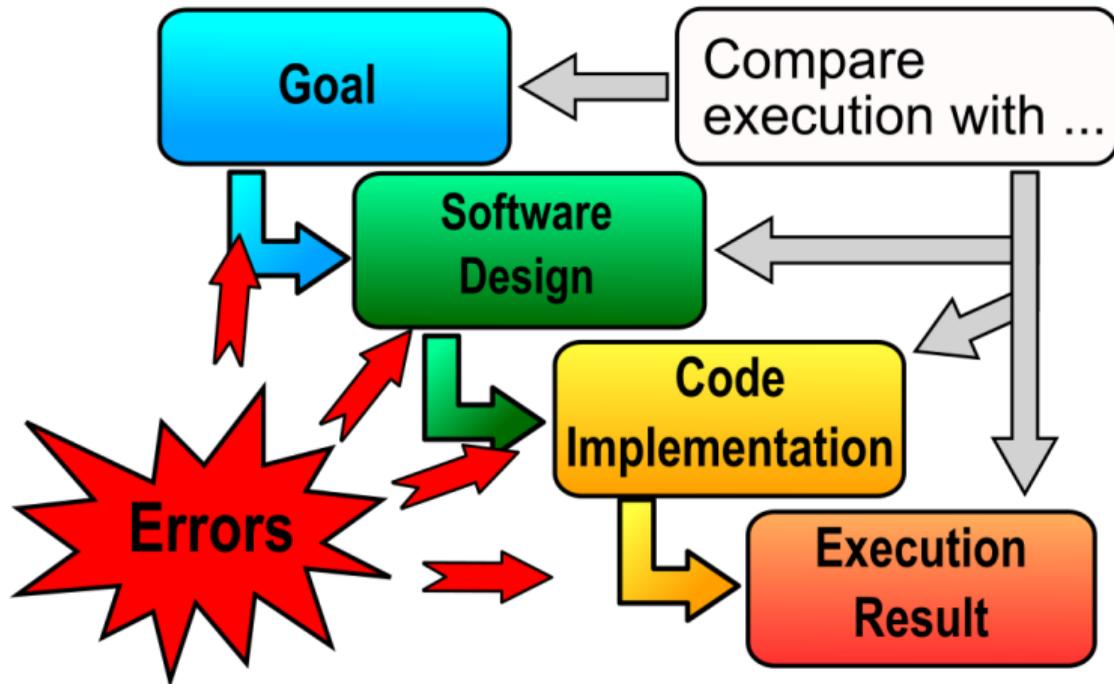
**FIX:** Use `malloc` to allocate "heap" memory, do not return local memory.



- ▶ **Debugging** is the process of finding and resolving **bugs**
- ▶ A **bug** is a defect or problem that prevents the correct operation, results in unintended behaviour, or generates the wrong output
- ▶ Debugging strategies may involve
  - ▶ Interactive methods, incl. monitoring application, or system
  - ▶ Analytical methods, incl. control-flow analysis, log-file analysis, analysis of memory dumps, profiling and tracing,
  - ▶ Testing methods: unit testing, integration testing

Src.: [wikipedia.org/wiki/Debugging](https://wikipedia.org/wiki/Debugging)

Debug observations against all levels; check intent vs observation:



Totalview is a comprehensive debugging solution for demanding parallel and multi-core applications:

- ▶ **Interactive debugging,**
- ▶ **Analyzing core-dumps** and
- ▶ **live program inspection.**

Key features:

- ▶ Wide compiler & platform support: C/C++, Fortran, UPC, Assembly and Python.
- ▶ Integrated Memory Debugging
- ▶ **Reverse Debugging**
- ▶ Concurrency & HPC debugging support: **MPI** and **OpenMP**



- ▶ Prepare and load minimal Totalview environment:

shell

```
>$ module load totalview
```

- ▶ Totalview uses X-Windows to display its UI. A guide for the X-environment is available at [HPC-Wiki.info](https://wiki.hpc-wiki.info)→Linux in HPC→SSH Graphics and File Transfer.
- ▶ Two user-interfaces:

The new UI, new features continuously added.

sell

```
>$ totalview -newUI
```

Classic UI, feature complete, to be phased out.

shell

```
>$ totalview -classicUI
```

We will discuss Totalview using the program found in the **demo01** folder: It recursively computes factorials. There is a bug regarding the ordering of the programs output.

- ▶ INPUT.txt                      Some example inputs
- ▶ Makefile                        Maketargets
- ▶ demo01.cc                      Original sourcefile
- ▶ demo01B.cc                    Intermediate solution

The makefile has four targets: *demo01.exe*, *demo01A.exe*, *demo01B.exe* and *clean*.  
The program accepts input via STDIN or as program arguments.  
Please consult the readme.md for more details.

shell

```
>$ ./demo01.exe 1 3 5 7
```

## Two requirements:

- 1 Debug markers must be present in binary  
Use compiler option **-g** for every source-file to be debugged
- 2 Source-code must be present at the original location  
debug information stores the line-numbers and path to the source-file in the executable.

```

Contents of the debug_info section:
Compilation Unit @ offset 0x0:
Length:          0x0045 (32-bit)
Version:         4
Abbrev Offset:   0x0
Pointer Size:    8
-@-@-@- Abbrev Number: 128 [DW_TAG_compile_unit]
  <- DW_AT_producer : [indirect string, offset: 0x0d1]: GNU C++ [4.8.4] 20090928 (Red Hat 4.8.4-11) -bunugeneric_marchx86_64.g
  <- DW_AT_language : 4 (C++)
  <- DW_AT_name : [indirect string, offset: 0x0ad1]: demo01.cc
  <- DW_AT_comp_dir : [indirect string, offset: 0x1d13]: /home/HKHLR/Kurs/totalviewcourse/demo/demo01
  <- DW_AT_namespace : 0x0
  <- DW_AT_low_pc : 0x0
  <- DW_AT_high_pc : 0x0
  <- DW_AT_start_line : 0x0
  <- DW_AT_start_column : 0x0
-@-@-@- Abbrev Number: 127 [DW_TAG_namespace]
  <- DW_AT_name : 0x0
  <- DW_AT_decl_file : 0x0
  <- DW_AT_decl_line : 0
  <- DW_AT_sibling : @00000
-@-@-@- Abbrev Number: 187 [DW_TAG_namespace]
  <- DW_AT_name : [indirect string, offset: 0x1f0c]: __cxx11
  <- DW_AT_decl_file : 0x0
  <- DW_AT_decl_line : 0x0
  <- DW_AT_decl_column : 0x0
  <- DW_AT_export_symbols : 1
  <- DW_AT_sibling : @00000
-@-@-@- Abbrev Number: 41 [DW_TAG_class_type]
  <- DW_AT_name : [indirect string, offset: 0x0a2d]: basic_stringchar, std::char_traits<char>, std::allocator<char>
  <- DW_AT_byte_size : 22
  <- DW_AT_decl_file : 2
  <- DW_AT_decl_line : 77
  <- DW_AT_decl_column : 11
  <- DW_AT_sibling : @00000
-@-@-@- Abbrev Number: 18 [DW_TAG_structure_type]
  <- DW_AT_name : [indirect string, offset: 0xb33f]: _Alloc_hider
  <- DW_AT_byte_size : 4
  <- DW_AT_decl_file : 2
  <- DW_AT_decl_line : 139
  <- DW_AT_decl_column : 14
  <- DW_AT_sibling : @00000
-@-@-@- Abbrev Number: 50 [DW_TAG_structure_type]
  <- DW_AT_name : [indirect string, offset: 0xb33f]: _Alloc_hider
  <- DW_AT_data_member_location : 0
  <- DW_AT_external : 1
  <- DW_AT_decl_file : 2
  <- DW_AT_decl_line : 2

```

## Shell

```
>$ checkDebugInfo.sh <path to binary>
```

- ▶ Helper tool by HKHLR.
- ▶ Available in "course material → scripts" directory

Example for missing debug infos:

```
[kurs64701@logc0004 demo01]$ checkDebugInfo.sh demo01.exe  
No Debuginformation found in demo01.exe
```

Example for available debug infos:

```
[kurs64701@logc0004 demo01]$ checkDebugInfo.sh demo01A.exe  
"demo01A.exe" contains debug information.  
The following source files were compiled with -g:  
/home/kurse/kurs00047/kurs64701/totalviewcourse/demos/demo01/demo01A.cc
```

We recommend to always use the following compiler flags when debugging:

- ▶ `-g` to get debug information and see the source code
- ▶ `-O0` to *disable* compiler optimization such as code motion, so that one can actually follow along the code when it is executed in the debugger



- ▶ Function-, method, and subroutine names referred as symbols in compilation,
- ▶ For linking (and for dynamic libraries) each used symbol, be it function, method or subroutine, a matching implementation is required.
- ▶ Command line tool `nm` exploring function symbols in a binary or shared library.
- ▶ Different languages and compilers transform symbols during "compilation".
  - ▶ C++ compilers (GNU, Intel and other) heavily transform most symbols,
  - ▶ GCC compatible compilers canonize Fortran symbols.



## C++ code example

```

1 #include <iostream>
2 using namespace std;
3 class myClass {
4     public:
5     double toDouble(int k){
6         return (double)k;
7     }
8 };
9 void aFunction(int arg){
10     cout << "Output" << arg
11         << endl;
12 }
13 int main(int c, char ** v){
14     aFunction(10);
15     return 0;
16 }

```

## Symbol-list excerpt

```

1 _GLOBAL__sub_I__Z9aFunctioni
2 _Z41__static_initialization_and_destruction_0ii
3 _Z9aFunctioni
4 _ZN7myClass8toDoubleEi
5 _ZNSolsEPFRSoS_E
6 _ZNSolsEi
7 _ZNSt8ios_base4InitC1Ev
8 _ZNSt8ios_base4InitD1Ev
9 _ZSt4cout
10 _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
11 _ZStL19piecewise_construct
12 _ZStL8__ioinit
13 _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
14 __cxa_atexit
15 __dso_handle
16 main

```

The symbol `main` is not transformed; called by program loader.



## Demangling mangled C++ symbols:

- ▶ `"_Z9aFunctioni"`:
  - ▶ `_Z`: A function or method,
  - ▶ `9aFunction`: Length of identifier and identifier, and
  - ▶ `i`: One integer argument.
- ▶ `"_ZN4demo10demoMethodEif"`:
  - ▶ `_Z`: A function or method,
  - ▶ `N .. E`: Begin (`N`) and end (`E`) of fully qualified identifier,
  - ▶ `4demo`: Length and class name, namespace or identifier (`demo`),
  - ▶ `10demoMethod`: Length and class name, namespace or identifier (`demoMethod`), and
  - ▶ `if`: Two arguments: an integer and a float.

## Command line tool `c++filt` decodes mangled symbols:

```
c++filt _ZN4demo10demoMethodEif → demo::demoMethod(int, float).
```



## "Fortran code example"

```
1 module arithmetic
2   public :: add
3   contains
4   subroutine add( a, b, result )
5       integer, intent( in ) :: a, b
6       integer, intent( out ) :: result
7       result = a + b
8   end subroutine add
9 end module arithmetic
10 subroutine aFunction(n)
11   integer, intent(in) :: n
12   print *, 'Some Output: ', n
13 end subroutine aFunction
14 program demo
15   call aFunction(10)
16 end program demo
```

## "Symbol excerpt"

```
1 MAIN__
2 __arithmetic_MOD_add
3 _gfortran_set_args
4 _gfortran_set_options
5 _gfortran_st_write
6 _gfortran_st_write_done
7 _gfortran_transfer_character_write
8 _gfortran_transfer_integer_write
9 afunction_
10 main
11 options.1.3785
```

## "C code example"

```
1 #include <stdio.h>
2 void aFunction(int argument1)
3 {
4     printf("Output %i\n",\
5         argument1);
6 }
7 int main(int argc, char ** argv)
8 {
9     aFunction(10);
10    return 0;
11 }
```

## "Symbol excerpt"

```
1 __libc_start_main@@GLIBC_2.2.5
2 _fini
3 _init
4 _start
5 aFunction
6 main
7 printf@@GLIBC_2.2.5
```

## shell

nm demo.exe

Output in 3 columns:

- 1 Relative address.
- 2 Symbol status:  
 u: undefined symbol;  
 typically a function  
 from shared library.  
 T or t: Defined symbol;  
 implementation in file.
- 3 Symbol: Symbol  
 (mangled) as stored in  
 the file.

## "Symbol excerpt"

```

1 0...4008e1 t _Z41__static_initialization_and_destruction_0ii
2 0...400876 T _Z9aFunctioni
3 0...400934 W _ZN7myClass8toDoubleEi
4           U _ZNSt8ios_base4InitC1Ev@@GLIBCXX_3.4
5           U _ZNSt8ios_base4InitD1Ev@@GLIBCXX_3.4
6 0...601060 B _ZSt4cout@@GLIBCXX_3.4
7 0...4009e8 r _ZStL19piecewise_construct
8 0...601171 b _ZStL8__ioinit
9 0...400790 T _start
10 0...601170 b completed.7295
11 0...601050 W data_start
12 0...4007d0 t deregister_tm_clones
13 0...400870 t frame_dummy
14 0...4008b0 T main

```

## shell

```
nm demo.exe | c++filt
```

## "Symbol excerpt"

```
1 00000000004008e1 t __static_initialization_and_destruction_0(int, int)
2 0000000000400876 T aFunction(int)
3 0000000000400934 W MyClass::toDouble(int)
4          U std::ios_base::Init::Init()@@GLIBCXX_3.4
5          U std::ios_base::Init::~~Init()@@GLIBCXX_3.4
6 0000000000601060 B std::cout@@GLIBCXX_3.400000000004009e8 r std::piecewise_construct
7 0000000000601171 b std::_ioinit0000000000400790 T _start
8 0000000000601170 b completed.7295
9 0000000000601050 W data_start
10 00000000004007d0 t deregister_tm_clones
11 0000000000400870 t frame_dummy
12 00000000004008b0 T main
```

In Demo 1 we covered:

- ▶ Loading Totalview and starting a debugging session
- ▶ Compiler Flags for debugging
- ▶ View the call stack in Totalview
- ▶ Redirect program Input
- ▶ Stepping:
  - ▶ go <**g**>, halt <**h**>, kill <**Ctrl-z**>, restart
  - ▶ next <**n**> - go to the next instruction in the current function
  - ▶ step <**s**> - go to the next instruction, following the control flow into called functions
  - ▶ out <**o**> - go to the next instruction in the calling function (out of current function)
  - ▶ run to <**r**> - run up to the point where the cursor is
- ▶ Setting breakpoints



We will discuss more features of Totalview using the program found in the **demo02** folder: Demo 2 implements a two-way connected list with a payload data field.

- ▶ Makefile
- ▶ demo02.cc
- ▶ demo02B.cc - demo02F.cc
- ▶ readme.md

Original sourcefile for *Part III*  
Incremental fixes.

The makefile has 7 targets: *demo02.exe*, *demo02B.exe* to *demo02F.exe* and *clean*.  
The program has no input.  
Please consult *readme.md* for more details.

shell

```
>$ ./demo02.exe
```

Causes for abnormal program termination, next to *exit* or *abort*:

- ▶ Unhandled system signal, e.g. no signalhandler,
- ▶ Unhandled C++ exception and
- ▶ External signal trigger, e.g. kill.

Noteworthy signals:

- ▶ **SIGSEGV - Segmentation fault:** the program accessed an uninitialized memory page, tried to execute non-executable labeled memory addresses, tried to write to read-only memory, attempted to access read-protected memory (kernel space).
- ▶ **SIGILL - Illegal Instruction fault:** the program used an assembly instruction, not supported by the CPU-architecture, or not allowed for the user.
- ▶ **SIGFPE - Floating Point exception:** the program performed an illegal floatingpoint OR integer operation, e.g. division by zero.

More <https://pubs.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html>



- ▶ Contains memory content of the program at the time of the abnormal abort:
  - ▶ Last executed statement,
  - ▶ Stack,
  - ▶ Heap, and
  - ▶ CPU-registers.
- ▶ Can be large  $>$ size of the system's memory
- ▶ Governed by OS policy:

shell

```
ulimit -c $CORESIZE
```

- ▶ Core-dumps are controlled by system configuration:

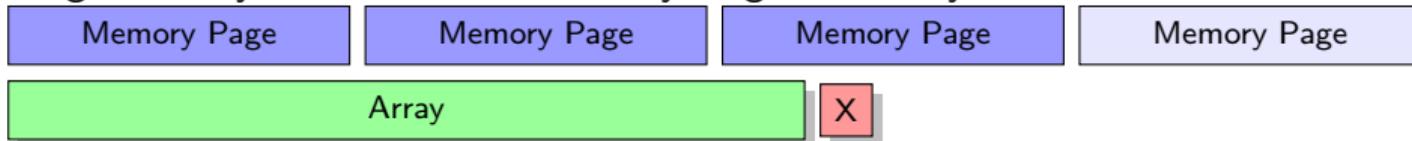
### configuration file

```
/proc/sys/kernel/core_pattern
```

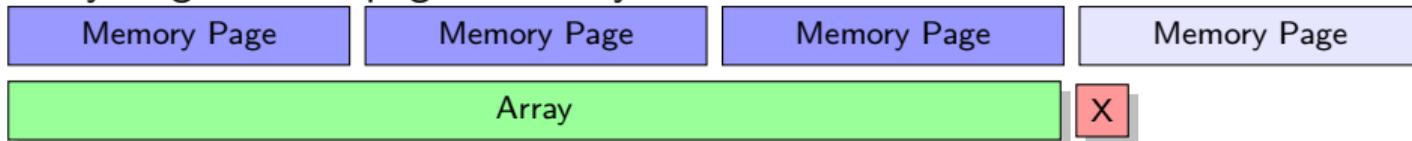
- ▶ Core-dump file configuration: `core.%e.%p.%h.%t`  
Direct creation of core-dump file.  
Example configuration components: `%p`: process id, `%h` hostname, `%e` executable filename, ...
- ▶ Program redirection: `|/usr/lib/systemd/systemd-coredump %P %u %g`  
Note the "pipe" redirection to the system-coredump application. Redirection of the core to a program. System can be configured independently of the kernel.  
`%P`: initial PID of namespace, `%u` user ID, `%g` group ID, ...



Original array ends inside of Memory Page boundary:



Array Aligned with page boundary:



In this Demo we covered:

- ▶ Core-file, ulimits & core-file debugging,
- ▶ local variable inspection,
- ▶ data inspection by the data-view window, and
- ▶ data transformations.



We will discuss more features of Totalview using the program found in the **demo03** folder: Demo 3 implements a loop nest with an out of bound array write.

- ▶ Makefile
- ▶ demo03.c
- ▶ readme.md

Original sourcefile for *Part IV*

The makefile has 2 targets: *demo03.exe* and *clean*.

The program has no input.

Please consult readme.md for more details.

shell

```
>$ ./demo03.exe
```

## Two categories of variables:

### ▶ Direct access:

▶ `double i=pi;`

▶ `int j=1;`

### ▶ Indirect access:

▶ `double a[10];`

`a[j]=i;`

▶ `long int * list;`

`list=(long int*) malloc(sizeof(long int)*100);`

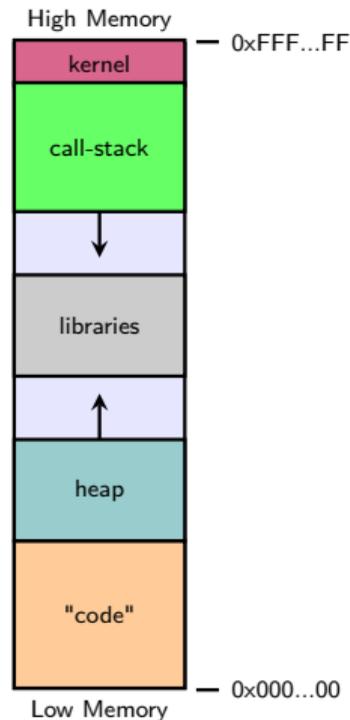
## Out of bound access:

▶ `a[11]=0;`

▶ `list[-1]=0;`

The virtual memory of any process is segmented with a structured defined by conventions:

- ▶ Call-stack,
- ▶ Heap,
- ▶ Static variables, constants, and code (called text-segment),
- ▶ Space for dynamic libraries,
- ▶ Kernel space,
- ▶ Other "reserved" memory as required.

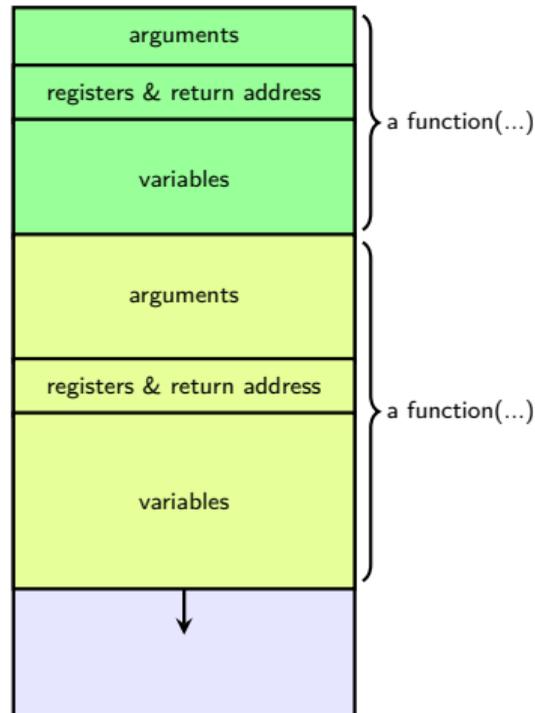


Stack-frame contents:

- ▶ Function arguments in reverse order,
- ▶ Return address,
- ▶ Saved registers, and
- ▶ Local variables.

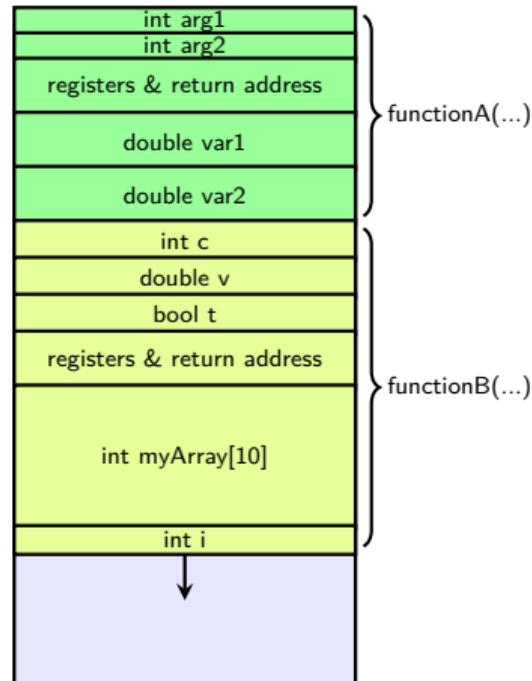
Local variables and fields can be:

- ▶ Rearranged,
- ▶ Padded,
- ▶ and aligned.



## "Code example"

```
1 functionA(int arg1, int arg2){  
2     double var1, var2;  
3     functionB(var1,10.0,true);  
4 }  
5  
6 functionB(int c, double v, bool t){  
7     int myArray[10];  
8     for (int i=0;i<15;++)  
9         myArray[i]=0;  
10 }
```



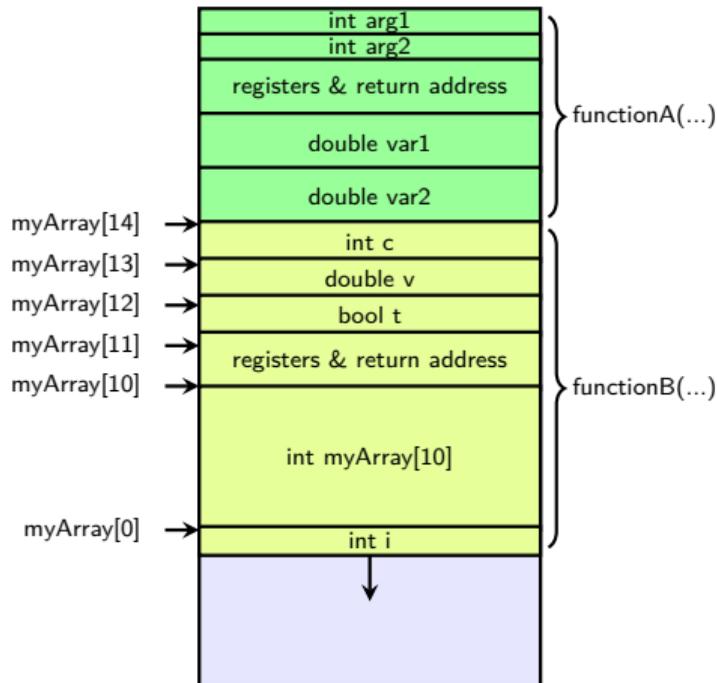
## "Code example"

```

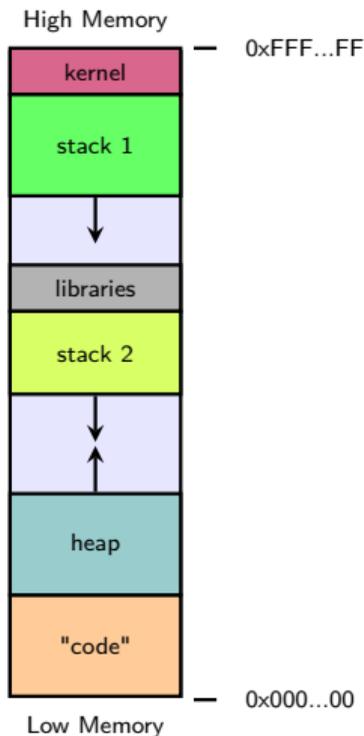
1 functionA(int arg1, int arg2){
2   double var1, var2;
3   functionB(var1,10.0,true);
4 }
5
6 functionB(int c, double v, bool t){
7   int myArray[10];
8   for (int i=0;i<15;++i)
9     myArray[i]=0;
10 }

```

The out of bounds access of `myArray[]` writes to return address and saved registers.



- ▶ Multi-threading and OpenMP require a call-stack for each thread,
- ▶ Additional call-stacks usually located next to mapped memory, such as libraries,
- ▶ PThreads have a default stack size of 2 MB on 64bit systems,
- ▶ PThread stack size can be modified by `RLIMIT_STACK` or `ulimit -s`,
- ▶ Default OpenMP stack size is 4 MB (2 MB) on 64bit (32bit) systems,
- ▶ OpenMP thread stack size is specified by `OMP_STACKSIZE`.
- ▶ In **Multi-threading** there are multiple stacks.



In this Demo we covered:

- ▶ Attaching & releasing processes,
- ▶ Action-points:
  - ▶ Break-points,
  - ▶ Watch-points and
  - ▶ Evaluation-points,
- ▶ Reverse debugging.



This video will be discussing Totalview using the program found in the **demo04** folder. Demo 4 uses basic OpenMP constructs. This demonstrator is without errors used to showcase parallel debugging techniques.

- ▶ Makefile
- ▶ demo04.cc
- ▶ readme.md

Original sourcefile for *Part V*

The makefile has 5 targets: *demo04.exe* and *clean*.

The program has no input.

Please consult readme.md for more details.

shell

```
>$ ./demo04.exe
```

In this Demo we covered:

- ▶ Classic userinterface,
- ▶ Basic parallel debugging techniques,
- ▶ Controlling individual threads and group of threads,
- ▶ Cross thread / process data inspection, and
- ▶ OpenMP debugging.



We will discuss more features of Totalview using the program found in the **demo05** folder:  
Demo 5 is a MPI-Parallelized Program.

- ▶ Makefile
- ▶ demo05.cc

Original sourcefile

The makefile has 2 targets: *demo05.exe* and *clean*. You need to load an MPI module for using this program (e.g. `module load openmpi`)

The program has no input.

shell

```
>$ mpirun -n 2 ./demo05.exe
```

Short explanation of the used MPI Functions in the example code:

- ▶ `MPI_Init`, `MPI_Finalize` initialize/finalize the MPI Library.
- ▶ `MPI_Send(buffer, count, datatype, destination, tag, communicator, status)`: Sends a message of *count* entries fo type *dataype* from *buffer* to the *destination* process using the given message *tag* and *communicator*.
- ▶ `MPI_Recv(buffer, count, datatype, source, tag, communicator, status)`: Wait for a message from the *source* process with the given message *tag* and receive it into the *buffer*.
- ▶ `MPI_Comm_Rank(communicator, rank)`: tells how many processes (*size*) are in the given in the current *communicator*
- ▶ `MPI_Comm_size(communicator, rank)`: tells the unique number of the calling process (*rank*) in the given in the current *communicator*

In this demo we covered:

- ▶ Additional basic parallel debugging techniques,
- ▶ Controlling individual processes and group of processes,
- ▶ Cross process data inspection, and
- ▶ MPI debugging.

