Debugging C
Monash/NCI Training Week

Stephen Sanderson

National Computational Infrastructure, Australia

Acknowledgement of Country

The National Computational Infrastructure acknowledges, celebrates and pays our respects to the Ngunnawal and Ngambri people of the Canberra region and to all First Nations Australians on whose traditional lands we meet and work, and whose cultures are among the oldest continuing cultures in human history.

Outline

Think of debugging like a scientific investigation:

- Identify a problem
- Form a hypothesis about the cause
- **Use appropriate tools to test hypothesis**
- Make adjustments based on data
- Repeat

Outline

Think of debugging like a scientific investigation:

- Identify a problem
- Form a hypothesis about the cause
- **Use appropriate tools to test hypothesis**
- Make adjustments based on data
- Repeat

This session builds on from yesterday's `List` code, and is designed to give an overview of common issues to look for and debugging tools to find them.

Outline

We will cover:

- Common bugs to watch for
- Basic debugging with `gdb`
- Checking for memory bugs with `valgrind`
- Debugging with Arm Forge.

Common bugs

Memory bugs

- Reading from uninitialised memory (i.e. reading from a memory location which hasn't been written to yet). Can consider using `calloc()` instead of `malloc()` to zero-initialise a block of memory when it's first allocated.

Memory bugs

- Reading from uninitialised memory (i.e. reading from a memory location which hasn't been written to yet). Can consider using `calloc()` instead of `malloc()` to zero-initialise a block of memory when it's first allocated.
- Reading from or writing to unallocated memory (can cause a segfault, but not always!). This can often be caused by an incorrect array size, for example, changing the type of `arr` in the code below from **`float`** to **`double`** but forgetting to also change the allocation:

```c
double* arr; // This used to be float* arr;
// ... some code in between
arr = malloc(sizeof(float) * N); // This now only allocates
                                 // half the needed memory!
```

Memory bugs

- Reading from uninitialised memory (i.e. reading from a memory location which hasn't been written to yet). Can consider using `calloc()` instead of `malloc()` to zero-initialise a block of memory when it's first allocated.
- Reading from or writing to unallocated memory (can cause a segfault, but not always!). This can often be caused by an incorrect array size, for example, changing the type of `arr` in the code below from `float` to `double` but forgetting to also change the allocation:

```
double* arr; // This used to be float* arr;
// ... some code in between
arr = malloc(sizeof(float) * N); // This now only allocates
                                 // half the needed memory!
```

This can be avoided by preferring the syntax:

```
arr = malloc(sizeof(*arr) * N);
```

Memory bugs

- Sometimes memory bugs can be difficult to detect, for example a missing null terminator (`'\0'`) at the end of a C string can cause all sorts of issues:

```c
char a[] = "hello"; // This is length 6 with '\0'
char a_cpy[5];

// This just copies ['h','e','l','l','o'] without the '\0'
memcpy(a_cpy, a, 5);

// This will read past the end of a_cpy until it finds a '\0'
printf("%s\n", a_cpy); // Potential memory bug!
```

Memory bugs

- Pointer aliasing can also cause issues. Two pointers alias if they point to the same block of memory. Some functions (e.g. `memcpy`) require that this is not the case, either for soundness or performance. They specify this requirement by marking pointers as **`restrict`** (applies to the left like **`const`**. Breaking this promise to the compiler can cause undefined behaviour!

```c
// Trying to insert an element in a list
void list_insert(List* list, size_t idx, double val) {
    assert(idx <= list->len); // Make sure idx is a valid index
    list->data = realloc(list->data, list->len+1);
    if (idx < list->len) {
        // This is a memory bug!
        // Input pointers to memcpy are marked restrict (from C99),
        // but here we're passing overlapping memory blocks.
        memcpy(&list->data[idx+1], &list->data[idx], list->len - idx);
    }
    list->data[idx] = val;
    list->len++;
}
```

Logic bugs

- A logic bug occurs when a program makes an incorrect decision. Some common causes:

Logic bugs

- A logic bug occurs when a program makes an incorrect decision. Some common causes:
  - Operator precedence – particularly when mixing boolean and bit-wise operators, be careful to make sure operations occur in the order you expect. The easiest way to ensure this is to use brackets. (e.g. `(a << b) == (c & d)`)

Logic bugs

- A logic bug occurs when a program makes an incorrect decision. Some common causes:
  - Operator precedence – particularly when mixing boolean and bit-wise operators, be careful to make sure operations occur in the order you expect. The easiest way to ensure this is to use brackets. (e.g. `(a << b) == (c & d)`)
  - Broken invariants – if your logic code depends on certain properties, make sure these are well documented. Ideally, you should also try to include some form of explicit error if the invariants are broken. The `assert()` macro in `assert.h` is useful for this, and compiles away to nothing in release builds with `-DNDEBUG`.

Logic bugs

- For functions which process tricky logic (and many other classes of functions as well) consider writing unit tests which check for expected behaviour. This can be a separate compile target which just runs through a set of tests. For more advanced unit testing, frameworks like `ctest` can be useful, particularly in combination with `cmake`.

```c
#include <assert.h>
// ...
void fibonacci(int N, int* fib_numbers); /* implemented elsewhere */
void test_fibonacci(void) {
    cont int N = 10;   int fib_result[N];
    memset(fib_result, 0, sizeof(int) * N);

    // Using N-1 to make sure it stops at the correct number
    fibonacci(N-1, fib_result);
    int answer[] = [1, 1, 2, 3, 5, 8, 13, 21, 34, 0];
    for (int i = 0; i < N; ++i) // Exit with an error if wrong answer
        assert(fib_result[i] == answer[i]);
}
```

Missed error reporting

- Many C standard library functions (and functions in some other libraries) indicate an error by returning some form of error code. For example:

Missed error reporting

- Many C standard library functions (and functions in some other libraries) indicate an error by returning some form of error code. For example:
  - `malloc()`, `calloc()` and `realloc()` return a `NULL` pointer if the allocation failed. Not checking for and handling this case can result in code trying to dereference a `NULL` pointer, which is a memory bug.

Missed error reporting

- Many C standard library functions (and functions in some other libraries) indicate an error by returning some form of error code. For example:
  - `malloc()`, `calloc()` and `realloc()` return a `NULL` pointer if the allocation failed. Not checking for and handling this case can result in code trying to dereference a `NULL` pointer, which is a memory bug.
  - Some functions indicate an error by setting the special macro `errno` to a non-zero value. For example, math functions can set `errno` if an invalid argument is passed (like `acos(2)`) or if a divide by zero occurs. Not checking `errno` can allow errors to slip through.

```c
// ... include stdio.h, math.h, errno.h, string.h
int main(void) {
    errno = 0;
    printf("log(-1.0) = %f\n", log(-1.0));
    printf("%s\n", strerror(errno));
}
// Possible output:
// log(-1.0) = nan
// Numerical argument out of domain
```

Missed error reporting

- In your own code, if you need to indicate an error, consider returning the error code explicitly (e.g. as an `enum`) to make it clear that this should be handled, and consider carefully whether ignoring a possible error could cause an invalid program state, particularly if it's one that would allow (incorrect) execution to continue.

Missed error reporting

- In your own code, if you need to indicate an error, consider returning the error code explicitly (e.g. as an **enum**) to make it clear that this should be handled, and consider carefully whether ignoring a possible error could cause an invalid program state, particularly if it's one that would allow (incorrect) execution to continue.
- A common pattern for error reporting (used, for example, in the CUDA library) is to have return values be the error code, with output parameters passed as pointer arguments when needed. A (slightly overkill) example:

```
typedef enum {  MET_SUCCESS = 0, MET_INDEX_TOO_LARGE,
                MET_UNINITIALIZED, MET_OUTPUT_NULL } MyErrorType;
MyErrorType list_sum_to_index(List list, size_t index, double* sum) {
    if (index >= list.len) return MET_INDEX_TOO_LARGE;
    if (list.data == NULL) return MET_UNINITIALIZED;
    if (sum == NULL)       return MET_OUTPUT_NULL;
    double out = 0.0;
    for (size_t i = 0; i < index; ++i) { out += list.data[i]; }
    *sum = out; return MET_SUCCESS;
}
```

Off by one errors

- Off by 1 errors are surprisingly common. For example, accessing element `N` of an array of length `N` would access memory one element past the end! This can be non-obvious, for example:

```
int* arr = malloc(sizeof(*arr) * N);
int i = 0;
while (i < N) {
    ++i;
    arr[i] = i * i;
}
```

- This class of error is often one to look out for as the root cause behind a logic or memory bug.

Questions on common bugs?

`printf()`

`printf()`

`printf` (or other forms of logging) are often the first debugging tool to reach for.

Advantages:

- Easy way to inspect variables or check control flow.
- Doesn't require external software.
- Hands-off – you can let the program run and then just inspect the log afterwards.

`printf()`

`printf` (or other forms of logging) are often the first debugging tool to reach for.

Advantages:

- Easy way to inspect variables or check control flow.
- Doesn't require external software.
- Hands-off – you can let the program run and then just inspect the log afterwards.

Disadvantages:

- Need to re-compile to test new things (could be slow).
- Can become time consuming scrolling through output from a long loop.
- Existence of the `printf` call can change behaviour of optimised builds!
- More difficult to debug parallel code.

`printf()`

Some general tips:

- If you're not sure where to start looking, try printing out various `"I'm here"` strings to narrow down the problem. From there, you can start bisecting.

printf()

Some general tips:

- If you're not sure where to start looking, try printing out various `"I'm here"` strings to narrow down the problem. From there, you can start bisecting.
- The `__FILE__` and `__LINE__` preprocessor macros can help make a handy re-usable print debugging marker. For example:

```
#define PRINT_HERE printf("Reached %s:%d\n", __FILE__, __LINE__)
```

printf()

Some general tips:

- If you're not sure where to start looking, try printing out various `"I'm here"` strings to narrow down the problem. From there, you can start bisecting.
- The `__FILE__` and `__LINE__` preprocessor macros can help make a handy re-usable print debugging marker. For example:

  ```
  #define PRINT_HERE printf("Reached %s:%d\n", __FILE__, __LINE__)
  ```
- If your code is slow to compile or run, it is often better to add more print statements than you think you need to help narrow down the problem faster.

printf()

Some general tips:

- If you're not sure where to start looking, try printing out various `"I'm here"` strings to narrow down the problem. From there, you can start bisecting.
- The `__FILE__` and `__LINE__` preprocessor macros can help make a handy re-usable print debugging marker. For example:

  ```
  #define PRINT_HERE printf("Reached %s:%d\n", __FILE__, __LINE__)
  ```
- If your code is slow to compile or run, it is often better to add more print statements than you think you need to help narrow down the problem faster.
- Format what you print in a way that's easily searchable with tools like `grep`.

printf()

Some general tips:

- If you're not sure where to start looking, try printing out various `"I'm here"` strings to narrow down the problem. From there, you can start bisecting.
- The `__FILE__` and `__LINE__` preprocessor macros can help make a handy re-usable print debugging marker. For example:

```
#define PRINT_HERE printf("Reached %s:%d\n", __FILE__, __LINE__)
```

- If your code is slow to compile or run, it is often better to add more print statements than you think you need to help narrow down the problem faster.
- Format what you print in a way that's easily searchable with tools like `grep`.
- Use `if` statements to avoid printing irrelevant information.

printf()

Some general tips:

- If you're not sure where to start looking, try printing out various `"I'm here"` strings to narrow down the problem. From there, you can start bisecting.
- The `__FILE__` and `__LINE__` preprocessor macros can help make a handy re-usable print debugging marker. For example:

```
#define PRINT_HERE printf("Reached %s:%d\n", __FILE__, __LINE__)
```

- If your code is slow to compile or run, it is often better to add more print statements than you think you need to help narrow down the problem faster.
- Format what you print in a way that's easily searchable with tools like `grep`.
- Use **`if`** statements to avoid printing irrelevant information.
- Be careful to use the correct format string. For example, printing out a floating point number with the `%d` specifier will make it look wrong even if it's correct. Compiler warnings are good at catching this for you!

`gdb`

gdb

`gdb` is the next step up from using `printf`. It's a powerful command line debugger which is available on most systems.

Advantages:

- Can set breakpoints, inspect variables, step through code, and walk up and down the call stack.
- No need to recompile code (so long as it was compiled with `-g` for debug symbols).
- Conditional breakpoints allow code to run until the (expected) problem conditions occur, then stop and allow inspection.
- Command-line interface is easy to use even through a remote session.

gdb

`gdb` is the next step up from using `printf`. It's a powerful command line debugger which is available on most systems.

Advantages:

- Can set breakpoints, inspect variables, step through code, and walk up and down the call stack.
- No need to recompile code (so long as it was compiled with `-g` for debug symbols).
- Conditional breakpoints allow code to run until the (expected) problem conditions occur, then stop and allow inspection.
- Command-line interface is easy to use even through a remote session.

Disadvantages:

- Difficult to debug parallel code, and doesn't work with MPI.
- Default interface is command-line only (but some GUI wrappers exist for editors like Visual Studio Code).
- Can be difficult to use with optimisations turned on (although this applies to any debugger)

Basic `gdb` usage

Basic `gdb` usage:

- run with `gdb ./my_exe` or `gdb --args ./my_exe arg1 arg2`

Basic `gdb` usage

Basic `gdb` usage:

- run with `gdb ./my_exe` or `gdb --args ./my_exe arg1 arg2`
- Set a breakpoint with `break main.c:10` or `break function_name`
  - Delete a breakpoint with `delete breakpoint#` where `breakpoint#` is the number of the breakpoint to delete.
  - `info breakpoints` gives information about all current breakpoints (including their `breakpoint#`)
  - `delete` without an argument deletes all breakpoints
  - Breakpoints can be set to trigger only on a condition (C syntax) with `break file:line if condition`

Basic `gdb` usage

Basic `gdb` usage:

- run with `gdb ./my_exe` or `gdb --args ./my_exe arg1 arg2`
- Set a breakpoint with `break main.c:10` or `break function_name`
  - Delete a breakpoint with `delete breakpoint#` where `breakpoint#` is the number of the breakpoint to delete.
  - `info breakpoints` gives information about all current breakpoints (including their `breakpoint#`)
  - `delete` without an argument deletes all breakpoints
  - Breakpoints can be set to trigger only on a condition (C syntax) with `break file:line if condition`
- Similarly, `watch` sets a watch point on a variable to stop whenever that variable changes (good for catching memory bugs).

Basic `gdb` usage

Basic `gdb` usage:

- run with `gdb ./my_exe` or `gdb --args ./my_exe arg1 arg2`
- Set a breakpoint with `break main.c:10` or `break function_name`
  - Delete a breakpoint with `delete breakpoint#` where `breakpoint#` is the number of the breakpoint to delete.
  - `info breakpoints` gives information about all current breakpoints (including their `breakpoint#`)
  - `delete` without an argument deletes all breakpoints
  - Breakpoints can be set to trigger only on a condition (C syntax) with `break file:line if condition`
- Similarly, `watch` sets a watch point on a variable to stop whenever that variable changes (good for catching memory bugs).
- Start running the program with `run`. It will run until the program reaches a breakpoint, completes, or crashes. Ctrl+C will also stop the program (but not `gdb`).

Basic gdb usage

Basic gdb usage:

- run with `gdb ./my_exe` or `gdb --args ./my_exe arg1 arg2`
- Set a breakpoint with `break main.c:10` or `break function_name`
  - Delete a breakpoint with `delete breakpoint#` where `breakpoint#` is the number of the breakpoint to delete.
  - `info breakpoints` gives information about all current breakpoints (including their `breakpoint#`)
  - `delete` without an argument deletes all breakpoints
  - Breakpoints can be set to trigger only on a condition (C syntax) with `break file:line if condition`
- Similarly, `watch` sets a watch point on a variable to stop whenever that variable changes (good for catching memory bugs).
- Start running the program with `run`. It will run until the program reaches a breakpoint, completes, or crashes. Ctrl+C will also stop the program (but not `gdb`).
- `kill` will exit the program (but not `gdb`)

Basic `gdb` usage

Basic `gdb` usage:

- run with `gdb ./my_exe` or `gdb --args ./my_exe arg1 arg2`
- Set a breakpoint with `break main.c:10` or `break function_name`
  - Delete a breakpoint with `delete breakpoint#` where `breakpoint#` is the number of the breakpoint to delete.
  - `info breakpoints` gives information about all current breakpoints (including their `breakpoint#`)
  - `delete` without an argument deletes all breakpoints
  - Breakpoints can be set to trigger only on a condition (C syntax) with `break file:line if condition`
- Similarly, `watch` sets a watch point on a variable to stop whenever that variable changes (good for catching memory bugs).
- Start running the program with `run`. It will run until the program reaches a breakpoint, completes, or crashes. Ctrl+C will also stop the program (but not `gdb`).
- `kill` will exit the program (but not `gdb`)
- `quit` or `exit` will exit `gdb`

Basic `gdb` usage

With the program stopped:

- Use `info locals` to show all current local variables, and `info args` for arguments to the current function.

Basic `gdb` usage

With the program stopped:

- Use `info locals` to show all current local variables, and `info args` for arguments to the current function.
- Print out the value of a variable with `print expr`, where `expr` is generally a C-syntax expression (typically a variable name).
  - For array elements use `print arr[0]`
  - For all elements of a stack array, just `print arr`
  - For `len` elements of a heap array, use `print *arr@len`
  - Math works, so we can get an array slice with `print *(arr+start)@len`
  - Can format the printing with `print/f` (for example, for floating point formatting)

Basic gdb usage

With the program stopped:

- Use `info locals` to show all current local variables, and `info args` for arguments to the current function.
- Print out the value of a variable with `print expr`, where `expr` is generally a C-syntax expression (typically a variable name).
  - For array elements use `print arr[0]`
  - For all elements of a stack array, just `print arr`
  - For `len` elements of a heap array, use `print *arr@len`
  - Math works, so we can get an array slice with `print *(arr+start)@len`
  - Can format the printing with `print/f` (for example, for floating point formatting)
- `step` will execute the next line of source code, or `next` will do the same but without descending into function calls.

Basic gdb usage

With the program stopped:

- Use `info locals` to show all current local variables, and `info args` for arguments to the current function.
- Print out the value of a variable with `print expr`, where `expr` is generally a C-syntax expression (typically a variable name).
  - For array elements use `print arr[0]`
  - For all elements of a stack array, just `print arr`
  - For `len` elements of a heap array, use `print *arr@len`
  - Math works, so we can get an array slice with `print *(arr+start)@len`
  - Can format the printing with `print/f` (for example, for floating point formatting)
- `step` will execute the next line of source code, or `next` will do the same but without descending into function calls.
- `backtrace` will print out the current call stack, and `up` and `down` will go up or down the call stack by one function.

Basic `gdb` usage

- We can get variables to automatically be printed whenever the program stops with `display expr`
  - Use `info display` to see all active `display` commands
  - Use `undisplay display#` to remove one from the list

Basic gdb usage

- We can get variables to automatically be printed whenever the program stops with
  `display expr`
  - Use `info display` to see all active `display` commands
  - Use `undisplay display#` to remove one from the list
- We can also automatically execute other commands when a breakpoint is triggered with (e.g.)

```
commands breakpoint#
watch foo
end
```

Basic `gdb` usage

- We can get variables to automatically be printed whenever the program stops with `display expr`
  - Use `info display` to see all active `display` commands
  - Use `undisplay display#` to remove one from the list
- We can also automatically execute other commands when a breakpoint is triggered with (e.g.)

```
commands breakpoint#
watch foo
end
```

- We can alter execution by setting variables or returning early:
  - `return expr` will return immediately from the current function with the value of `expr` (which is just a C-syntax expression).
  - `set variable_name = expr` will overwrite a variable with a new value

Basic gdb usage

- We can get variables to automatically be printed whenever the program stops with `display expr`
  - Use `info display` to see all active `display` commands
  - Use `undisplay display#` to remove one from the list
- We can also automatically execute other commands when a breakpoint is triggered with (e.g.)

```
commands breakpoint#
watch foo
end
```

- We can alter execution by setting variables or returning early:
  - `return expr` will return immediately from the current function with the value of `expr` (which is just a C-syntax expression).
  - `set variable_name = expr` will overwrite a variable with a new value
- `finish` will continue execution until the end of the function, or `continue` will run until the end of the program (or the next breakpoint/crash)

For a more detailed reference, see http://users.ece.utexas.edu/∼adnan/gdb-refcard.pdf

Let's experiment with `gdb` on yesterday's `List` code.

Let's experiment with `gdb` on yesterday's `List` code.

The goal from yesterday was an interface that looked like this:

In Python, we can do this:

```python
a = []
for v in range(6):
    a.append(v)
a.insert(3, 2.5)
a.pop(1)
print(a)
# Prints:
# [0, 2, 2.5, 3, 4, 5]
```

In C, we'd like to be able to do similar:

```c
List a = list_init();
for (int v = 0; v < 6; ++v)
    list_append(&a, v);
list_insert(&a, 3, 2.5);
list_pop(&a, 1);
list_print_contents(a);
// Prints:
// [0, 2, 2.5, 3, 4, 5]
```

But we never got around to implementing `list_pop()` or `list_insert()`!
Let's try implementing `list_pop()` now and see how `gdb` can help.

`valgrind`

valgrind

`valgrind` has a number of tool options. The default, and the focus of this session, is `memcheck`.

`memcheck` helps find memory bugs such as:

- Use after free and double free
- Leaks (pointer falls out of scope without calling `free`)
- Invalid access (e.g. reading/writing past the end of an array)

valgrind

valgrind has a number of tool options. The default, and the focus of this session, is memcheck.

memcheck helps find memory bugs such as:

- Use after free and double free
- Leaks (pointer falls out of scope without calling free)
- Invalid access (e.g. reading/writing past the end of an array)

Advantages:

- Good at finding memory problems
- Can use with optimisations turned on (although will occasionally give false positives)

`valgrind`

`valgrind` has a number of tool options. The default, and the focus of this session, is `memcheck`.

`memcheck` helps find memory bugs such as:

- Use after free and double free
- Leaks (pointer falls out of scope without calling `free`)
- Invalid access (e.g. reading/writing past the end of an array)

Advantages:

- Good at finding memory problems
- Can use with optimisations turned on (although will occasionally give false positives)

Disadvantages:

- Program could run many times slower, and use significantly more memory
- Can sometimes throw false positives when code does complex things (but it's right 99% of the time), or occasionally miss things.
- Can give a lot of output to search through – generally start fixing things from the start since later issues could be caused by earlier ones.

valgrind

We can run `valgrind` with:

```
$ valgrind [valgrind arguments] my_exe [my_exe arguments]
```

valgrind

We can run `valgrind` with:

```
$ valgrind [valgrind arguments] my_exe [my_exe arguments]
```

To differentiate `valgrind` output from program output, it's formatted as:

```
==12345== some-message-from-valgrind
```

where `12345` is the process ID (helps differentiate multiple processes from a single program).

`valgrind`

We can run `valgrind` with:

```
$ valgrind [valgrind arguments] my_exe [my_exe arguments]
```

To differentiate `valgrind` output from program output, it's formatted as:

```
==12345== some-message-from-valgrind
```

where `12345` is the process ID (helps differentiate multiple processes from a single program).

Alternatively, output can be redirected to a log file with `--log-file=filename`. In this case, the file name can include `%p`, which will be replaced with the process ID to get one file per process. This is especially useful for HPC software that uses parallel frameworks like MPI, in which case `--trace-children=on` needs to be set.

valgrind

We can run `valgrind` with:

```
$ valgrind [valgrind arguments] my_exe [my_exe arguments]
```

To differentiate `valgrind` output from program output, it's formatted as:

```
==12345== some-message-from-valgrind
```

where `12345` is the process ID (helps differentiate multiple processes from a single program).

Alternatively, output can be redirected to a log file with `--log-file=filename`. In this case, the file name can include `%p`, which will be replaced with the process ID to get one file per process. This is especially useful for HPC software that uses parallel frameworks like MPI, in which case `--trace-children=on` needs to be set.

Duplicates of the same error are only reported the first time by default. To show all of them, use the `-v` flag.

Let's experiment with `valgrind` on some of our `List` code.

More details of `valgrind` options can be found here:

https://valgrind.org/docs/manual/manual-core.html

Hands-on

Hands-on: use `gdb` and `valgrind` to help identify and fix issues in a finite difference solver.

Arm Forge

Arm Forge is a proprietary set of debugging and profiling tools available on Gadi. It includes the Arm DDT debugger, and the Arm MAP profiler.

Arm Forge

Arm Forge is a proprietary set of debugging and profiling tools available on Gadi. It includes the Arm DDT debugger, and the Arm MAP profiler.

Advantages:

- Supports MPI, OpenMP, and GPUs* – essential for debugging parallel HPC code.
- Graphical interface (accessible via ARE virtual desktop)
  - Easy fine-grained control over memory debugging
- Similar feature set to `gdb` + `valgrind`
- Can attach to an already-running process
- Also supports Python (with some limitations)

Arm Forge

Arm Forge is a proprietary set of debugging and profiling tools available on Gadi. It includes the Arm DDT debugger, and the Arm MAP profiler.

Advantages:

- Supports MPI, OpenMP, and GPUs* – essential for debugging parallel HPC code.
- Graphical interface (accessible via ARE virtual desktop)
  - Easy fine-grained control over memory debugging
- Similar feature set to `gdb` + `valgrind`
- Can attach to an already-running process
- Also supports Python (with some limitations)

Disadvantages:

- Proprietary
- Can be overkill for simple debugging tasks.
- Similar downsides to `valgrind` when doing memory analysis

* GPUs not available with current NCI license - can use NVIDIA tools instead

Arm Forge

Name	Description
basic	Detect invalid pointers passed to memory functions (such as malloc, free, ALLOCATE, and DEALLOCATE)
check-funcs	Check the arguments of addition functions (mostly string operations) for invalid pointers.
check-heap	Check for heap corruption, for example, due to writes to invalid memory addresses.
check-fence	Check the end of an allocation has not been overwritten when it is freed.
alloc-blank	Initialize the bytes of new allocations with the known value of dmalloc-alloc byte (hex 0xda, decimal 218).
realloc-copy	Always copy data to a new pointer when reallocating a memory allocation (for example, due to realloc).

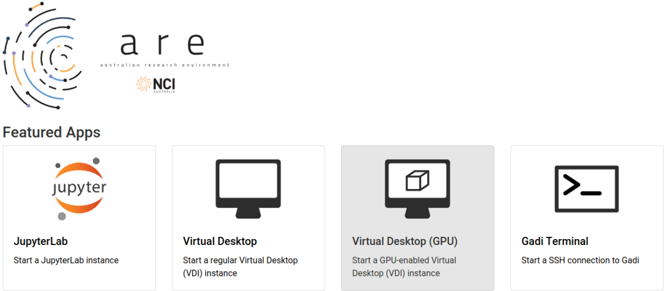
Arm Forge - Memory debugging options

free-blank	Overwrite the bytes of freed memory with the known value of the dmalloc-free byte (hex 0xdf, decimal 223). If this check is enabled, the library overwrites memory when it is freed, using dmalloc-free. This can be used, for example, to check for corrupted allocations. This also checks and reports when a free byte has been written to, which can indicate if a freed pointer is being used.
check-blank	Check to see if blanked space has been overwritten. Space is blanked when it either has a pointer allocated to it, or the pointer has been freed. This enables alloc-blank and free-blank.
free-protect	Protect freed memory where possible (using hardware memory protection) so subsequent read/writes cause a fatal error.

Note: if your binary is statically linked, check the documentation for the necessary extra compile flags to enable memory debugging.

Arm Forge

From ARE, launch a VDI Virtual Desktop instance, then open Arm Forge:



Arm Forge

From ARE, launch a VDI Virtual Desktop instance, then open Arm Forge:

VDI Desktop

Launch a regular desktop environment

Walltime (hours)

2

Number of hours your desktop can run (maximum). e.g. 1.5, 8, 24, 48

Queue

normal

Compute Size

small

Amount of CPU/Memory resources available to your desktop session

Project

vp91

Project to submit gadi job under; requires an SU allocation

Storage

scratch/vp91

/scratch/vp91 gdata/fc8 gdata/il82 gdata/m72 gdata/wx77 scratch/fc8 scratch/il82

Software

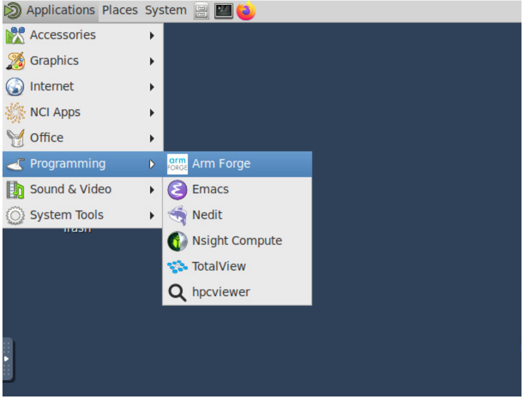
abaqus abaqus_rmit adf ansys_monash ansys_mq ansys_nci ansys_rmit

☐ **I would like to receive an email when the session starts**

Advanced options ...

Arm Forge

From ARE, launch a VDI Virtual Desktop instance, then open Arm Forge:



The user documentation is available here:
https://developer.arm.com/documentation/101136/22-1-1/DDT/Get-started-with-DDT?lang=en

Hands-on

Try adding a `list_insert` function to our `List` library and debugging any problems.

It should have the signature:

```c
void list_insert(List* list, size_t index, double value);
```

For example:

```c
list_print_contents(list);    // prints: [0, 1, 2, 3, 4, 5]
list_insert(&list, 2, 1.5);   // Insert the value 1.5 at index 2
list_print_contents(list);    // prints: [0, 1, 1.5, 2, 3, 4, 5]
```

Summary

Treat debugging like a scientific investigation!

Common Tools:

- `printf` and logging
- `gdb`
- `valgrind`
- Arm DDT (or TotalView offers similar capabilities)

Common Bugs

- Bohrbug
  - “Good solid bug"
  - Doesn’t change its behaviour
- Heisenbug
  - “Know either what it does or where it is"
  - Disappears when you attempt to debug
- Schrödinbug
  - “Both a bug and not a bug"
  - A bug in code that never should have worked
- Mandelbug
  - “A fractal bug"
  - Chaotic and non-deterministic