Software Design and Development

Checking and Diagnostics

N.M. Maclaren

nmm1@cam.ac.uk

September 2019

2.1 Introduction

This lecture covers the most useful coding techniques that you can use to make your code largely self-checking – people are often surprised that is possible, but it is. One reason for doing this is that it is not always possible to use debuggers; they do not always work under schedulers, MPI and so on, but you can almost always use these methods. These methods are also all suitable for use in production code; most research projects involve ongoing change, so that debugging programs while doing real analyses is a necessity.

Warning: You will not use all of them in every program; remember that you have to use your judgement.

2.1.1 Reasons for Checking

A lot of checks in code is one sign of competence in a programmer, and good programmers often check everything before every use, if the cost is not too much. The reason for this is such 'random' checks will often pick up unexpected bugs – for example:

Function `A` makes object `W`'s value invalid
Function `B` uses `W` and mangles object `X`
Function `C` uses `X` and overflows array `Y`
This causes the **unrelated** structure `Z` to be trashed
Much later the function `D` uses `Z` and crashes

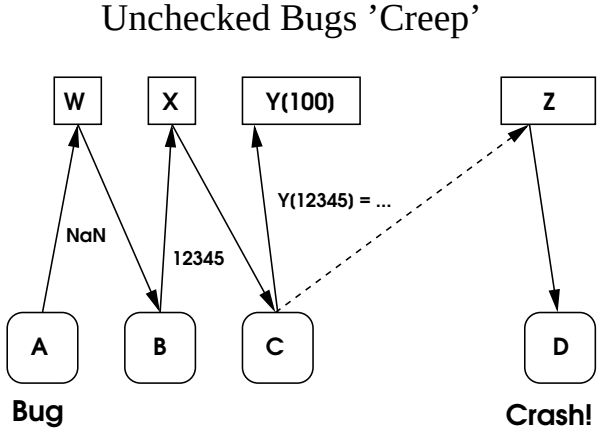

Figure 1.1

Checking `X` in `C` catches **an** error (though not the original one), hopefully before too much evidence has been lost. You still have to track back to the original error in `A`, but your task is vastly easier. It would be even better if `B` had checked the value of `W`, of course.

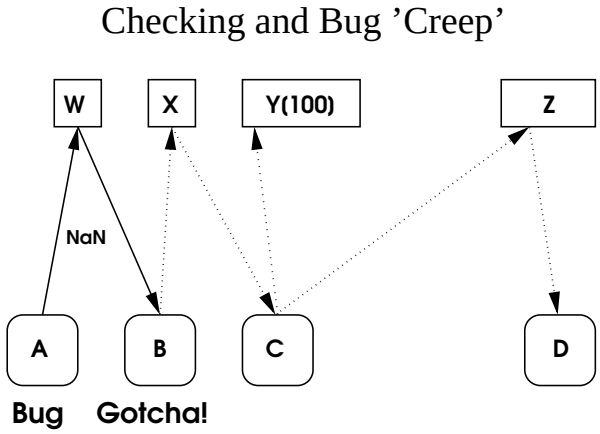

Figure 1.2

The worst common problems in many codes are:

- Invalid array indices and pointers, due to bounds errors, using uninitialised data, scoping errors (including dangling pointers) etc.
- Race conditions and related bugs, sometimes due to timing effects but not always. These are most common in parallel codes.
- Code bugs causing optimisation problems. These are almost always because you have broken some subtle language rule or constraint, causing the compiler to make a false assumption; true optimisation bugs are rare.

All tend to disappear or change symptom very easily, including **any** code change (sometimes in completely unrelated code) or difference in compiler options, including any changes due to the preprocessor use. You should minimise use of recompilation as a way of getting diagnostics, but note that even run-time environment changes can provoke problem movement.

Even worse are unpredictable problems. Race conditions are very common in some forms of parallel code, but there are many other fairly common causes (usually due to hidden parallelism). The failure may even be probabilistic and occur when the same executable is run on the same data in the same environment. The symptoms are usually predictable, but may move around. In **very** nasty cases, the failure may be both rare and occurs well into the run.

- It is worth doing a lot to avoid such problems arising in the first place, because it is not easy to track down such bugs statistically.

Be warned, however, by the moral of the James Thurber story *The Bear Who Let It Alone*: *You might as well fall flat on your face as lean over too far backward.*

Adding a lot of checking code takes time, and checking code can itself include bugs, so there is an optimum amount to maximise coding efficiency. This is a far higher proportion than most programs have, but it is still possible to have too much.

- Deciding the level is a matter of judgement.

2.1.2 Numeric errors

Numeric errors like overflow, division by zero, etc. are a particular issue. Very little is trapped and diagnosed nowadays, often only integer division by zero, and almost all errors will return a nonsense value and continue. You must do any checking yourself, and this is especially true for complex arithmetic. There is much more on this in the next lecture.

The reason for being so concerned about numeric errors is that they do not just cause obviously wrong output. Untrapped numeric errors often cause logic errors, untrapped logic errors often cause overwriting errors and untrapped overwriting errors often cause crashes – or, **much worse**, often cause nonsense output. In really nasty cases, the output looks right, but is completely wrong.

- A consequence of this sort of problem is that lots of simple checks on a wide range of values is best – a few, perfect checks helps **less** to detect this sort of corruption.

The also help when you make changes later and forget what you were assuming elsewhere; when I make errors like that, quite a lot of them fail on my own checks. You can check that values, indices and bounds are in range or check consistency properties; such checks are often very simple, such as `if A < 0 then Y > 2`.

2.1.3 Test Methodology

You should test each component before including it, though you sometimes need to test several together if they are not functional on their own. This is much less confusing than testing the whole program, and so quicker and more reliable. Remember to test the error handling, at least roughly, because that helps to avoid wasting time with later failures getting confused over whether the error handling or the main code is broken. This will not pick up all bugs, and exceptional cases are particularly likely to escape, so do not assume that tested means bug-free.

You should keep your test data and the output from it, so that you can rerun it and check that nothing has broken – this is known as regression testing. The same applies to unit test programs and data, and it is often useful to put components in libraries to help with this; you can include them in your program or run a unit test on them. Generally, when you make a significant change to your code, you should rerun appropriate regression tests and check them. Automated testing is a vast saving in effort – it is not a perfect solution, but it can save a lot of manual debugging.

2.2 The Object Approach

2.2.1 Object Orientation

Most of these techniques will be described in terms of object-orientation; remember that it is an approach, not a dogma, and the techniques are useful far more generally. As used in this course, an object is a set of data that is connected together in some way and is changed and accessed together. It may be (and often is) merely a collection of scalars and arrays.

A practical point is that an object is often made up of sub-objects, and you can use this structure to keep your code simple. For example, instead of duplicating code, you can call the next level down, and you can use recursion if it matches your structure.

It is useful to tag each object with an identifier; Unix file formats often use '*magic numbers*' for that purpose. For example:

```
#define WOMBAT_ID 3579138481
typedef struct {
        int id;         // Always WOMBAT_ID
        ...
} wombat;
```

- You should choose a value that is unlikely to occur very often by accident.

This is usually some fixed-length text or an integer, but any type can be used. You should use this for checking that a pointer refers to an initialised value of the right type; it is also very useful when using an interactive debugger, in the same way. Obviously, the identifier needs to be in a known location; it is simplest to put it at the very start of the structure, and that is what most people do, but it is not essential.

- This is very useful for C and C++, but much less so in Fortran.

This is because the less type safe the language is, the more useful an identification field is; it is pointless in a truly type-safe language. The technique can be extended in many ways – for example, it could include a readable identifier, and a more secure hash code. That is complete overkill for most scientific programming, but can occasionally be useful. For example, the following will detect if `memcpy` has been used rather than calling the proper copy constructor, which can be useful for RDMA and MPI buffers.

```
typedef struct {
        char[8];         // Always "Wombat"
        uintptr_t hash;          // (&object)^HASH_CODE
        ...
} wombat;
```

2.2.2 Basic Object Methods

This gives some details that were omitted in the previous lecture.

- Almost always initialise objects explicitly.

This does not apply to just static data, but to automatic (stack) and and dynamic ('heap', allocated etc.) You should not trust the automatic clearing to zero that is specified by C and C++, because the standards do not say what most people think they do; it will almost always work, and then may fail unexpectedly on a new system. Despite common belief, initialisation is **not** too expensive, because the cost is only linear in the size of the object, and the cost of using it is usually much more.

- You should use a 'constructor' to create objects.

That is a single procedure to allocate and initialise a particular set of data. This need not be a formal constructor, even in C++, but should be a single, well-defined place (or perhaps a couple of places) where construction occurs.

This often does both the memory allocation and the initialisation, but may initialise only for objects where the space is allocated by the language. Having a single constructor keeps the initialisation consistent, and is a useful hook to add diagnostic code.

- It is best to use an invalid value if the object is initially unset.

This should preferably be one that causes a crash if used, but that is not always feasible. That may seem strange, but a crash is better than unpredictably wrong results; initialising to zero (or, more generally, a default value) has its uses, though. Unset values can be valid but implausible ones like -1.23×10^{300} or -123456789, and IEEE 754 NaNs are useful for this, in compilers that support them.

An advanced diagnostic trick is to vary the value, to see if bugs move around, or one can even use different values to flag the history of data. But avoid such tricks unless you are a fairly skilled programmer.

- Do not forget to use an explicit 'destructor'.

This is the converse of a constructor and is a useful hook for checking and tracing. Consider resetting the contents of an object to invalid values as the last action before freeing the data or releasing it; this is very rarely done, but can be very useful. It is worthwhile mainly if the code may have used a pointer to the object, because your code may have one saved somewhere that will outlast the object's lifetime. Almost all C and C++ programs are like that, and it can be useful in some non-pointer codes.

Huge sparse arrays are the one case where initialisation can dominate; these are arrays that are gigabytes or terabytes in size but where only a small fraction (usually less than 1%, sometimes very much less) is used, and the rest of the array is zero. They are a lazy way of making the virtual memory system do indexed lookup, and are an absolute nightmare in a great many ways. Whether they work is system-dependent (sometimes systems allocate all the pages anyway), you cannot use memory limits to trap runaway code, and they introduce other problems.

- Doing it yourself is easy and more flexible, especially in a language like C++, and the caching algorithm can be tuned for your application.

2.2.3 Enabling Diagnostics

I do not like preprocessor options much, because you have to rebuild the code to add diagnostics, and many of the nastier problems then move around, as described above.

- I strongly recommend a **run-time** option, where you can select the diagnostic level you want.

You can select this by an environment variable, or by a program argument that sets a flag (`-v` is common), or by whether a suitable file exists, or by a command in the input, or in any other suitable way. This typically needs a throughness parameter, which is set as the value; for example, bounds checking etc., checking all values, and everything including cross-checks. It can be useful for your run-time option to set the default, and to be able to override that in the call.

- Exactly the same remarks apply to tracing and object display, and you may prefer separate options for those.

The minimum costs are then testing the option value, which is a single, scalar global, and so access to it is efficient because it will usually be in the cache. Some minimal C/C++ and Fortran examples of its use are:

```
#include "diag.h"

if (diag_level > 0) check_object(diag_level, ...);
```

```
USE diag

IF (diag_level > 0) CALL check_object(diag_level, ...)
```

You can use the C and C++ `assert` macro if you like, but it does not do anything that you cannot do better yourself, very easily.

2.2.4 Object Display

- All objects should have a display primitive.

This displays the contents so that you can see what they are; as a reminder, this is for diagnostics and not the results. It is merely a convenience, but can save a huge amount of effort, and can be very useful with some debuggers, as described later. It typically needs a level parameter, which says how far to indirect in structured data and how much of large arrays to display, or you can have more than one primitive.

- Remember **not** to assume that the object is correct.

You very often want to display broken data, when investigating problems. It should work if pointers are null or not allocated, it should check indices and pointers for being in range before using them, and it should assume **any** values, whether they are possible in correct execution or not. You may want to call the checker before calling it, or may call it from the checker if that fails; the latter is probably the most generally useful approach.

2.2.5 Object Checking

- All objects should have a checking primitive.

This answers the question *Is this object vaguely correct?* For example, are the values within their limits, and is the object self-consistent. It is tedious to write, but **incredibly** useful. You can call it automatically, or insert calls manually, and can call it from many debuggers, too.

- Design objects to be thoroughly checkable.

You should keep the data clean (e.g. arrays should be always defined or unset), with checkable constraints. Where reasonable, you should make the data redundant and maintain invariants on the redundancy (described later).

- Generally call the checker automatically, at least once.

This is to ensure that the checking code remains correct, though it has other advantages. Suitable places are at the end of initialisation, at the start of termination, when you have detected an error, and so on; none of those should happen very often. I strongly recommend adding a lot more calls, which is the most important reason for a run-time option; without that, doing so is too expensive.

If you do that, you can set it high for debugging, and lower for production code. If you hit a problem, then just rerun with checking set to high.

Manual use is a very effective way of debugging, and it is the way that I generally debug non-trivial code. Let us say that an object goes bad after 30 minutes running, and the program fails. You can checks where they will be called fairly often, and the first failure will tell you more precisely where the error started. So you work out why, add suitable checks for that, and repeat until you have located where the problem arose.

You can often call procedures from debuggers, and calling checking procedures saves a lot of effort, compared to checking objects by hand.

`DPOSV` is the LAPACK Cholesky solver, and this is an example of checking arrays before and after its call:

```
call check\_upper (n, a, lda)
call check\_rect (n, nrhs, b, ldb)
call dposv ('u', n, nrhs, a, lda, b, ldb, info)
call check\_upper (n, a, lda)
call check\_rect (n, nrhs, b, ldb)
```

The calculation is $O(n^3)$, but simple checking is only $O(n^2)$.

2.2.6 Invariants

- Invariants are things that are always true; in the context of programming, this means from then end of initialisation to the start of termination, except possibly inside one of the object's methods which is in the process of updating it.

If they are ever false, then something has gone wrong wrong – it might be a logic error in your code or it might be overwriting. You may not be able to tell what happened, but

you know that you have a problem. Invariants can be programmatic (e.g. array indices), numeric (e.g. values must be within certain limits), or properties like an array must be positive definite.

- Invariants have the immensely useful property that they can be checked essentially anywhere.

This is very useful for tracking down where things have failed. If you write a procedure to check an invariant, and you have a problem where the invariant becomes false, you can put calls to the procedure everywhere in the code and see where the invariant first failed to be true. It may help to show some code; the example chosen is an indexed array of real numbers, where the numbers and indices are stored in separate arrays, and not all of the array need be in use. Fortran arrays start at 1, by default.

```
INTEGER :: used, index(size), j
REAL(FP) :: data(size)

IF (used < 1 .OR. used > SIZE) CALL Diag(...)

DO j = 1 , size
    IF (index(j) < 1 .OR. index(j) > size) CALL Diag(...)
END DO
```

The first check is very basic and cheap, and can be used everywhere; the second is linear in time, but more powerful. However, both of them will always be true if the data are valid.

To make the best use of such invariants, you should initialise all of `INDEX` to an invalid value (e.g. –123456789) and initialise all of `DATA` to (say) -1.0e300 or a NaN. You should also remember to reset the array elements to those values when they cease being used. You can now check that the valid values of `INDEX` match `USED`; all before `USED` should be good and point to valid data, and all after are bad and should have the 'invalid' value. You could also check that precisely the indexed values of `DATA` are valid. This will also detect some random overwriting.

Generally, that is overkill, but it is useful to have a procedure to do that, which you can call if you suspect a problem, as described above – and, for that, you need the data to be maintained in the consistent state. Scalar invariants (such as the first one in the example) are generally more useful, because they cost almost nothing to check, and pick up a great many mistakes.

- Whenever you get the opportunity, create, maintain and use invariants.

2.3 More on Checking and Tracing

2.3.1 Argument and Result Checking

Ideally, a major procedure should check all of its input arguments on entry, and its result and output arguments before returning, such as:

```
double operate (double left [ ], int right [ ]) {
    check_double_array(left);
    check_int_array(right);
    . . .
    result = . . .
    check_value(result);
    return result;
}
```

Even if you do not do that, you should at least check such things as array bounds and other critical scalars. The purpose of checking the results is to catch errors that may be made by the procedure or other ones that it calls, before they spread further.

2.3.2 Tracing

The most common form of tracing is tracing the control flow, and it helps to answer the question *How did we get* **here***?* But you can also trace events, data flow and state changes, which help to answer the question *How did we get into* **this** *mess?* Yes, compilers and debuggers should provide these as options, but they regard it as someone else's problem. Let us start with simple function tracing in Fortran:

```
FUNCTION Fred (x, y, z)
    USE Diagnose
    INTEGER :: Fred, x, y, z
    IF (diag\_flag) CALL Diag ('Fred', 0)
     . . .
    IF (diag\_flag) CALL Diag ('Fred', 1)
END FUNCTION  Fred
```

You can add the calls using a suitable preprocessor (e.g. a Python script) or by hand. A similar C/C++ example is:

```
#define DIAG(X,Y) if (diag_flag) Diag(x,y);

#include "diagnose.h"
    int fred (int x, int y, int z) {
    DIAG ('fred', 0)
    . . .
    DIAG ('fred', 1)
}
```

Or can add them in just the same way as for Fortran, provided that your coding style is systematic enough.

- You usually want to trace the critical argument and result data.

The details are entirely dependent on your requirements. It might be just the identity of an object being acted upon (e.g. an index), but it might include some of the argument values, a summary of the action, or anything else useful.

As mentioned above, it is best if `diag_flag` is a run-time option, because you can enable and disable it without recompiling. Tracing can produce a lot of output, so it is generally best to trace to a file (possibly selectable by a run-time option), and not a standard output unit or stream.

- You may need to select the type and level of tracing.

For example, consider file tracing: the lowest level might be open and close, followed by all control operations, and finally all transfers. Or state changes: the main changes, all changes, and finally all uses of the relevant state. Remember, the decision should be taken primarily on a basis of what saves you most time.

Do not forget that there may be more than one `return` statement, and reaching the end of the procedure is an implicit return. Also, in C and C++, remember that there are alternate return mechanisms: `setjmp/longjmp` and `try/catch/throw`. Tracing these is particularly useful, as they are hard to use correctly.

- You can flush the output to the file each time for safety.

You can do this by calling `fflush` in C or C++, or the `FLUSH` statement or subroutine in Fortran; or, in C and C++, using `setvbuf(<file>,NULL,BUFSIZ,_IOLBF)` on the file. Without this, crashes will lose data, but this can cause quite a performance impact, which is a case for considering another run-time option.

2.3.3 Use Of Tracing

In the tracing procedure, you could print the entry and exit information; as mentioned, you should do that to a file, as it can be can be voluminous. It is then easy to write tool to display the calls and returns as a tree, to display a traceback at any particular point, to count the calls of each procedure, or anything else appropriate. There are often compiler options to do those (e.g. `-pg` for use with `gprof`) but, as they stand, they are not very useful. If you have printed extra critical information, you can select on that data, and look at just the calls relevant to your specific problem.

- You can save the active names (i.e. those that would be in a traceback) in an array.

This is a trivial example of using your own stack, and you need to remember to push the stack back (i.e. 'pop' it) when procedures return. You can now can write your own traceback function , and call that when your program hits a problem or is signalled. Very few compilers provide this, but they could easily do so. However, it need not trace the returns, and could just keep a record of the last N calls (for some fixed N), which gives a history of calls rather than a traceback, and this is also useful.

- To store a history, use a circular trace buffer.

This maintains a list of the last N calls, or calls and returns; the latter is generally more useful, but is tricker to do, as described above. It is a **very** useful facility, which seems to be little taught now, and preserves the most critical data, but uses only a fixed amount of memory. Do not forget to write a procedure to display it. Each buffer saves just one kind of trace data, but there can be an arbitrary number of buffers, often dozens. It is easiest to explain graphically:

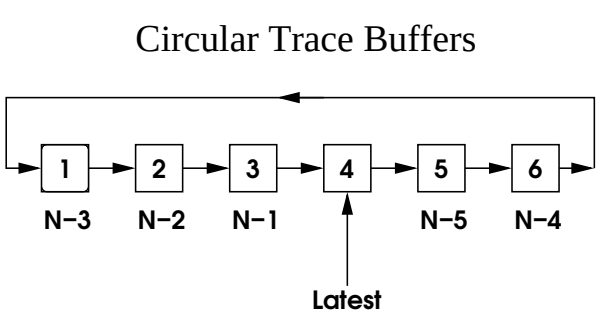

Figure 3.1

A simple C/C++ example of the data structures and a function to insert data is:

```
#define SIZE 3
static const char *names[SIZE];
static int actions[SIZE], entry = -1, looped = 0;

void trace (const char *name, int action) {
     if (++entry >= SIZE) {
         entry = 0;
         looped = 1;
     }
     names[entry] = name;
     actions[entry] = action;
}
```

And a simple function that displays its contents is:

```
void display () {
    int n = entry;
    if (n < 0) return;
    while (1) {
        cerr << names[n] << "   " << actions[n] << endl;
        if (--n < 0) {
            if (! looped) return;
            n = SIZE-1;
        }
        if (n == entry) break;
    }
}
```

- Tracing is not restricted to function calls.

You can trace any action, event, or anything of that nature, whenever you want to know the order of actions or events. For example, you can trace changes or accesses to selected data or changes to state, whether the program's or the system's. You can annotate the trace with the context, such as the component responsible for the change or access. To see a random example of the use of the technique, look at '`man mtrace`' under *Linux*.

2.3.4 Miscellaneous Points

What this lecture has been teaching is methodologies, and not just tools; they are not simply recipes to copy, but techniques that can be used for much more general purposes.

- You should always think *Should I automate this?*

The answer is very often that it is infeasible or not worth the effort, but sometimes it can save a massive amount of effort. As always, *TANSTAAFL* – automation costs time, but can save much more – and one of the hallmarks of a software engineer is good judgement about such things.

The overheads of checking and tracing are not all that much on a modern system, and depend on what the function actually does. I/O and access to a lot of data are expensive, but mere logic is cheap. The checking examples above are designed to be very cheap in the case that `diag_flag` is unset, because they just drop through, and most hardware will predict that correctly.

- It may still be too expensive to do it for all calls.

If that is the case, it can be removed from heavily used auxiliaries, and you will still get most of the benefit.

- Many debuggers can call program code, though that is of no use if the data are completely corrupt.

Calling many functions changes program state. and so calling them from debuggers is dangerous, but your checking, display and trace display functions should not do so, at least if you have coded them correctly and there is no failure! This makes such debuggers much more powerful. The Old Guard (like the author) insert such checks manually, but that is irrelevant, because you need the same primitives.

Displaying data structures is a a real problem, however you do it. Scalars are easy, but arrays are less so and pointers are worse. How far down do you want to indirect, or do you want the pointer values and target addresses? There is no general solution, and debuggers do not help; writing display functions is always tedious. You can implement your own `printf`-like solution, which can help; that is painful in Fortran, because you need one call per argument.

2.3.5 Other Information

2.3.6 More Advanced Use

These are a miscellaneous collection of fairly advanced techniques; it is worth being aware that they are possible, but avoid them until you really need them. You are **not** recommended to use them unless you have to.

- I said that global state is horrible.

Unfortunately, there is lots of it in C, C++ and POSIX, and it is a a **big** problem if it is wrong at a component boundary (i.e. component `A` sets it to a value that component `B` cannot handle). Such problems are very hard to locate, but try tracing the state and

component changes; that is generally the best method of tracking this issue down. By far the biggest problem is instrumenting your code to track the state changes and accesses, but that can be trivial if you have encapsulated all of the actions that use the state.

You often lose diagnostic output after crashes, though the techniques described above should stop that. There is an alternative approach, which is to trap most signals and close the open files; good libraries do that by default, but most do not. If you do this, you need a run-time option in order to get a dump, of course.

More usefully, such a handler can also call traceback procedures, can print out history or display critical objects, or produce any other useful diagnostics. This may not always work, but your program was about to crash anyway! The details and reasons are repulsive, but you do not need to know them – all you need to know is that this is worth trying, but is neither reliable nor portable.

Production programs often have suites of data used for testing; these are often rerun with new versions of the code to check the old data still works (often called 'regression testing'). But a lot of bugs get through, and such tests are very unreliable when there are deliberate, major changes to the output, because you cannot check the results automatically.

- Enable full checking for your test suites.

The tests will run slower, but you will have more confidence in the result, because you will know that the changes are less likely to have triggered an unexpected bug. This still will not check that the answers are **right**, both because some bugs still get through and because you may have made an error in the science or numerical analysis.

- The tracing hooks also allow use-counting or timing.

Again, you can select this mode with just a run-time option – no changes to the body of the program are needed. All the code needs to do is to count or time the actions, rather than write them to a file. These are also good places to insert checking code, or can even call back to a debugger (e.g. by calling trapped function or failing). You can do that when the context is appropriate – for example, the 1513th time that procedure `fred` calls procedure `joe` and then calls procedure `alf`.

2.3.7 Long-Running Problems

This is a stray point, but is a common requirement in scientific computing. Most systems have a fairly small job time limit – often 24 or 48 hours – both to protect against power cuts and other unexpected failure and to make routine maintenance possible. However, you often need to run analyses that will take longer.

The solution is that the program writes its current state to a file (this is often called checkpointing), and may resubmit another to the job queue as its last task before it finishes. The next job starts by restoring from the checkpoint. You obviously need to design the program so that it can work in that way, and do not need to save all workspace arrays. Most production HPC programs work this way.

- It is best to use alternate checkpoint files, as protection against a crash while one of them is being written.

The program should also check its input state carefully when it reads it, and not continue if there is any problem. The user can then restore the last good checkpoint file and rerun using that.

2.3.8 External Tools

`make` is a tool for managing the rebuilding of a program after some sources have been changed, and recompiles all changed sources and dependent files, but only those. There are many equivalent programs and derivatives, dating back to the 1970s. It is essential when the program's file structure gets complicated, because it saves a lot of build time and reduces mistakes. For a few files, a simple recompilation script is usually adequate, though not really much simpler. It is not covered in this course, but is strongly recommended.

- The golden rule of makefiles is *KISS*. Far too many make derivatives and makefiles are completely insane with features, and the resulting complexity causes serious non-portability and very hard-to-fix bugs.

Version control systems (also called *source* or *revision control systems*) are even older than make, and more recent ones include `CVS` and `subversion`; there are a zillion others. They manage source code updates and working with variant versions, and usually allow archiving, roll-back and so on. The main alternative to these is disciplined file management, such as taking snapshots of your source at frequent intervals. I do not personally like these, for a variety of reasons and, again, they are not covered in this course

- However, these are essential for projects with several developers, because coordinating updates manually is extremely error prone.

Integrated Development Environments are commonly recommended, but I regard them as very often little more than snake oil; even expressed more kindly, they are usually only a GUI toolkit for development. They often include version control (like `CVS` etc.), and an integrated equivalent to `make`. The best ones provide automated regression testing and similar facilities, but nothing that you canot do with scripts. Use them if you need to or you like them, but they will not help with debugging.

Syntax-Aware Editors were a popular bandwagon in the 1980s, and are still here, but are a near-total waste of time and money. Who spends 50% of their time fixing syntax errors? Only users on their first programming course, and perhaps senior executives and similar casual, non-technical programmers. Experienced programmers spend more like 1% of their time fixing syntax errors. These editors also make certain classes of error more common.

- What programmers **need** is run-time checking.

This includes cases of undefined behaviour (i.e. the result of executing invalid code) and, much worse, logical errors. Some compilers and debuggers do a little of this, and there may also be some special tools; Intel has some tools for shared-memory parallelism, but I have not had time to investigate them. Array bound and pointer checking is perhaps the most useful, followed by uses of uninitialised data and similar errors, and (in scientific codes) trapping of arithmetic errors. All are rare in Fortran, and essentially impossible in

C and C++. There is nothing available for logical errors, so you have no option but to include your own checks.

The remarks about C and C++ need clarifying. There **are** some compiler options and tools to do such checks, but they generally pick up only the most extreme and obvious errors, such as using an address outside both the stack and all allocated objects. Unfortunately, most hard-to-locate bugs are more subtle, and are things like accessing the wrong object or breaking the standard and confusing the optimiser. This, in turn, is made an insoluble problem by the standards' ambiguities and inconsistencies.

Data access, pointer validity and array bounds is perhaps the worst area, but is too complicated to describe here. However, integer overflow and floating-point errors are also important, caused by the signed/unsigned morass and the C standard's interpretation of IEEE 754.

2.3.9 Compiler Options

See lecture 3 of this course for advice on arithmetic issues and lecture 5 on which versions of the standard to use.

You should almost always enable all warnings and usually select checking for language standard conformance, which makes your code much safer on new systems or even with new compiler version. Always develop with your full target optimisation enabled, both because you have to debug only once and because compilers give better diagnostics as part of the analysis for optimisation.

Run-time *check all* options often run very slowly, and often disable all optimisation; sometimes only by a factor of 3, but sometimes by a factor of more than 30. However, try to test all of your code with such options at least once. These include `nagfor -C=all`, `gfortran -fcheck=all` and `ifort -check all`; there is generally no equivalent for C or C++, unfortunately. Some of the numeric error trapping options are cheap, but others have the same issue.

For GNU, you should use "`gcc/g++ -O3 -Wall -Wextra -pedantic -ftrapv`", preferably also `-std=c99/c++11`, possibly "`-Wshadow -Wcast-qual -Wwrite-strings`" and perhaps `-Wconversion` as well; some experts recommend yet more. `-Wconversion` may trap too many spurious errors with some libraries, so you may need to omit it. They now have a 'sanitizer' to detect some errors fairly efficiently, and you should try `-fsanitize=undefined,address` for debugging. There are also facilities for detecting pointer errors and some others – see the specification for details. "`gcc/g++/gfortran -g -O3`" works properly, so you do **not** need to set `-O0` to use `-g` in order to add symbols.

You should also use other compilers if you have access to them, because different ones have different checking. For Intel, use "`icc/icpc -O3 -debug all -w2 -ansi-alias -ffpe-trap=invalid,zero,overflow`", preferably also `-std=c99/c++11`; for Fortran, use "`ifort -O3 -debug all -warn -ansi-alias -fpe0`", preferably also `-std=c99/c++11`. Sun C/C++ has `-xcheck` for stack overflow, and Intel C/C++ and others do something similar; some have limited pre-initialisation of data. Unfortunately, that is more-or-less all of the checking options available. Note that not all compilers will work with all libraries, so you may not get much choice.

Run-time checking is largely futile in C or C++, because the languages effectively forbid it – in particular, most array bound and pointer checking is specifically forbidden. And, in the current C (and forthcoming C++) standards, that is even more so for arithmetic errors, much as in Java. That is why you have no option but to do it yourself.

Unfortunately, while this is what you *should* do, many libraries are not well-written, and the compiler will have hysterics if you ask for that much checking. In that case, you have no option but to reduce it, but try to not reduce it more than necessary and do so only for the files that include the offending libraries.

Fortran:

Ideally convert old code to Fortran 90 or a later language standard, because it has much better checking than Fortran 77; assumed-shape arrays, explicit interfaces and so on mean that many more errors can be picked up automatically. But, if you cannot, all correct Fortran 77 programs are also correct Fortran 90 ones and will work with no further action. See:

Converting Old To Modern Fortran
`https://www-internal.lsc.phy.cam.ac.uk/nmm1/OldFortran/`

As with C and C++, using multiple compilers is useful, and unexpected warnings often indicate a bug. NAG Fortran has by far the best run-time checking, but you should use at least `nagfor -O3 -gline`, and preferably also `-C=all`; this is not bulletproof, but is very close to it. For GNU, you should use "`gfortran -O3 -Wall -Wextra -pedantic -ftrapv -ffpe-trap=invalid,zero,overflow`", preferably also `gfortran -std=f08 -fcheck=all`. For Intel, you should use `ifort -O3 -warn -fpe0`, and preferably also `-stand=f08 -check all`. You can also use the same options for stack checking as for C and C++, in compilers that have them. Even more than for C and C++, not all compilers will work with all libraries.

Others:

You may use other languages for some tasks, by which I mean Python, Java, Matlab etc. Some errors (e.g. array bounds), are usually trapped, others (e.g. overflow) are usually turned into logical errors. Python is good in this respect, Matlab is not too bad, but Perl and Java are truly horrible; Mathematica is somewhere in between. I have little experience with Excel, XML and so on.

2.3.10 Debuggers etc.

I do not use debuggers much, for a variety of reasons, so cannot recommend any particular ones. Serial debuggers cannot handle MPI or OpenMP programs, and the only proper parallel debuggers are commercial; the current open source ones are usually hopeless. I have used `gdb` on simple OpenMP code and it has worked fairly well, so that may be too negative. Theoretically, serial debuggers can be used on core dumps, but they **far** too often just say "No stack", even in cases where the stack is definitely not corrupted. The GPU course may cover GPU debugging.

- Use debuggers if you find they save you time, but do **not** rely on them doing so.

There are some tools for many kinds of memory problem; Valgrind is a popular one, but is very verbose with lots of false positives, and it will not detect problems within the stack or within structures. You generally need a Python or Perl script to munge its output into a form that you can read. "`gcc/g++ -fsanitize=address,leak`" may be better. There are lots of simpler ones, and it is not hard to write your own; they are a great help in some circumstances and of no use in others.

A C++-specific warning is that it does a lot of memory management, which prevents some problems, but makes others worse. If you get crashes in destructors, it will often mean that you have an unrelated bug, rather than one specifically to do with memory management.

2.3.11 Checked Languages Calling C and Fortran

This includes checked languages like Matlab, Mathematica or Python calling C or Fortran or libraries based on them, such as LAPACK, FFTW, numpy/scipy or MPI. Some simple errors are trapped and diagnosed correctly, but they are usually only very obvious ones. Nastier and more realistic ones often cause the calling language to crash, usually much later (so that its traceback, if any, is completely useless) or even in such a way that you get a `glibc` memory dump. Such bugs are truly foul to find, and are caused by such things as:

Overwriting bugs (obviously)
Returning bad pointers or structures
Getting the use count handling wrong
Calling API functions inappropriately
And so on ....

You can use the above techniques to minimise such failures, preferably by ensuring that they are trapped in a controlled environment, diagosed properly, and so do not happen. This is not a panacaea, but helps.