

# How to Help Programs Debug Themselves

## *Checking and Diagnostics*

Nick Maclaren

Computing Service

**nmm1@cam.ac.uk, ext. 34761**

August 2010

# Summary

This covers the most useful coding **techniques**  
How you can make your code largely **self-checking**

- It's not always possible to use **debuggers**  
Don't always work under **schedulers**, **MPI** etc.  
Can almost always use **these** methods
- They are all suitable for use in **production** code  
Most **research projects** involve **ongoing change**
- You won't use **all** of them in **every** program  
Remember that you have to use your **judgement**

# Inserting Checking

- Lots of **checks** is sign of **competence**  
Check **before use** if cost is not too much  
Will often pick up **unexpected bugs** – **why?**

Function **A** makes object **W**'s value invalid

Function **B** uses **W** and mangles object **X**

Function **C** uses **X** and overflows array **Y**

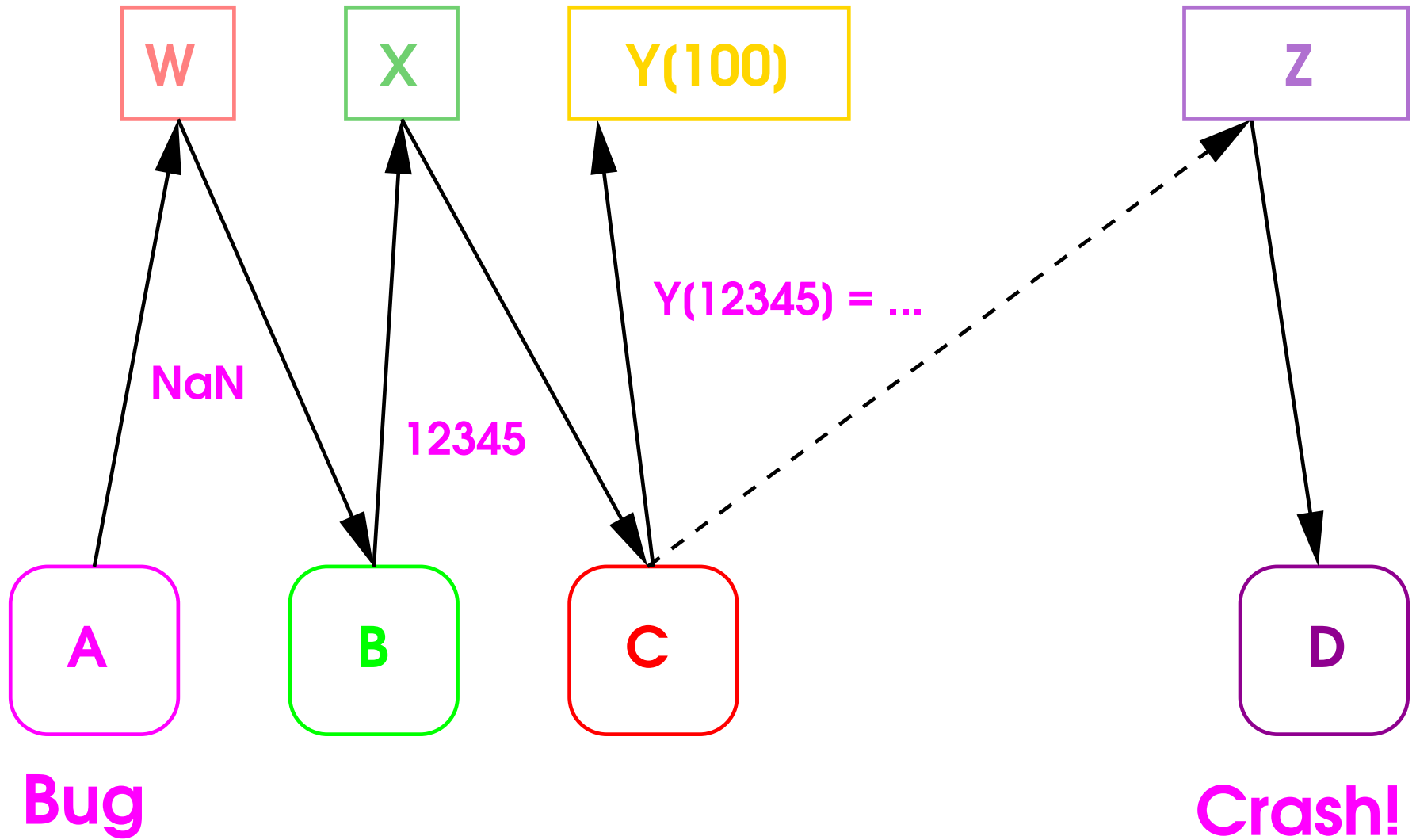
Causes **unrelated** structure **Z** to be trashed

**Much** later function **D** uses **Z** and crashes

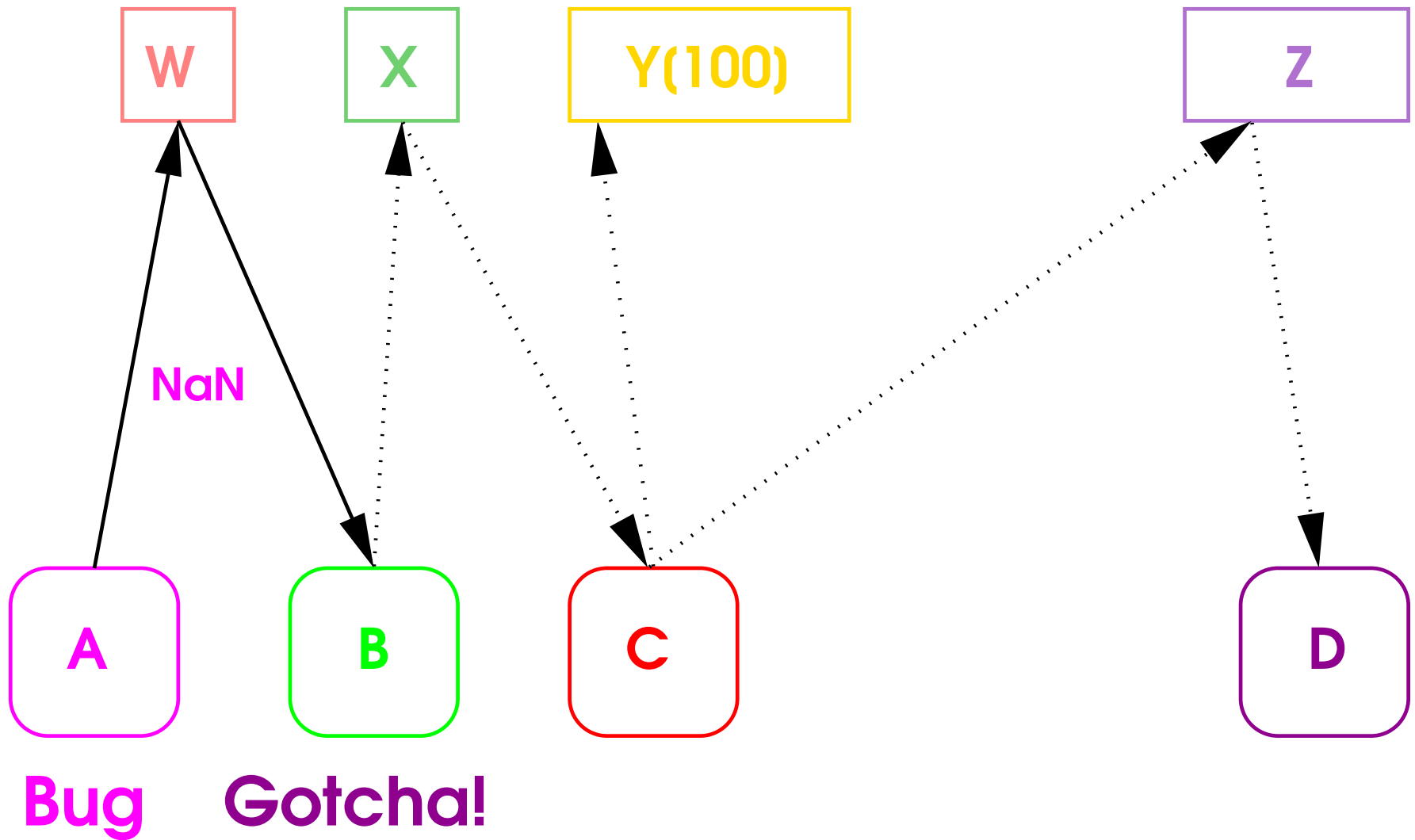
Checking **X** in **B** (or even **C**) catches **AN** error

- Hopefully before too much evidence lost!

# Unchecked Bugs 'Creep'



# Checking and Bug 'Creep'



# Problem Movement

Worst common problems in many codes are:

- Invalid **array indices** and **pointers**
- **Race conditions** and related bugs
- **Code bugs** causing **optimisation** problems

All tend to **disappear** or **change symptom** easily

- **ANY** code change or **compiler option** difference

Including any changes due to the **preprocessor** use

⇒ Minimise **recompilation** for diagnostics

Even **run-time environment** changes can provoke this

# Unpredictable Problems

Race conditions very common in parallel code  
But there are many other fairly common causes

- May be probabilistic – same executable and data  
Symptoms usually predictable, but may move around  
Failure may be rare and occurs well into the run
- Worth doing a lot to avoid such problems arising  
It's not easy to track down bugs statistically

# Warning

You might as well fall flat on your face as lean over too far backward.

James Thurber, “The Bear Who Let It Alone”

- Adding lots of **checking code** takes time  
And **checking code** can itself include **bugs**  
An **optimum** amount to maximise coding **efficiency**

A far higher proportion than most programs have  
But it's still **possible** to have **too much**

- Deciding the level is a matter of **judgement**

# Numeric Errors

Things like **overflow**, **division by zero**, etc.  
Very little is **trapped** and diagnosed nowadays  
Often **only** integer division by zero

- You **must** do any checking yourself  
**Especially** true for **complex** arithmetic

Untrapped **numeric errors** often cause **logic errors**  
Untrapped **logic errors** often cause **overwriting errors**  
Untrapped **overwriting errors** often cause **crashes**  
Or, **much worse**, often cause **nonsense output**

# Consequences

- Lots of **random, simple** checks is best  
Few, **perfect** checks helps **less** with corruption
- Also helps when **making changes** later  
Forget what you were assuming elsewhere?  
Many of **my** errors like that fail on **my** checks
- Check **value ranges, indices/bounds** etc.  
Check any **consistency** properties that you can  
Often simple ones, like **if  $A < 0$  then  $Y > 2$**

# Unit Testing

- Test each **component** before including it  
Sometimes need to test several together  
Much **less confusing** than testing **whole program**
- Remember to test **error handling**, at least roughly  
Helps to avoid **wasted time** with later failures
- Won't pick up all bugs – especially **exceptions**  
**Don't** assume that **tested** means **bug-free**
- Often useful to put **components** in **libraries**  
Can include them in **program** or run a **test** on them

# Test Suites

- Keep your **test data** and the **output** from it  
Can **rerun** and check – known as **regression testing**

Same applies to **unit test** programs and data

When you make a **significant change** to your code

- Rerun appropriate **regression tests** and check
- Automated testing is a **vast** saving in effort  
**Not perfect**, but can save a lot of **manual debugging**

# Object Orientation

Will describe in terms of **object-orientation**

- That is an **approach**, not a **dogma**

The **techniques** are useful far more generally

- **Object**  $\equiv$  **coherent set of data**

May be a **collection** of scalars and arrays

An **object** is often made up of **sub-objects**

- Use this **structure** to keep your code **simple**

E.g. don't **duplicate** code – call the **next level**

And use **recursion** if it matches your **structure**

# Object Identification (1)

It is useful to tag each object with an **identifier**  
**Unix file formats** use 'magic numbers' for this

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

- An **unlikely** value (usually **text** or **integer**)  
Use for **checking** a **pointer** refers to right **type**  
Also very useful when using an interactive debugger

# Object Identification (2)

- Obviously, needs to be in a **known location**  
Simplest to put at very **start of structure**
- Very useful for **C/C++** – less so in **Fortran**  
The **less type safe** the language, the **more useful**

Can be extended – this is usually **overkill**:

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

# Initialisation (1)

- Almost always **initialise explicitly**  
Not just **static data**, but **stack** and **allocated**
  - Don't trust automatic clearing to zero  
**Standards** don't say what most people think they do
  - No, it's **not** too expensive!  
Cost is only linear – use is usually much more
- Use a '**constructor**' to create '**objects**'  
Often does both **allocation** and **initialisation**  
May **initialise only** for language-allocated objects

# Initialisation (2)

- Best to use an **invalid value** if object is unset  
Preferably one that causes a crash if used
- Better than **unpredictably wrong** results  
Initialising to **zero** has its uses, though

Values like **-1.23e300**, **-123456789** etc.  
**IEEE 754 NaN** is useful for this, too

- Useful to **vary value**, to see if bugs move  
Can use different values to flag history

# Object Termination

Don't forget to use an explicit 'destructor'  
Useful hook for **checking** and **tracing**

- Consider **resetting** contents to invalid on **disuse**  
**Last action** before freeing data or returning  
Very rarely done – but can be very useful
- Worthwhile mainly if code uses **pointers**  
Your code may have one **saved** somewhere  
Can be useful in some **non-pointer** codes

# Huge Sparse Arrays

One case where **initialisation dominates**  
**GB–TB arrays** with only (say) **1%** used  
Lazy way of making system do **indexed lookup**

- Absolute **nightmare**, in a great many ways  
Can't use **memory limits** to trap **runaway code**  
And sometimes systems allocate **all** the pages
- Doing it yourself is easy and more flexible  
The caching can be tuned for the application  
Ask for help if you need to do this

# Enabling Diagnostics

- I don't love **preprocessors** much  
Have to **rebuild code** to add diagnostics  
**Many nasty** problems then **move around**
  - Strongly recommend a **run-time** option  
Can select the **diagnostic level** you want
- Can make selectable by **environment variable**  
Or by a **program argument** setting a flag  
Or by whether a suitable **file exists**  
Or by a **command** in the input, or . . .

# Diagnostic Design (1)

- Typically needs a **thoroughness** parameter  
E.g. bounds etc.; all values; cross-checks
- Useful for **run-time option** to set **default**  
And to be able to **override** that in the call
- Exactly the same applies to **tracing**  
May prefer a **separate** option for **tracing**
- Exactly the same applies to object **display**  
Again, you may prefer a **separate** option

## Diagnostic Design (2)

Minimum costs are then **testing** the option  
This is a single, scalar, global, so **efficient**  
Minimal **C/C++** and **Fortran** examples of use are:

```
#include "diag.h"
```

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

```
USE diag
```

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

Can use **C/C++ assert** macro if you like  
But you can do **better** yourself, very **easily**

# Object Display (1)

- All objects should have a **display** primitive  
**Displays** contents so that **you** can see what they are

Merely a convenience – but, oh!, how much!  
Very useful with some **debuggers** – see later

- Typically needs a **level** parameter  
Says how far to **indirect** in structured data  
And how much of **large arrays** to display!
- Or have more than one primitive

# Object Display (2)

- Remember **not** to assume object is correct  
Very often want to **display** broken data

Should work if **pointers** are **null** or **not allocated**  
Check **indices** and **pointers** for being **in range**  
Assume **any** values, whether '**possible**' or not

- May want to call the checker **before** calling it
- Or may call it **from** the checker if that **fails**  
Probably the most generally useful approach

# Object Checking

- All objects should have a **checking** primitive  
Answers “**Is this object vaguely correct?**”  
E.g. values within limits, **self-consistent**
- Tedious to write, but **incredibly** useful  
Can call **automatically**, or insert **manually**  
Can call from many **debuggers**, too
- **Design** objects to be thoroughly **checkable**  
Keep data **clean**, with checkable **constraints**  
Make data **redundant**, maintain **invariants**

# Automatic Use

- Generally call **automatically**, at least **once**  
To ensure that checking code remains correct  
Perhaps at end of **initialisation**, start of **termination**,  
in **error handlers**, and . . .

- Strongly recommend adding a **lot more** calls  
Most important reason for a **run-time option**

High for debugging, lower for production

- Hit a **problem**? Rerun with **checking**

# Manual Use

- This is a very effective way of **debugging**  
It's the way that I generally debug non-trivial code
- An **object** goes bad after **30 minutes** running  
Put **checks** where they will be called **fairly often**
- Now you know **more precisely** where things started  
Find out **why**, add checks for that, and **repeat**
- Can often call **procedures** from **debuggers**  
Calling **checking procedures** saves a **lot** of effort

# Example

- `dposv` is **LAPACK** Cholesky solver

Example of checking arrays before and after:

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)`

$O(n^3)$  calculation –  $O(n^2)$  checking cost

# Invariants

These are things that are **always** true

I.e. from **after initialisation** to **before termination**

- If they are ever **false**, then something is **wrong**  
Perhaps a **logic error** or perhaps **overwriting**

- Every **invariant** can be checked **anywhere**  
Very useful to track down **where** things have failed

They can be **programmatic** – e.g. **array indices**

Or **numeric** – e.g. **values** have certain **limits**

Or things like an **array** must be **positive definite**

# Checking Example

```
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
```

First is **basic check**, can call everywhere

Second is **linear in time**, but more powerful

# Using Invariants

Initialise all of **INDEX** to (say)  $-123456789$

Initialise all of **DATA** to (say)  $-1.0e300$  or NaN

Remember to **reset** the values on **disuse**

- Can now check **valid values** match **USED**

All before **USED** are good, all after are bad

- Will also detect **some** random overwriting

- **Scalar invariants** are generally more useful

Dirt cheap to check, and pick up many mistakes

- **Create, maintain** and **use** invariants when possible

# Argument/Result Checking

Ideally, something like:

```
double operate (double array [ ], int size) {  
    if (size <= 0 || size > MAXARRAY) fail(...);  
    check_array(array,size);  
    . . .  
    result = . . .  
    check_value(result);  
    return result;  
}
```

All **major** procedures should have some of this

# Tracing

- Most common form is tracing **control flow**  
Answers “**How did we get HERE?**”
- Also **events, data flow** and **state changes**  
I.e. “**How did we get into THIS mess?**”

Yes, the **compiler/debugger should** do this  
But providing that is “**Someone Else’s Problem**”

Let’s start with simple function tracing

# Fortran Example

```
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
```

Can add using a [preprocessor](#) (e.g. a [Python](#) script)

# C/C++ Example

```
#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 in same way as for [Fortran](#)

# What to Trace

Usually want **critical** argument and **result** data  
E.g. **identity** of object being **acted upon**

- **Details** are entirely dependent on **requirements**

Might just be an **object id** (e.g. a **index**)

Might include some of the argument **values**

Might include a summary of the **action**

Might include anything else useful . . .

# Controlling Tracing

- Best if **diag\_flag** is a **run-time option**  
Can enable and disable without **recompiling**
- **Tracing** can produce **a lot of output**  
Usually trace to a **file**, not **standard units**
- May need to select **type** and **level**  
E.g. **file tracing**: open/close, all control, all transfers  
Or **state changes**: main ones, all changes, all uses

Remember, **primarily** what saves you most time

# Don't Forget

- May be more than one **return** statement  
Plus reaching end of **procedure**, of course  
Remember **setjmp/longjmp**, **try/catch/throw**
- Can **flush** file each time for safety  
**fflush** in **C/C++**; **FLUSH** in **Fortran**  
Or in **C/C++**: **setvbuf(<file>,NULL,BUFSIZ,\_IOLBF)**

Crashes lose data otherwise – but can be slow  
A case for having another run-time option  
See later for another approach

# What Do We Do Then?

- Could print **entry** and **exit** information  
Do that to a **file**, as can be **voluminous**

Then is easy to **write tool** to display as **tree**  
Or display a **traceback** or **count calls**, or . . .  
There are often **compiler options** to do those  
**As they stand**, they aren't very useful

But **you** can select on **other data** you printed  
Look at just the calls relevant to **specific** problem

# Storing The Data

- Can save **active names** (traceback) in **array**  
A trivial example of using your own **stack**  
This form needs **pushback** when functions **return**
- Now can write your own **traceback** function  
Call when program hits a **problem** or is **signalled**

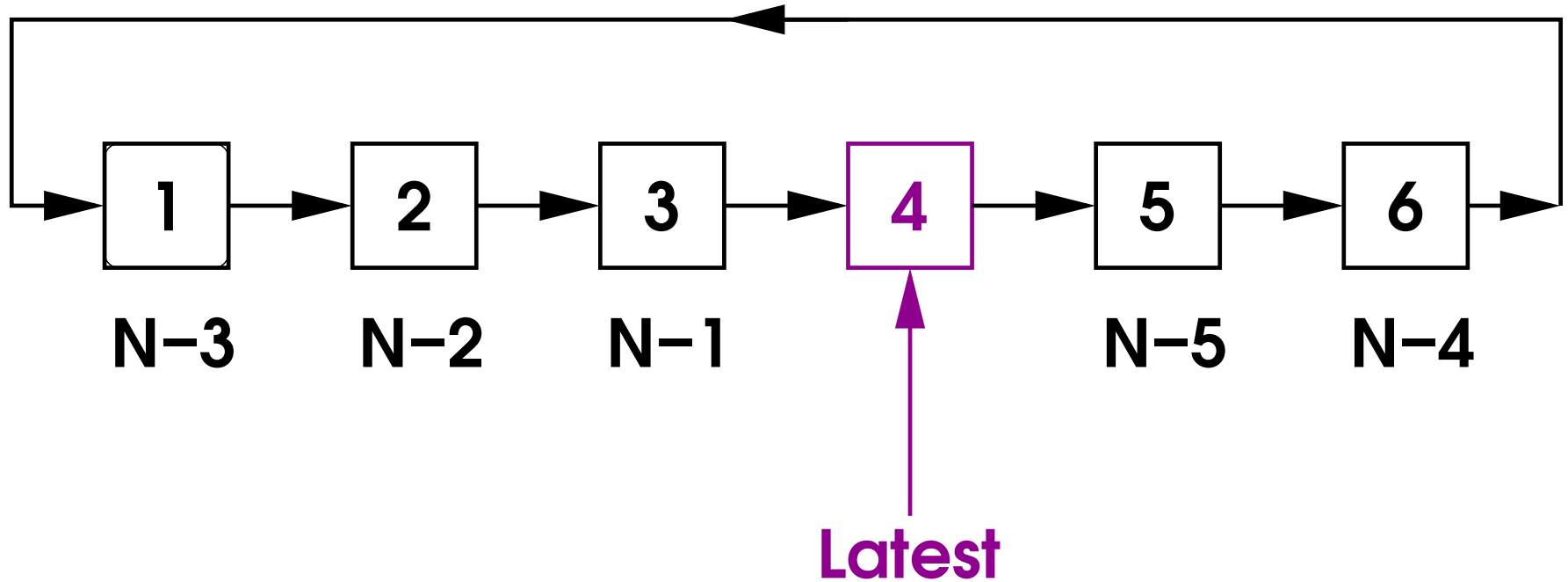
Very few compilers provide this – why not all?

But **needn't** trace returns – just keep last **N** calls  
Gives a **history** of calls, which is also useful

# Circular Trace Buffers

- To do this, use a **circular trace buffer**  
Maintains **last N** calls, or calls and returns
- VERY** useful facility, little taught now  
The most **critical** data, for **fixed memory use**
- Don't forget a function to **display** it
  - Each buffer saves just **one kind** of trace data  
Arbitrary number of buffers – often dozens

# Circular Trace Buffers



# C/C++ Example (1)

```
#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;
}
```

## C/C++ Example (2)

```
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;
    }
}
```

# Event Tracing

- Tracing not restricted to **function calls**  
Can trace any **action**, **event** or similar  
Want to know **order** of **actions** or **events**

Trace **changes** or **accesses** to selected data  
Or changes to **state** – program's or system

Can annotate trace with the **context**  
E.g. **component** responsible for the change

For example, '**man mtrace**' under **Linux**  
Just a random example of use of technique

# Methodologies

These are **methodologies** – not just **tools**  
**Techniques** are much more **general**

- Always think “**Should I automate this?**”  
Answer is often “**infeasible**” or “**it’s not worth it**”  
But sometimes it can save **massive** effort

**TANSTAAFL**

**There Ain’t No Such Thing As A Free Lunch**

**Automation** costs time, but can save much more

# Overheads

- Not all that much on a modern system  
Depends on what the function **actually does**  
**I/O**, **data access** costs; mere **logic** is cheap
- Example above is designed to be very cheap  
If **diag\_flag** is unset, **drops through**  
Most **hardware** will **predict** that correctly
- May be too expensive to do it for all calls
- Can omit from **heavily** used **auxiliaries**  
Will still get **most** of the benefit

# Using From Debuggers

Many **debuggers** can call **program code**

- No use if **data** are completely **corrupt** :-)

Calling **many** functions changes program **state**

But not **checking**, **display** and **tracing** functions

At least if you have coded them right!

- Makes **use** of debugger much more powerful

The **Old Guard** (who? me?) do that **manually**

It's irrelevant – you need the same primitives

# Displaying Data Structures

- A real problem, **however** you do it  
**Scalars** are easy, but **arrays**? And **pointers**?  
How **far down** do you want to **indirect**?  
Or do you want pointer **values** and target **addresses**?

No general solution, and debuggers don't help  
Writing **display functions** is always tedious

- You can implement your own **printf**-imitation  
Painful in **Fortran** – one call per argument

# Tracing State

I said that **global** state is horrible

There's lots of it in **C**, **C++**, **POSIX**

**Big** problem if wrong at **component** boundary

Try tracing **state** and **component** changes

Best method of tracking this issue down

**Biggest problem** is instrumenting your code

It's trivial if you have **encapsulated** the actions

# Handling Crashes

Often lose diagnostic output after crashes

- Can **trap** most **signals** and close files

**Good libraries** do that by default

Need a run-time option to get a **dump**, of course

Can also call **traceback** procedures in such a **handler**

Or can print out **history** or **objects**, or ...

- That may not work – but there was a **crash** anyway

**Details** are **repulsive**, but don't need to know them

# Using In Test Suites

Often have **suites of data** used for **testing**  
“**Regression testing**” checks **old data** still works

- But a **lot** of bugs get through
- And what when changes are to **output**?  
Can't check results **automatically** any longer

Using good **checking primitives** helps a **lot**  
Runs **slower**, but **more confidence** in result

- Still won't check answers are **right**

# Using Tracing Hooks

- Tracing **hooks** allow **use-counting** or **timing**  
Can select with just a **run-time option**

- Good place to insert **checking** code

- Or can **call back** to debugger

E.g. by calling **trapped function** or **failing**

Can enable when context is appropriate

**1513th** time **fred**  $\Rightarrow$  **joe**  $\Rightarrow$  **alf**

# Long-Running Problems

- Most systems have a fairly small **job time limit**  
For **RAS**, **maintenance** etc. – often **24 hours**
- A **program** may write its **current state** to a file  
[ This is often called **checkpointing** ]
- The **job** may resubmit another as it finishes  
It starts by **restoring** from the **checkpoint**
- Best to use **alternate** checkpoint files  
In case of a **crash** while it is being **written**  
Email **scientific-computing@ucs** if you need help here

# Make

**make** is a tool for managing program rebuilding  
Recompiles all changed sources and only those  
Many equivalent programs and derivatives

- Essential when file structure gets complicated  
Saves a lot of build time – and reduces mistakes!  
For a few files, a simple recompilation script is OK

Not covered in this course – but recommended

- Golden rule of makefiles: **KISS**  
Complexity causes non-portability and bugs

# Source/version/revision Control

CVS, subversion and a zillion others

Manage source code updates and variant versions

Usually allow archiving, roll-back etc.

Main alternative is disciplined file management

E.g. taking snapshots of source at intervals

- But they are essential if several developers  
Manual coordination is extremely error prone

I don't like these, for a variety of reasons

Again, not covered in this course

# Integrated Development Environments

Very often little more than **snake oil**  
More kindly, a **GUI toolkit** for development

Often include **version control** (**CVS** etc.)  
Plus **integrated make** equivalent

- Use them if you need to or like them
- But they **WON'T** help with **debugging**

Best ones provide **regression testing** etc.

- Nothing that you can't do with **scripts**

# Syntax-Aware Editors

Popular bandwagon in **1980s** – still here

**Near-total waste** of time and money

Who spends **50%** of time fixing **syntax errors**?

Users on **first** programming course, that's who!

And, of course, senior **executives** and similar

- **Experienced** programmers spend **≈1%**

Also make certain classes of error more common

What we **need** is **run-time checking**

- Cases of **undefined** (invalid) behaviour
- And, much worse, **logical errors**

# Run-Time Checking

Some compilers and debuggers do a little

There may also be special tools

Intel has some tools for parallelism

Array bound & pointer checking is useful

Also uses of uninitialised data etc.

So is trapping of arithmetic errors

- All rare in Fortran, impossible in C/C++

Nothing available for logical errors

- No option but to include your own

# C/C++ Compiler Options (1)

Use at least `gcc -Wall -Wextra -pedantic`  
Possibly `-Wconversion -Wshadow -Wcast-qual`  
`-Wwrite-strings`

And some experts recommend yet more . . .

- `gcc -g -O3` works properly!

You do **not** need to set `-O0` to use `-g`

- Also use **other compilers** if you have them  
Different ones have **different checking**

# C/C++ Compiler Options (2)

Sun has `-xcheck` for stack overflow

Intel and others do something similar

Some have `limited` pre-initialisation

- That's `more-or-less` it, unfortunately

Run-time checking is futile in C or C++

Array bound / pointer checking? Get real!

Arithmetic errors are worse (as in Java)

# Fortran Compiler Options (1)

Ideally, convert old code to Fortran 90 or later

[http://www-uxsup.csx.cam.ac.uk/courses/...](http://www-uxsup.csx.cam.ac.uk/courses/.../OldFortran)  
[.../OldFortran](http://www-uxsup.csx.cam.ac.uk/courses/.../OldFortran)

Much better checking than Fortran 77

Assumed-shape arrays, explicit interfaces etc.

- Using multiple compilers still useful
- Unexpected warnings often indicate a bug

# Fortran Compiler Options (2)

**NAG Fortran** by far best run-time checking

Use at least **f90 -C=all** and maybe **-gline**

- Not **bulletproof**, but fairly close to it

For **gfortran** use similar options to **gcc**

For **Intel**, use **-warn -check all**

Can use **C/C++** options for stack checking

# Other Languages

I mean **Python**, **Java**, **Matlab** etc.

Some errors (e.g. **array bounds**) usually trapped  
Others (e.g. **arithmetic**) turned into **logical** errors

**Python** is good, **Matlab** not too bad

**Perl** and **Java** are truly horrible

**Mathematica** is somewhere in between

I have little experience with **Excel**, **XML** etc.

# Debuggers

I don't use these much, for a variety of reasons  
So can't recommend any particular ones

Serial debuggers can't handle MPI or OpenMP  
Only proper parallel debuggers are commercial  
Except possibly gdb etc. on OpenMP code

Theoretically, can be used on core dumps  
But far too often just say “No stack”

Use them if you find they save you time

- But don't rely on them doing so

# Memory leaks etc.

**Valgrind** etc. for many kinds of **memory problem**  
Very **verbose** – external libraries give **false positives!**  
Checking **stack** or **structures** is under development  
Need **Python** or **Perl** to munge output

Lots of simpler ones – not hard to write your own  
Great help sometimes – no use in others

**C++** does a lot of **memory management**  
**Prevents** some problems, makes others **worse**

- Crashes in **destructors** often mean **unrelated** bug

# Checked Languages $\Rightarrow$ C etc.

For example, **Matlab** calling **Fortran**, **C** or **MPI**  
**Some** simple errors trapped and diagnosed correctly  
Nasty ones often cause **calling language** to crash  
Usually **much** later or even a **glibc** memory dump

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

$\Rightarrow$  Use the above techniques to minimise these