

declared bounds. An incorrectly computed array index (i.e., subscript) is the immediate cause. This type of bug occurs in peoples' code very frequently. To discover instances of array bounds violations you can compile your code with a special bounds checking option. For the SGI the different compilations commands are:

```
f77/f90 -DEBUG:subscript_check=ON ...  
cc -DEBUG:subscript_check=ON ...
```

On the Cray the appropriate commands are:

```
cf77 -Wf"-Rb" ...  
f90 -Rb ...  
cc -h bounds ...
```

Upon execution, the system will emit messages on standard error about array bounds violations for each such encounter in the execution sequence. These messages will name the offending array as well as the routine where the violation took place.

After you're through with array bounds checking don't forget to recompile your code without this option. Executables with this option on run very slow.

Final Comments

I've commented on a small number of easy steps you can take to avoid the pain (and embarassment...) of pesky bugs. I urge you to adopt them in your programming work. You can find more information on these and related topics in the following man pages: `DEBUG_group` (SGI), `lint`, `cflint`, and `ftnlint`.

Spiros Vellas
Sr. Analyst, CIS

within the code (when available as a language feature). This is not recommended as a standard practice. The reason you may want to do that sometimes is to discover possible ULV existence. On the SGI you can specify static allocation (Fortran) at compilation by the sequence:

```
f77 -static ...  
f90 -static ...
```

On the Cray the specification for static allocation (Fortran) is:

```
cf77 -Wf"-a static" ...  
f90 -ev ...
```

If your program executes "fine" when static allocation is applied, while it crashes or yields wrong results without it, then there is a strong chance your code has ULVs. Static allocation for local variables is fine as long as you explicitly initialize them. Its defects are generation of bulkier executable files and prevention of parallelization. Static allocation is also an unavoidable expedient when porting code from compilers which, by default, allocate statically and initialize to zero at each routine invocation. This approach should only be adopted on a temporary basis.

High Optimization Levels

Specification of high optimization levels (e.g., -O3 on the SGI) carries some risk in obtaining incorrect results. This does not happen frequently, but it does happen nevertheless. Hence, from time to time, you would be well advised to double check results you obtained with high optimization levels with those you obtained with lower ones. The -O2 and lower optimization levels are quite safe on the SGI. Your concern for incorrect results should be especially heightened when parallelization options are combined with high levels of scalar optimizations. Examples of this would be, for example, on the SGI the compilation:

```
f77/f90 -O3 -pfa ...  
cc -O3 -pca ...
```

The analogous situation on the Cray would be compilations such as

```
cf77 -O task3 -O vector3 -O scalar3 ...  
f90 -O task3,vector3,scalar3 ...  
cc -h task3 -h vector3 -h scalar3 ...
```

where high levels of parallelization and vector/scalar optimizations are combined.

Array Bounds Checking

Array bounds checking should be another routine measure to rid your code of hidden bugs. Find out, that is, whether arrays are being accessed within the

new memory space is allocated it contains undefined (i.e., garbage) values. Hence, the need for explicit initialization. This holds true even for variables that are declared in the main routine only. The only exceptions to dynamic allocation are local variables that appear in: Fortran COMMON, DATA and SAVE (conceptually, at least) statements, Fortran routines compiled with the "static" option on, and in C "static" declarations.

ULVs, that is, local variables that participate in expressions with undetermined (i.e., possibly garbage) values, account for a big majority of all bugs. A ULV by participating in integer, pointer, or floating-point expressions can easily lead to erroneous expression evaluations, that in turn can lead to erroneous DO-loop limits, erroneous array indices, etc. A huge number of bugs have ULVs as their root cause, even though at first sight this may not seem to be the case.

ULVs are a well recognized problem in code development. Most compilers offer options for their detection. This feature, however, is not totally effective. Hence, again, the need for explicit initialization of local variables.

On the SGI Power Challenge the "-DEBUG:" compiler (f77/f90 and cc/CC) option provides one way to help discover ULV's. One sequence of commands to accomplish that is,

```
setenv TRAP_FPE "ALL=NAN,ABORT,TRACE"
f77/f90 -DEBUG:trap_uninitialized=ON files.f -lfpe
cc -DEBUG:trap_uninitialized=ON files.c -lfpe
```

On the Cray the equivalent sequence of commands is,

```
f90 -ei ...
cf77 -Wf"-ei" ...
cc -h indef ...
```

If your program "works" when compiled without the above options, but crashes when they are applied, then at least one of the bugs in it is an ULV. Also, if the program crashes at one place without these options, but crashes at another when they are applied, then again one of the bugs is likely to be an ULV. Sometimes, especially when integer ULVs are involved, the above trap will not be effective. You are strongly encouraged to try these options, or similar options, on more than one system. This, many times, enhances your chances in catching pesky code bugs.

On both Cray and SGI machines f90 issues at compilation, among others, the diagnostic message, "Variable X may be used before it is assigned". Do not ignore such messages. They indicate the presence of ULVs. To obtain a comprehensive analysis of your source code for potentially risky constructs, you can run cflint (or ftnlint on the SGI), a Fortran source diagnostic tool. The generated report from cflint will list each detection of an ULV by issuing the statement,

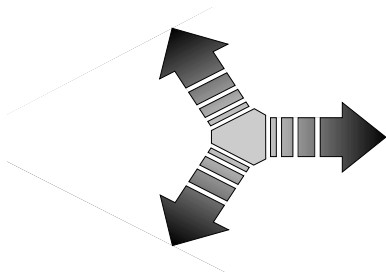
```
Local variable "X" may be used before it is defined
```

To obtain the report (called prog.cflint or prog.ftnlint below) from cflint and ftnlint carry out the following sequence:

```
f90 -Ca ... prog.f (Cray)           f90 -cif ... prog.f (SGI)
cflint prog.f > prog.cflint         ftnlint prog,f > prog.ftnlint
```

(The appropriate cf77 option on the Cray is cf77 -Wf"-ca" ...)

You can force static memory allocation for local variables by applying the appropriate allocation option in the compilation command or by appropriate declaration



TEXAS A&M UNIVERSITY SUPERCOMPUTING FACILITY

203 Teague
College Station, TX 77843-3363
(409) 845-4139
help@sc.tamu.edu
<http://sc.tamu.edu/>

BASIC FIRST STEPS FOR BUG-FREE CODE

If you develop code (and that includes making even minor changes to one), a few basic steps on your part will help a lot to avoid many common bugs that can stay undetected for a long time. The list shown below summarizes some basic measures and ideas. Here I discuss only items one through five, leaving the rest for a visit to our Help Desk or a short course on debugging.

- 1) Initialize explicitly all local variables, scalar or array: integer, floating-point, character, strings, types (f90), structures, etc.
- 2) Use specialized code analysis tools (e.g. lint, cflint, ftntlnt) to identify and locate potential problem areas (e.g., uninitialized local variables).
- 3) Avoid "static" allocation for local variables unless it is carried out as a measure to ferret out bugs, or as temporary expedient when porting code from one system to another.
- 4) Confirm, as frequently as practicable, that results obtained with high optimization levels agree with those of lower ones.
- 5) Perform frequent array bounds checking when making code changes.
- 6) Avoid making calls with aliased arguments, call `subx(a,x,a,...)`, being an example.
- 7) Avoid making equality comparisons between floating-point variables.
- 8) Be aware that numeric data in binary floating-point form when transferred (and, therefore, transformed) to ASCII (text) formatted files lose accuracy. Subsequent use of these ASCII files as input may easily accentuate roundoff effects to unacceptable levels.

Uninitialized Local Variables (ULVs): Items 1-3

ALWAYS, ALWAYS, ... INITIALIZE EXPLICITLY ALL LOCAL VARIABLES TO APPROPRIATE VALUES. DO NOT DEPEND ON THE COMPILER FOR SUCH ACTION.

A local variable, scalar or array, is one whose scope is limited within the boundaries of the routine (function, subroutine, module, etc.) it is declared. Memory storage allocation for such variables can be either static or dynamic. By default, it is dynamic. In such a case, when a routine is invoked, its local variables are allocated new space at a (potentially) different memory address at each call. This space is returned to the system when the procedure returns to the caller. Each time