CS 536 — Spring 1997

Purify

What is Purify?

Purify is a commercial software package that is used to quickly identify and eliminate common run-time errors and memory leaks. In "Purified" code, all run-time memory references are checked for the following types of mistakes:

- reading or writing beyond the bounds of an array
- reading from or writing to deleted memory
- deleting memory multiple times
- reading or using uninitialized memory
- dereferencing incorrect or illegal pointers
- dynamically allocated memory that is never deleted (memory leaks)

Unlike gdb, most of Purify's error-checking is automatic. You simply instrument a C++ program to include Purify's run-time checks. A window summarizing the errors that have been discovered is automatically created as the compiled program executes.

How to Use Purify

Purify is available for use on all Sun workstations. It is in `/s/purify/bin/purify`. (You may want to add `/s/purify/bin` to your search path, in `.cshrc.local`). To add Purify's capabilities to a program, just prepend `purify` (or `/s/purify/bin/purify`) to the program step that builds the executable file for your program. Thus if you enter

```
g++ -g test.cc
```

to translate `test.cc` and create `a.out`, you'd simply run

```
purify g++ -g test.cc
```

to translate `test.cc` and create a version of `a.out` that contains Purify's run-time checks. (It's wise to include the "`-g`" option so that Purify has access to the program's source text.)

You can now run the executable file. The program will produce the same output as usual. Purify will open a window summarizing the results of its tests. If the first line in the window shows "0 errors." you're OK. Otherwise, the listing it produces will detail the problems it has found.

An Example

Consider the following simple program, found in `~cs536-1/public/purify/hello.cc`

```
#include <stream.h>

static char *helloWorld = "Hello, World";

int main()
{
   char *mystr = new char[strlen(helloWorld)];

   strncpy(mystr, helloWorld, 12);
   cout << mystr << "\n";
}
```

Copy this program into your directory and compile it using g++. It will appear to execute properly, producing the expected output. However, errors lurk in the shadows! Now compile it using Purify (and the trusty `-g` option):

```
purify g++ -g hello.cc
```

When we execute `a.out`, the window Purify creates shows that an error has been detected. The line titled **"ABR: Array bounds read"** suggests an error involving an illegal access beyond the end of an array. Clicking on the triangle to the left of the message provides additional information.

13 bytes were read from a block of heap memory containing 12 bytes. (`malloc` is called indirectly from `new`.) Clicking on the triangle to the left of `main` will show us just where the illegal access was. At the print statement, `mystr`, a character array, exceeded its upper bound. Purify also shows us where the array was allocated—at the call to `new`. A moment's reflection tells us what went wrong. We forgot to include an extra character for the null that terminates the string. You can edit the program to allocate that extra character, and rerun the test, again using Purify.

Surprisingly, another error has appeared! We now have a **"UMR: Uninitialized Memory Read"** error. Clicking the triangle will show us where. Again, the error is at that print statement. 13 bytes of `mystr` were read, but not all of the bytes were initialized. Looking at where `mystr` got its value shows the problem—we only copied 12 bytes in the `strncpy` call. We need to include the terminating null character. Once we fix that error Purify will tell us that no new errors are detected by it. (The report of 13 bytes of "memory leak" is accurate, but not important as all memory is automatically freed once `main` completes.)

Using Purify in Makefiles

Probably the easiest way to use Purify in debugging your class projects is to edit the `Makefile` you use to create a "Purified" version of your program as well as an ordinary executable. To do this, find the lines in your `Makefile` that define the creation of the final executable file. This is normally a call to g++ that contains a list of all the "`.o`" files in your program. For example, in project 3 the lines in the `Makefile` are

```
proj3: main.o scanner.o parser.o ast.o unparse.o templates.o
     g++  -o  proj3   main.o  scanner.o  parser.o  ast.o  unparse.o   \
         templates.o -ll
```

We'll add lines to the `Makefile` to make a Purified version of the executable file. We'll add a suffix "`.pur`" to remind us this executable contains Purify's checking code. In the above, we'll change `proj3` to `proj3.pur` in the copied lines. We'll also prepend the call to `purify` to the immediate left of the call to `g++`. Thus we have

```
proj3.pur: main.o scanner.o parser.o ast.o unparse.o templates.o
     purify  g++  -o  proj3.pur  main.o  scanner.o  parser.o  ast.o \
         unparse.o templates.o -ll
```

If there are `Makefile` rules to build and test a program, you might want to add additional rules to build and test a Purified version of the program. Thus we might add to the following to the `Makefile` for project 3

```
puretest: proj3.pur
      proj3.pur testfile
```

Take a look at the directory in `~cs536-1/public/purify/proj3`. It is a version of the project 3 CSX-lite parser. The `Makefile` in this directory has been extended to include Purified versions (as outlined above). Copy the contents of this directory to a test directory and enter

```
make test
```

When the parser for CSX is built and tested, you'll see a bug—all identifiers are unparsed as `aa`! gdb isn't of much immediate help, since the program doesn't dump a `core` file that points to an identifiable erroneous line. Maybe Purify can help. Enter

```
make puretest
```

Now when the parser is tested Purify runs its tests. Sure enough, it finds run-time errors (18 of them!) involving use of an uninitialized value. The messages it produces should be sufficient to quickly find (and correct) the bug in this test program. Try this and see how quickly you find the bug!