# Memory Analysis tools

# The Necessity

- Application behaviour:
  - Crashes intermittently
  - Uses too much memory
  - Runs too slowly
  - Isn’t well tested
  - Is about to ship
- You need something
  - See what your code is really doing
  - Help spend less time finding bugs
  - Develop fast, reliable code

# Type of Memory Errors

- ABR/ABW – Array bounds read/write
- FMR/FMW – Freed memory read/write
- UMR – Uninitialized memory read
- MSE – Memory Segment Error
- MLK – Memory Leak
- NPR/NRW – Null Pointer Read
- FMM – Freeing Mismatched Memory

# Software Quality Management

- **IBM-Rational Products:**
  - A set of runtime analysis tools
  - Increase code quality
- **Purify**
  - Detecting Runtime Errors
  - Automatically pinpoints hard-to-find bugs
  - Profiling .NET Managed Code (Memory Profiling)
- Quantify
  - Application profiler
  - Highlights performance bottlenecks
- PureCoverage
  - Source code coverage analysis
  - Helps avoid shipping untested code

# Features and benefits

- **Who can use it?**
  - For everybody
  - Unix, Windows, C, C++, Java, .NET
  - Developers and testers
- **Source Code required?**
  - Monitors components with no source code
- **Interesting Features:**
  - Rich command line interface and batch mode for automation
  - Integrations with Rational Robot, Rational ClearQuest, Rational ClearCase, and Microsoft Visual Studio.NET

**Current Version: PurifyPlus**

# What about static analysis tools

- Static analysis tools are a great complement to Purify
  - Find errors that you don't exercise in test cases
  - Find richer semantic errors, e.g. type safety
  - Find potential errors if calling patterns change
  - Analyze code sections before you have a working executable

# What about static analysis tools

- Static analysis tools have limitations
  - Only find errors in the code you have source for
    - Not in libraries that you pass bad data too or that are buggy
  - Can miss errors whose cause & effect are distant in time & space
    - Or bury you in “possible” errors
  - Can take a long time to run

# How does Purify work?

- Take your compiled application apart
- Find interesting places to insert probes
- Put your application back together again
- Run it and play

- Detecting memory leaks
  - GC-like algorithm
    Maintain list of all allocated blocks in all heaps
    And call chain of allocator
  - On demand, scan memory for all in-scope pointers
  - Starting from anchors –stack, statics, registers
  - Any block not pointed to is “leaked”(MLK/PLK)
    All other blocks are “memory in use”(MIU)

# Technique

- Track state of each byte in process address space

  -Red = logically unallocated to app

  -Yellow = allocated (malloc/new) but not written to yet (uninitialized)

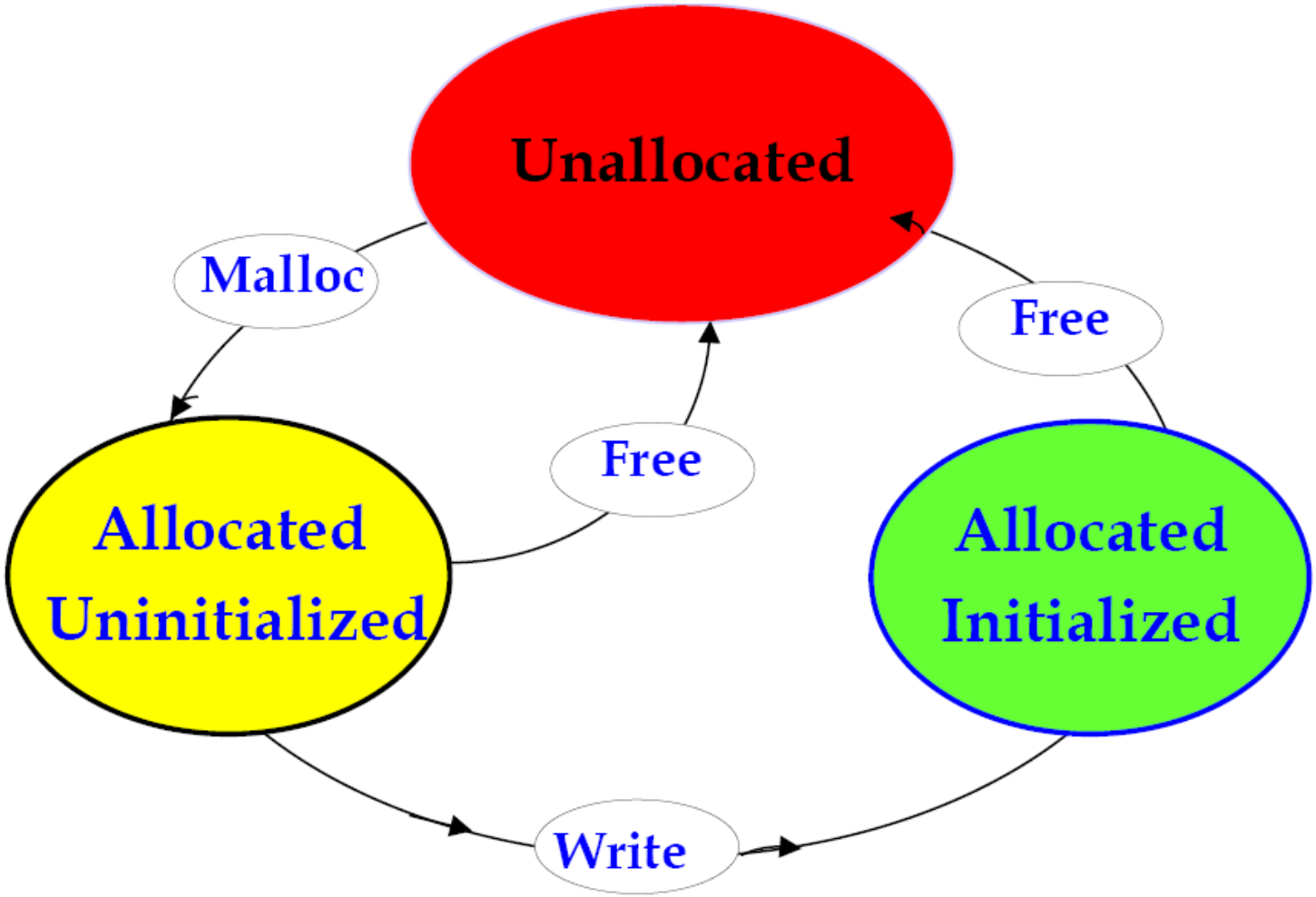
  -Green = allocated and initialized

  – Add red guard zone to the ends of allocated blocks

    -To catch buffer overrun errors

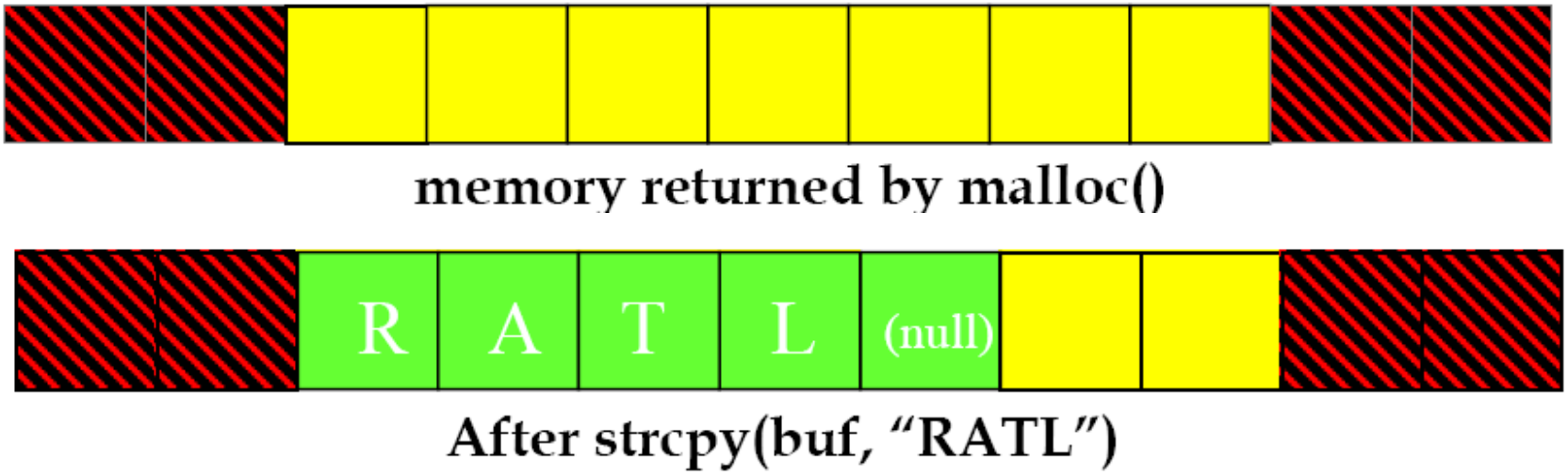
  – Monitor every read and write instruction

# Purify's memory state tracking



# Purify's array bounds detection

- Inserts guard zones around each block allocated. (Guard zones are colored red. A read or write to red memory triggers an array bounds violation)




**Uninitialized read/write memory triggers ???**

# Valgrind

# Case 1:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
        int i;
        int *a = malloc(sizeof(int) * 10);
        if (!a) return -1; /*malloc failed*/

        for (i = 0; i < 11; i++)
            a[i] = i;

        free(a);
        return 0;
}
```

- $ gcc -Wall -pedantic -g example1.c -o example
- $ valgrind ./example ==23779== Memcheck, a memory error detector ==23779== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al. ==23779== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info ==23779== Command: ./example ==23779== ==23779==

Invalid write of size 4 ==23779== at 0x400548: main (example1.c:9) ==23779== Address 0x4c30068 is 0 bytes after a block of size 40 alloc'd ==23779== at 0x4A05E46: malloc (vg_replace_malloc.c:195) ==23779== by 0x40051C: main (example1.c:6) ==23779== ==23779== ==23779==

HEAP SUMMARY: ==23779== in use at exit: 0 bytes in 0 blocks ==23779== total heap usage: 1 allocs, 1 frees, 40 bytes allocated ==23779== ==23779== All heap blocks were freed -- no leaks are possible ==23779== ==23779== For counts of detected and suppressed errors, rerun with: -v ==23779== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)

# Case 2:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
        int i;
        int a[10];
        for (i = 0; i < 9; i++)
            a[i] = i;

        for (i = 0; i < 10; i++)
            printf("%d ", a[i]);

        printf("\n");
        return 0;
}
```

- ==24599== Conditional jump or move depends on uninitialised value(s)
- ==24599== at 0x33A8648196: vfprintf (in /lib64/libc-2.13.so) ==24599== by 0x33A864FB59: printf (in /lib64/libc-2.13.so)
- ==24599== by 0x400567: main (example2.c:11)
- ==24599== ==24599== Use of uninitialised value of size 8 ==24599== at 0x33A864484B: _itoa_word (in /lib64/libc-2.13.so) ==24599== by 0x33A8646D50: vfprintf (in /lib64/libc-2.13.so) ==24599== by 0x33A864FB59: printf (in /lib64/libc-2.13.so) ==24599== by 0x400567: main (example2.c:11)

- ==24599== ==24599== Conditional jump or move depends on uninitialised value(s) ==24599== at 0x33A8644855: _itoa_word (in /lib64/libc-2.13.so) ==24599== by 0x33A8646D50: vfprintf (in /lib64/libc-2.13.so) ==24599== by 0x33A864FB59: printf (in /lib64/libc-2.13.so) ==24599== by 0x400567: main (example2.c:11) ==24599==

0 1 2 3 4 5 6 7 8 7

# Case 3:

```
#include <stdio.h>
#include <stdlib.h>

int main(){
        int i; int *a;
        for (i=0; i < 10; i++){
          a = malloc(sizeof(int) * 100);
        }
        free(a);
        return 0;
}
```

# What valgrind is not:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
        int i;
        int x = 0;
        int a[10];
        for (i = 0; i < 11; i++)
            a[i] = i;
        printf("x is %d\n", x);
        return 0;
}
```

# What valgrind is not:

```
#include <stdio.h> #include <stdlib.h>

int main(){
    char *str = malloc(10);
    gets(str);
    printf("%s\n",str);
    return 0;
}
```