Purify at a Glance

Here's a quick overview of the most common Purify error messages. If you want to know more about these errors, or about other errors not listed here, you can run `man purify` from any of the Sweet Hall machines.

ABR (Array bounds read)
ABW (Array bounds write)

This error indicates that you've gone past the end of an array (whether allocated directly on the stack — e.g., char *words[MAX_WORDS] — or on the heap with malloc). For example, trying to do the following will result in an ABR:

```
int *arr, i, j;
arr = malloc(sizeof(int) * 10);
for(i = 0; i < 10; i++)
   arr[i] = i;
j = arr[11];  /* ABR */
```

As you might expect, an ABW occurs when you try to write past the end of an array:

```
/* continued from above */
arr[11] = 23;
```

Note, however, that you don't actually have to have an array per se to get this error. If you go past the end of any block of memory you'll have this problem (think of non-arrays as arrays of size one). For example, assume that short ints are 2 bytes and longs are 4 bytes:

```
long *num;
num = malloc(sizeof(short int));
*num = 65537;
```

This code only allocates two bytes, but then tries to use four. So, Purify will report an ABW here with something like:

```
ABW: Array bounds write:
  * This is occurring while in:
       main           [ccZTDiy_1.o]
       _start         [crt1.o]
  * Writing 4 bytes to 0x93750 in the heap (2 bytes at 0x93752 illegal).
  * Address 0x93750 is at the beginning of a malloc'd block of 2 bytes.
  * This block was allocated from:
       malloc         [rtlib.o]
       main           [ccZTDiy_1.o]
       _start         [crt1.o]
```

ABR and ABW will probably be the most common Purify error you'll see in this class (with MLK a close third).

MLK (Memory leak)

One of the most useful features of Purify is its ability to track your dynamically-allocated memory and let you know when your program leaks it. Unfortunately, Purify has no way of knowing where you meant to free the memory (or should free the memory), so the best it can do is show you where you allocated it.

To continue from the short example with the ABW, if the memory is not freed before the program ends, Purify will report the following:

```
MLK: 2 bytes leaked at 0x93750
  * This memory was allocated from:
        malloc          [rtlib.o]
        main            [ccZTDiy_1.o]
        _start          [crt1.o]

Purify Heap Analysis (combining suppressed and unsuppressed blocks)
                         Blocks        Bytes
              Leaked          1            2
  Potentially Leaked          0            0
              In-Use          0            0
  ----------------------------------------
     Total Allocated          1            2
```

In addition to specifying each leak, Purify also gives an overall analysis at the end of the program. If you forget to fclose() a file that you've opened, you'll probably get 4100 bytes (per instance) on the "potentially leaked" line. In-Use refers to memory allocated in the global data segment for things like randomizing (don't worry about what that means if you don't know). **You do not need to worry about anything but leaked memory on the first line.** However, if you can get rid of potentially leaked memory by closing open files, please do so.

FMR (Free memory read)

You're using memory after you've freed it. The most common cause of this is when two parts of the code share a pointer to the same string, and then one part frees it, while the other part of the code expects the string to remain in place. For example:

```
void Foo(const char *string)
{
  char *s = (char *) malloc(strlen(string) + 1);
  strcpy(s, string);
  Bar(s);
  printf("%s\n", s);  /* FMR */
  free(s);            /* FUM */
}

void Bar(char *x)
{
  free(x);
}
```

Watch out for FMRs when you have data structures being passed around your program. It's very easy to free it too soon.

FUM (Freeing unallocated memory)

This error is directly related to FMR. It simply means that you are freeing memory that has already been freed. So, in the example for FMR, the line free(s) in Foo will give this error.

FNH (Freeing non-heap memory)

Hopefully everyone remembers from 106 that you should always free memory you dynamically allocate, and that you should never try to free memory you didn't allocate. Stack memory should never be freed:

```
char buf[1000];
strcpy(buf, "Hello world");
free(buf);  /* FNH */
```

In this case, buf is not on the heap, and will be deallocated when the containing function ends.

NPW (Null pointer write)
NPR (Null pointer read)

NULL can never be a valid memory address. If you try to read it or write it, you'll get this error. Null pointer reads/writes are common causes of segmentation faults.

UMR (Uninitialized memory read)

This error comes up if you try to read a variable's value without setting it. For example, not giving counter variables an initial value of 0 is a common cause:

```
int count, ch;

while((ch = fgetc(fp)) != EOF)
  count++;
```

The first time count is incremented in this loop, it has no initial value. It's not guaranteed to be zero (although often it will be), so the code may or may not run correctly. The fix is simple, of course: set count = 0 before the loop.

COR (Fatal core dump)

When your program has a segmentation fault or bus error, the Purify log will also contain this message. It shows the exact line that caused the crash.