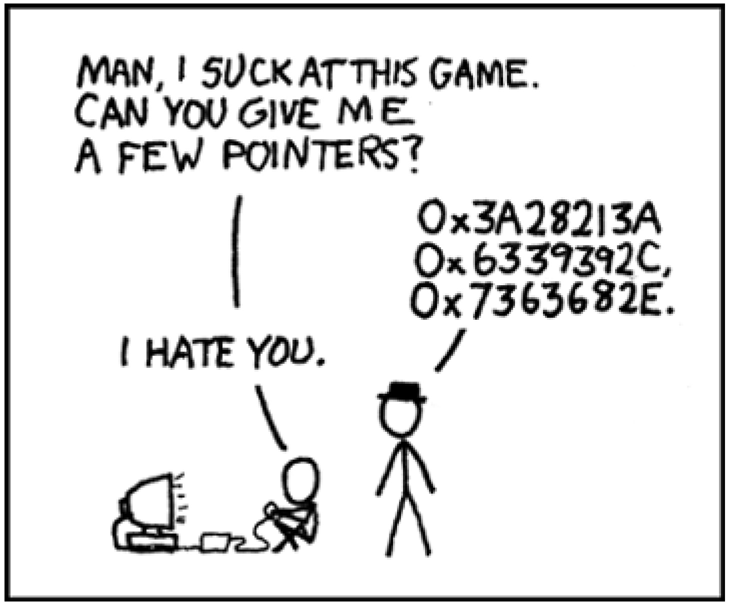
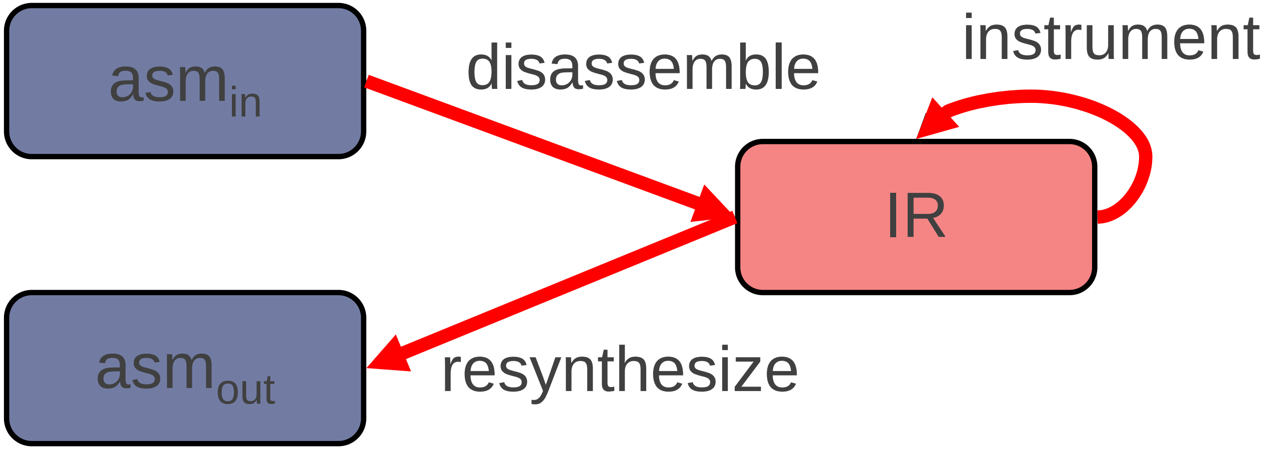
LS: Valgrind, Memory Leaks,



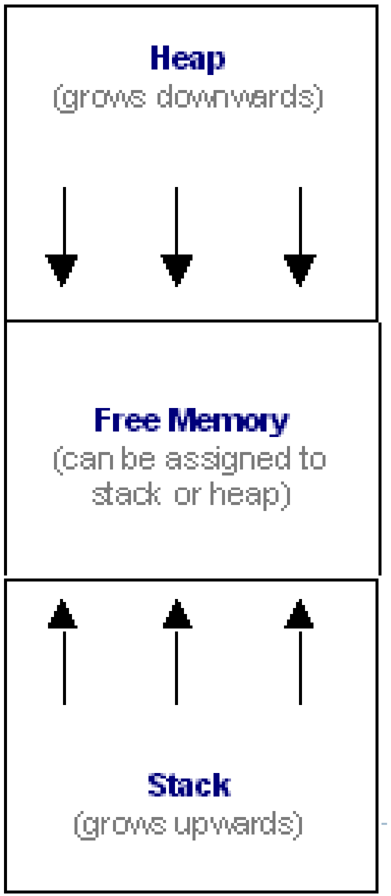
Code representation

D&R

Disassemble-and-resynthesize (Valgrind)



Memory Layout



- When a program is executed, it is given a fixed portion of memory to be used for its stack and heap.
- If the program is unable to allocate memory, it will throw an out of memory exception and this is likely to crash the program

Memory Leaks Revisited

- Stack memory is "freed" when a function returns and the current stack frame is popped off the stack.
- Therefore, memory leaks can only occur with memory on the heap.
- Dynamically allocated memory will not be freed until the delete command is called on it.

Impacts of Memory Leaks

- Many programs that leak memory, will do so very slowly.
- A program that leaks memory may run for days, weeks, or even longer before it causes a program to crash.
- This is a serious real world problem with software today!

Impacts of Memory Leaks (2)

- Programs in this class will probably never be large enough nor run long enough for memory leaks to have any noticeable effect.
- However, it is obviously bad programming practice and you will lose points on your MPs if they are leaking memory.
- A useful tool—valgrind—can be used to check a program for a variety of common errors including memory leaks

Valgrind Toolkit

- **Memcheck** is memory debugger
  - detects memory-management problems
- **Cachegrind** is a cache profiler
  - performs detailed simulation of the I1, D1 and L2 caches in your CPU
- **Massif** is a heap profiler
  - performs detailed heap profiling by taking regular snapshots of a program's heap
- **Helgrind** is a thread debugger
  - finds data races in multithreaded
  - programs

Memcheck Features

- When a program is run under Memcheck's supervision, all reads and writes of memory are checked, and calls to malloc/new/free/delete are intercepted
- Memcheck can detect:
  - Use of uninitialised memory
  - Reading/writing memory after it has been free'd
  - Reading/writing off the end of malloc'd blocks
  - Reading/writing inappropriate areas on the stack
  - Memory leaks -- where pointers to malloc'd blocks are lost forever
  - Passing of uninitialised and/or unaddressible memory to system calls
  - Mismatched use of malloc/new/new [] vs free/delete/delete []
  - Overlapping src and dst pointers in memcpy() and related functions
  - Some misuses of the POSIX pthreads API

Memcheck Example

Access of unallocated memory

Using non-initialized value

Memory leak

```
#include <iostream>

char * f() { char *cp=new char[17]; return cp; }

#define MM 100000
int main() {
  int *p= new int[10];
  p[10] = 6;

  int i,j;
  j= i+3;
  if (i>0) std::cout<<"Hi";

  f();
  free (p);
  return 0;
}
```

Using “free” of memory allocated by “new”

Memcheck Example (Cont.)

- Compile the program with –g flag:
  - g++ -c a.cc –g –o a.out
- Execute valgrind :
  - valgrind --tool=memcheck --leak-check=yes log

Debug leaks

Executable name

- View log

Memcheck report

```
Invalid write of size 4
   at 0x80486CA: main (a.cc:8)
 Address 0x1B92A050 is 0 bytes after a block of size 40 alloc'd
   at 0x1B904E35: operator new[](unsigned) (vg_replace_malloc.c:139)
   by 0x80486BD: main (a.cc:7)

Conditional jump or move depends on uninitialised value(s)
   at 0x80486DD: main (a.cc:12)

Mismatched free() / delete / delete []
   at 0x1B904FA1: free (vg_replace_malloc.c:153)
   by 0x8048703: main (a.cc:15)
 Address 0x1B92A028 is 0 bytes inside a block of size 40 alloc'd
   at 0x1B904E35: operator new[](unsigned) (vg_replace_malloc.c:139)
   by 0x80486BD: main (a.cc:7)
```

Memcheck report (cont.) Leaks detected:

```
ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 15 from 1)
malloc/free: in use at exit: 17 bytes in 1 blocks.
malloc/free: 2 allocs, 1 frees, 57 bytes allocated.
For counts of detected errors, rerun with: -v
searching for pointers to 1 not-freed blocks.
checked 2250336 bytes.


17 bytes in 1 blocks are definitely lost in loss record 1 of 1
   at 0x1B904E35: operator new[](unsigned) (vg_replace_malloc.c:139)
   by 0x8048697: f() (a.cc:3)
   by 0x80486F8: main (a.cc:14)

LEAK SUMMARY:
   definitely lost: 17 bytes in 1 blocks.
```

Before Using Valgrind

- Be sure that your executable was created from files that were complied with the -g and -O0 compiler flags
- IMPORTANT NOTE: valgrind will only detect memory leaks that are exposed by the code that executes.
  - Therefore, be sure you are running test cases that could potentially expose a leak, and be sure to test all branches of each conditional.

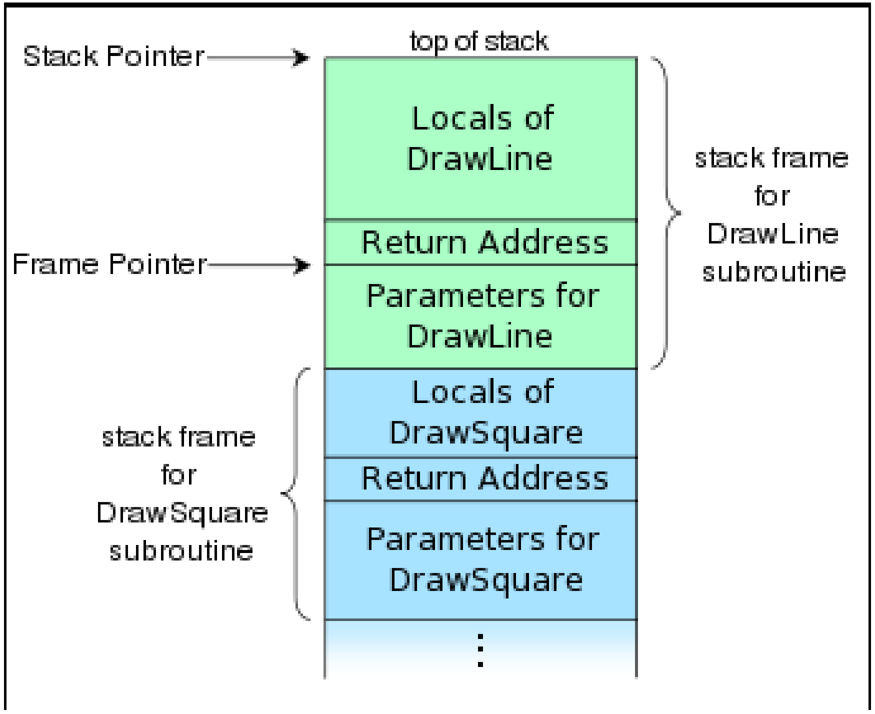
Memory Leaks in Valgrind

- Divides memory leaks into three categories:
  - "definitely lost" memory blocks
    - The pointer to the dynamically allocated memory is lost and there is no way to recover it
  - "possibly lost" memory blocks
    - The only pointer to the dynamically allocated memory is pointing to the interior of a block and may be unrelated
  - "still reachable" memory blocks
    - The pointer to the dynamically allocated memory still exists, but the memory was never freed at the end of the programs execution.

Running Valgrind

- Useful Flags:
- --leak-check=<no | summary | yes | full>
  - defaults to summary
  - yes or full will provide details for individual leaks which includes a stack trace to its location
- --show-reachable=<no | yes>
  - defaults to no
  - if enabled, valgrind will also provide information about any “still reachable” memory leaks, which are usually not considered to be serious.

A Note on Buffer Overflows



- Unrestricted access to an array stored on the stack can be exploited by a clever user
- If the return address is overwritten, malicious code might be executed

Safety Features in Java

- Java does not have this issue because:
  - It prohibits DMA (direct memory access)
  - All arrays are bounds-checked during run-time
  - Any attempt to read out of the bounds of an array will throw an ArrayIndexOutOfBounds exception.
- All of these safety features come at a performance cost.

Buffer Overflow Protection in C++

- Use the STL containers
  - They perform bounds checking for you.
- Use the std::String class rather than a C-style char* buffer when receiving input from the user