Code checking tools
Background on memory allocation
Types of problem
Uninitialized values
Invalid read / write
Mis-used delete
Memory leaks

Lab 7: Code checking tools

Comp Sci 1585
Data Structures Lab:
Tools for Computer Scientists

Code checking tools
Background on memory allocation
Types of problem
Uninitialized values
Invalid read / write
Mis-used delete
Memory leaks

Outline

1. Code checking tools
2. Background on memory allocation
3. Types of problem
   - Uninitialized values
   - Invalid read / write
   - Mis-used delete
   - Memory leaks

Code Checking Tools

Code checking tools
Background on memory allocation
Types of problem
Uninitialized values
Invalid read / write
Mis-used delete
Memory leaks

Today we will talk about tools that will help you find bugs in your code.

- `$ valgrind` and its memcheck tool
- `$ asan` is part runtime library, part compiler feature that instruments your code at compile time.
- `$ cppcheck` does static code checking (some overlap).

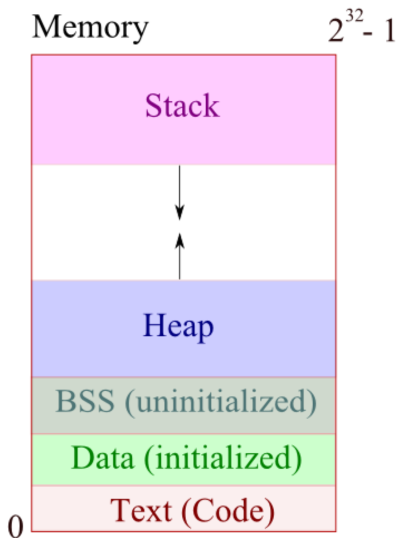
Code checking tools
Background on memory allocation
Types of problem
Uninitialized values
Invalid read / write
Mis-used delete
Memory leaks

Outline

1. Code checking tools
2. Background on memory allocation
3. Types of problem
   - Uninitialized values
   - Invalid read / write
   - Mis-used delete
   - Memory leaks

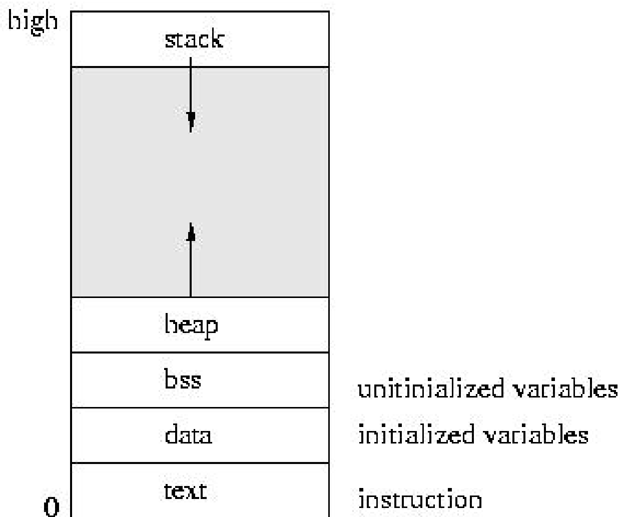
Stack and Heap

Recall the stack frames in GDB
(which you can navigate through using bt, up, down, etc)



Code checking tools
Background on memory allocation
Types of problem
Uninitialized values
Invalid read / write
Mis-used delete
Memory leaks

Stack and Heap



Stack and Heap

- The stack (on x86) starts at a high address and grows down
- The heap (on x86) starts at the bottom and grows up
- Destructors on stack-allocated class instances are called when the function returns
- Destructors on heap-allocated class instances are called when `delete` is called on the pointer

Code checking tools
Background on memory allocation
Types of problem
Uninitialized values
Invalid read / write
Mis-used delete
Memory leaks

Outline

1. Code checking tools
2. Background on memory allocation
3. Types of problem
   - Uninitialized values
   - Invalid read / write
   - Mis-used delete
   - Memory leaks

Code checking tools
Background on memory allocation
Types of problem
Uninitialized values
Invalid read / write
Mis-used delete
Memory leaks

Types of problem

1. Uninitialized values
2. Unallocated or out-of-bounds read / write
   - Out-of-bounds stack access
   - Out-of-bounds heap access
   - Use after free
3. Mismatched or double delete
4. Memory leaks

Code checking tools
Background on memory allocation
Types of problem
Uninitialized values
Invalid read / write
Mis-used delete
Memory leaks

Outline

1. Code checking tools
2. Background on memory allocation
3. Types of problem
   - Uninitialized values
   - Invalid read / write
   - Mis-used delete
   - Memory leaks

Uninitialized Values: valgrind, memory-sanitizer

Code checking tools
Background on memory allocation
Types of problem
Uninitialized values
Invalid read / write
Mis-used delete
Memory leaks

- Reading a value that hasn't been initialized from the stack or the heap.
- Especially dangerous when program flow depends on that value.
- **valgrind**
  `$ valgrind --track-origins=yes` keeps track of where uninitialized values were allocated.
- **asan** is faster
  ```
  $ g++ -g -fsanitize=address -fno-omit-frame-pointer invalid-stack.cpp -o invalid-stack
  ```
  and set environment variables (script provided today in repo: `symbolizer.sh`)
- Some IDEs check unititialized values via plugins (e.g., CodeBlocks/KDevelop and Cppcheck plugin).

Outline

1 Code checking tools

2 Background on memory allocation

3 Types of problem
- Uninitialized values
- Invalid read / write
- Mis-used delete
- Memory leaks

Invalid Reads / Write: valgrind, address-sanitizer

Code checking tools
Background on memory allocation
Types of problem
Uninitialized values
Invalid read / write
Mis-used delete
Memory leaks

- Reading or writing values from unallocated memory.
- Sometimes may result in a segfault, but not always.
- **valgrind** isn't perfect:
  you can invalidly read and write to things on the stack without complaint, though it can detect out-of-bounds heap access and use-after-free.
- **asan** works for all of these types:

```
$ g++ -g -fsanitize=address -fno-omit-frame-pointer invalid-stack.cpp -o invalid-stack
```

Outline

Code checking tools
Background on memory allocation
Types of problem
Uninitialized values
Invalid read / write
Mis-used delete
Memory leaks

1. Code checking tools
2. Background on memory allocation
3. Types of problem
   - Uninitialized values
   - Invalid read / write
   - Mis-used delete
   - Memory leaks

Misused delete: valgrind, address-sanitizer

Code checking tools
Background on memory allocation
Types of problem
Uninitialized values
Invalid read / write
Mis-used delete
Memory leaks

1. Mismatched delete, using:
   `new` with `delete[]` or
   `new[]` with `delete`
   Both are problematic, why?
2. Double delete: deleting the same memory twice.
   Why is this an issue?

valgrind and asan can both detect both

Code checking tools
Background on memory allocation
Types of problem
Uninitialized values
Invalid read / write
Mis-used delete
Memory leaks

Outline

1. Code checking tools
2. Background on memory allocation
3. Types of problem
   - Uninitialized values
   - Invalid read / write
   - Mis-used delete
   - Memory leaks

Memory Leaks: valgrind

Valgrind runs leak checks after the program terminates:

- **Directly lost:** No pointer to that block anymore.
- **Indirectly lost:** A pointer to that block exists, but it's in a directly lost block.
- **Still reachable:** Still have a pointer to that block (don't worry about this)
- **Possibly lost:** No pointer to the beginning of the block, but a pointer to somewhere inside the block.
- `$ valgrind --leak-check=full` may help you determine where
- Valgrind Memcheck Manual: `http://valgrind.org/docs/manual/mc-manual.html`

The first two are the important ones to check for on homeworks