CS 50:
Software Design and Implementation

valgrind

Agenda

1. Memory errors
2. Memory leaks
3. Activity

Often programs will compile and run, but may have sneaky memory bugs

Memory error

- **Invalid read/write of size X** – The program was observed to read/write X bytes of memory that was invalid. Common causes include:
  - accessing beyond the end of a heap block
  - accessing memory that has been freed
  - accessing into an unallocated region such as from use of a uninitialized pointer.

- **Use of uninitialized value** or **Conditional jump or move depends on uninitialized value(s)** – access memory not initialized or use uninitialized values in a conditional

- **Source and destination overlap in memcpy()** – attempt to copy data from one location to another and range intersects

- **Invalid free()** – attempt to free non-heap address or free same block more than once

Adapted from https://web.stanford.edu/class/archive/cs/cs107/cs107.1174/guide_valgrind.html

Invalid read/write of size X

valgrind_test.c

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

const int SIZE = 10;

typedef struct node {
        char *name;
        struct node  *next;
} Node;

Node *head = NULL;

int main() {

        head = malloc(sizeof(struct node));
        head->name = malloc(sizeof(char)*SIZE);
        strcpy(head->name,"Alice A. Anderson");
        head->next = NULL;
        printf("name: %s\n",head->name);

        free(head->name);
        free(head);
        return 0;
}
```

Name is too long

Runs just fine!

```
$ mygcc valgrind_test.c
$ ./a.out
name: Alice A. Anderson
```

Invalid read/write of size X

valgrind_test.c

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

const int SIZE = 10;

typedef struct node {
        char *name;
        struct node  *next;
} Node;

Node *head = NULL;

int main() {

        head = malloc(sizeof(struct node));
        head->name = malloc(sizeof(char)*SIZE);
        strcpy(head->name,"Alice A. Anderson");
        head->next = NULL;
        printf("name: %s\n",head->name);

        free(head->name);
        free(head);
        return 0;
}
```

Runs just fine!

```
$ mygcc valgrind_test.c
$ ./a.out
name: Alice A. Anderson
$ alias myvalgrind
alias myvalgrind='valgrind --leak-check=full --show-leak-kinds=all'
```

myvalgrind is our alias

Name is too long

Invalid read/write of size X

valgrind_test.c

Valgrind shows `strcpy` writes beyond end of string, and also reading it during `printf`

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

const int SIZE = 10;

typedef struct node {
        char *name;
        struct node  *next;
} Node;

Node *head = NULL;

int main() {

        head = malloc(sizeof(struct node));
        head->name = malloc(sizeof(char)*SIZE);
        strcpy(head->name,"Alice A. Anderson");
        head->next = NULL;
        printf("name: %s\n",head->name);

        free(head->name);
        free(head);
        return 0;
}
```

Name is too long

```
$ mygcc valgrind_test.c
$ ./a.out
name: Alice A. Anderson
$ alias myvalgrind
alias myvalgrind='valgrind --leak-check=full --show-leak-kinds=all'
$ myvalgrind ./a.out
==7314== Memcheck, a memory error detector
==7314== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7314== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==7314== Command: ./a.out
==7314==
==7314== Invalid write of size 8
==7314==    at 0x10872E: main (test.c:18)
==7314==  Address 0x522f098 is 8 bytes inside a block of size 10 alloc'd
==7314==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==7314==    by 0x108709: main (test.c:17)
==7314==
==7314== Invalid write of size 2
==7314==    at 0x108732: main (test.c:18)
==7314==  Address 0x522f0a0 is 6 bytes after a block of size 10 alloc'd
==7314==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==7314==    by 0x108709: main (test.c:17)
==7314==
==7314== Invalid read of size 1
==7314==    at 0x4C34D04: strlen (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==7314==    by 0x4E9B4A2: vfprintf (vfprintf.c:1643)
==7314==    by 0x4EA2EE5: printf (printf.c:33)
==7314==    by 0x108764: main (test.c:20)
==7314==  Address 0x522f09a is 0 bytes after a block of size 10 alloc'd
==7314==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==7314==    by 0x108709: main (test.c:17)
<snip>
```

Use of uninitialised value or Conditional jump or move depends on uninitialised value(s)

The program read the value of a memory location that was not previously written to, i.e. uses random junk. The second more specifically indicates the read occurred in the test expression in an if/for/while.

valgrind_test2.c

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

const int SIZE = 10;

typedef struct node {
        char *name;
        struct node  *next;
} Node;

Node *head = NULL;

int main() {

        head = malloc(sizeof(struct node));
        head->name = malloc(sizeof(char)*SIZE);
        strcat(head->name,"Alice");
        head->next = NULL;
        printf("name: %s\n",head->name);

        free(head->name);
        free(head);
        return 0;
}
```

Runs just fine!

```
$ mygcc valgrind_test2.c
$ ./a.out
name: Alice
```

strcat concatenates strings

Alice now fits

Use of uninitialised value or Conditional jump or move depends on uninitialised value(s)

The program read the value of a memory location that was not previously written to, i.e. uses random junk. The second more specifically indicates the read occurred in the test expression in an if/for/while.

valgrind_test2.c

Valgrind shows `strcat` concatenates uninitialized memory with "Alice"

Runs just fine!

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

const int SIZE = 10;

typedef struct node {
        char *name;
        struct node  *next;
} Node;

Node *head = NULL;

int main() {

        head = malloc(sizeof(struct node));
        head->name = malloc(sizeof(char)*SIZE);
        strcat(head->name,"Alice");
        head->next = NULL;
        printf("name: %s\n",head->name);

        free(head->name);
        free(head);
        return 0;
}
```

`strcat` concatenates strings

Alice now fits

```
$ mygcc valgrind_test2.c
$ ./a.out
name: Alice
$ myvalgrind ./a.out
==7720== Memcheck, a memory error detector
==7720== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7720== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==7720== Command: ./a.out
==7720==
==7720== Conditional jump or move depends on uninitialised value(s)
==7720==    at 0x10872C: main (valgrind_test2.c:18)
==7720==
name: Alice
==7720==
==7720== HEAP SUMMARY:
==7720==     in use at exit: 0 bytes in 0 blocks
==7720==   total heap usage: 3 allocs, 3 frees, 1,050 bytes allocated
==7720==
==7720== All heap blocks were freed -- no leaks are possible
==7720==
==7720== For counts of detected and suppressed errors, rerun with: -v
==7720== Use --track-origins=yes to see where uninitialised values come from
==7720== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Invalid free()

The program attempted to free a non-heap address or free the same block more than once.

valgrind_test3.c

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

const int SIZE = 10;

typedef struct node {
        char *name;
        struct node  *next;
} Node;

Node *head = NULL;

int main() {

        head = malloc(sizeof(struct node));
        head->name = malloc(sizeof(char)*SIZE);
        strcpy(head->name,"Alice");
        head->next = NULL;
        printf("name: %s\n",head->name);

        free(head->name);
        free(head);
        free(head);
        return 0;
}
```

Runs but core dump at unknown location

strcpy copies strings (as before)

head freed twice

Valgrind shows location of double free

```
$ mygcc valgrind_test3.c
$ ./a.out
name: Alice
free(): double free detected in tcache 2
Aborted (core dumped)
$ myvalgrind ./a.out
$ myvalgrind ./a.out
==8066== Memcheck, a memory error detector
==8066== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8066== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==8066== Command: ./a.out
==8066==
name: Alice
==8066== Invalid free() / delete / delete[] / realloc()
==8066==    at 0x4C32D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==8066==    by 0x10877F: main (valgrind_test3.c:24)
==8066==  Address 0x522f040 is 0 bytes inside a block of size 16 free'd
==8066==    at 0x4C32D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==8066==    by 0x108770: main (valgrind_test3.c:23)
==8066==  Block was alloc'd at
==8066==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==8066==    by 0x1086EC: main (valgrind_test3.c:16)
```

Agenda

1. Memory errors
2. Memory leaks
3. Activity

Often programs will compile and run, but may have sneaky memory bugs

Memory leak

- **Definitely lost** – heap-allocated memory never freed, but lost pointer to it
- **Indirectly lost** - heap-allocated memory never freed to which only pointers to it are lost (e.g., free head, but loose rest of list)
- **Possibly lost** - heap-allocated memory never freed, but Valgrind is not sure if there is a pointer to it
- **Still reachable** - heap-allocated memory never freed to which the program still has a pointer at exit (typically this means a global variable points to it).

Adapted from https://web.stanford.edu/class/archive/cs/cs107/cs107.1174/guide_valgrind.html

Definitely lost

heap-allocated memory that was never freed to which the program no longer has a pointer. Valgrind knows that you once had the pointer, but have since lost track of it. This memory is definitely orphaned.

valgrind_test4.c

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

const int SIZE = 10;

typedef struct node {
        char *name;
        struct node  *next;
} Node;

Node *head = NULL;

int main() {

        head = malloc(sizeof(struct node));
        head->name = malloc(sizeof(char)*SIZE);
        strcpy(head->name,"Alice");
        head->next = NULL;
        printf("name: %s\n",head->name);

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

Runs fine

```
$ mygcc valgrind_test4.c
$ ./a.out
name: Alice
```

Double free fixed

Definitely lost

heap-allocated memory that was never freed to which the program no longer has a pointer. Valgrind knows that you once had the pointer, but have since lost track of it. This memory is definitely orphaned.

valgrind_test4.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  const int SIZE = 10;
6
7  typedef struct node {
8          char *name;
9          struct node  *next;
10 } Node;
11
12 Node *head = NULL;
13
14 int main() {
15
16         head = malloc(sizeof(struct node));
17         head->name = malloc(sizeof(char)*SIZE);
18         strcpy(head->name,"Alice");
19         head->next = NULL;
20         printf("name: %s\n",head->name);
21
22         free(head);
23         return 0;
24 }
```

Runs fine

name not freed

Free head

Valgrind shows location of malloc that was not freed

```
$ mygcc valgrind_test4.c
$ ./a.out
name: Alice
$ myvalgrind ./a.out
==8354== Memcheck, a memory error detector
==8354== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8354== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==8354== Command: ./a.out
==8354==
name: Alice
==8354==
==8354== HEAP SUMMARY:
==8354==     in use at exit: 10 bytes in 1 blocks
==8354==   total heap usage: 3 allocs, 2 frees, 1,050 bytes allocated
==8354==
==8354== 10 bytes in 1 blocks are definitely lost in loss record 1 of 1
==8354==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8354==    by 0x108709: main (valgrind_test4.c:17)
==8354==
==8354== LEAK SUMMARY:
==8354==    definitely lost: 10 bytes in 1 blocks
==8354==    indirectly lost: 0 bytes in 0 blocks
==8354==      possibly lost: 0 bytes in 0 blocks
==8354==    still reachable: 0 bytes in 0 blocks
==8354==         suppressed: 0 bytes in 0 blocks
==8354==
==8354== For counts of detected and suppressed errors, rerun with: -v
==8354== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Indirectly lost

heap-allocated memory that was never freed to which the only pointers to it also are lost. For example, if you orphan a linked list, the first node would be definitely lost, the subsequent nodes would be indirectly lost.

valgrind_test5.c

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

const int SIZE = 10;

typedef struct node {
        char *name;
        struct node  *next;
} Node;

Node *head = NULL;

int main() {

  head = malloc(sizeof(struct node));
  head->name = malloc(sizeof(char)*SIZE);
  strcpy(head->name,"Alice");
  head->next = NULL;
  printf("name: %s\n",head->name);

  head->next = malloc(sizeof(Node));
  head->next->name = malloc(sizeof(char)*SIZE);
  strcpy(head->next->name,"Bob");
  head->next->next = NULL;
  printf("name: %s\n",head->next->name);

  free(head->name);
  free(head);
  return 0;
}
```

Runs fine

```
$ mygcc valgrind_test5.c
$ ./a.out
name: Alice
name: Bob
```

Allocate second node

Indirectly lost

heap-allocated memory that was never freed to which the only pointers to it also are lost. For example, if you orphan a linked list, the first node would be definitely lost, the subsequent nodes would be indirectly lost.

valgrind_test5.c

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

const int SIZE = 10;

typedef struct node {
        char *name;
        struct node  *next;
} Node;

Node *head = NULL;

int main() {

  head = malloc(sizeof(struct node));
  head->name = malloc(sizeof(char)*SIZE);
  strcpy(head->name,"Alice");
  head->next = NULL;
  printf("name: %s\n",head->name);

  head->next = malloc(sizeof(Node));
  head->next->name = malloc(sizeof(char)*SIZE);
  strcpy(head->next->name,"Bob");
  head->next->next = NULL;
  printf("name: %s\n",head->next->name);

  free(head->name);
  free(head);
  return 0;
}
```

Runs fine

Allocate second node

```
$ mygcc valgrind_test5.c
$ ./a.out
name: Alice
name: Bob
$ myvalgrind ./a.out
==8938== Memcheck, a memory error detector
==8938== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8938== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==8938== Command: ./a.out
==8938==
name: Alice
name: Bob
==8938==
==8938== HEAP SUMMARY:
==8938==     in use at exit: 26 bytes in 2 blocks
==8938==   total heap usage: 5 allocs, 3 frees, 1,076 bytes allocated
==8938==
==8938== 10 bytes in 1 blocks are indirectly lost in loss record 1 of 2
==8938==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==8938==    by 0x10877E: main (valgrind_test5.c:23)
==8938==
==8938== 26 (16 direct, 10 indirect) bytes in 1 blocks are definitely lost
in loss record 2 of 2
==8938==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==8938==    by 0x108760: main (valgrind_test5.c:22)
==8938==
==8938== LEAK SUMMARY:
==8938==    definitely lost: 16 bytes in 1 blocks
==8938==    indirectly lost: 10 bytes in 1 blocks
==8938==      possibly lost: 0 bytes in 0 blocks
==8938==    still reachable: 0 bytes in 0 blocks
==8938==         suppressed: 0 bytes in 0 blocks
==8938==
==8938== For counts of detected and suppressed errors, rerun with: -v
==8938== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Second node definitely lost

Second name indirectly lost

Still reachable

heap-allocated memory that was never freed to which the program still has a pointer at exit (typically this means a global variable points to it).

valgrind_test6.c

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

const int SIZE = 10;

typedef struct node {
   char *name;
   struct node  *next;
} Node;

Node *head = NULL;

int main() {
   head = malloc(sizeof(struct node));
   head->name = malloc(sizeof(char)*SIZE);
   strcpy(head->name,"Alice");
   head->next = NULL;
   printf("name: %s\n",head->name);

   free(head->name);
   return 0;
}
```

Runs fine

Head is a global variable

head is not freed

```
$ mygcc valgrind_test6.c
$ ./a.out
name: Alice
$ myvalgrind ./a.out
==9263== Memcheck, a memory error detector
==9263== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==9263== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==9263== Command: ./a.out
==9263==
name: Alice
==9263==
==9263== HEAP SUMMARY:
==9263==     in use at exit: 16 bytes in 1 blocks
==9263==   total heap usage: 3 allocs, 2 frees, 1,050 bytes allocated
==9263==
==9263== 16 bytes in 1 blocks are still reachable in loss record 1 of 1
==9263==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==9263==    by 0x1086EC: main (valgrind_test6.c:15)
==9263==
==9263== LEAK SUMMARY:
==9263==    definitely lost: 0 bytes in 0 blocks
==9263==    indirectly lost: 0 bytes in 0 blocks
==9263==      possibly lost: 0 bytes in 0 blocks
==9263==    still reachable: 16 bytes in 1 blocks
==9263==         suppressed: 0 bytes in 0 blocks
==9263==
==9263== For counts of detected and suppressed errors, rerun with: -v
==9263== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Agenda

1. Memory errors
2. Memory leaks
3. Activity