CS29206 Systems Programming Laboratory Spring 2024

Introduction to valgrind

Abhijit Das
Pralay Mitra

What is valgrind?

- Valgrind is not value grinder. It is the gate to Valhalla (Hall of the Slain).
- Our valgrind is a memory debugging tool.
- Capable of detecting many common memory-related errors and problems.
- Also used for various memory-related profiling.
- Here, we use only the memcheck feature of valgrind.
    - Memory leaks
    - Invalid memory access

How to use valgrind

- Run as: `valgrind executable <command line options>`
- Example: `valgrind ./a.out 2022 -name "Sad Tijihba"`
- Sample valgrind output

```
==4825== Memcheck, a memory error detector
==4825== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4825== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4825== Command: ./a.out
==4825==

Input/Output of your program

==4825==
==4825== HEAP SUMMARY:
==4825==     in use at exit: 0 bytes in 0 blocks
==4825==   total heap usage: 23 allocs, 23 frees, 1,376 bytes allocated
==4825==
==4825== All heap blocks were freed -- no leaks are possible
==4825==
==4825== For lists of detected and suppressed errors, rerun with: -s
==4825== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

- valgrind can also issue error/warning messages inside your program's transcript.

Common memory issues

Memory leaks

- Memory is allocated in blocks by malloc, calloc, realloc.
- Allocated memory should be free'd when no longer in use.
- Types of memory allocated but not free'd.

Reachable	Blocks not free'd but can be accessed until the end of the program.
Definitely lost	Blocks not free'd and not accessible by any pointer.
Indirectly lost	Blocks not free'd and accessible only by pointers in other lost blocks.
Possibly lost	Blocks not free'd and accessible only by pointers in their interiors.

Invalid memory access

- Read/write a memory location outside the allocated area of an array.

An implementation of sorted linked lists

The node data type

```
typedef struct _node {
   int data;
   struct _node *next;
} node;
```

- In my machine, sizeof(node) is 16.
- We maintain a dummy node at the beginning of the list.
- An initialization function only creates the dummy node.
- An insert function makes a sorted insertion in the list. If the element to be inserted is already present, no new node is created.
- A delete function deletes a data from the list. If that data is not present in the list, the list is kept unchanged.

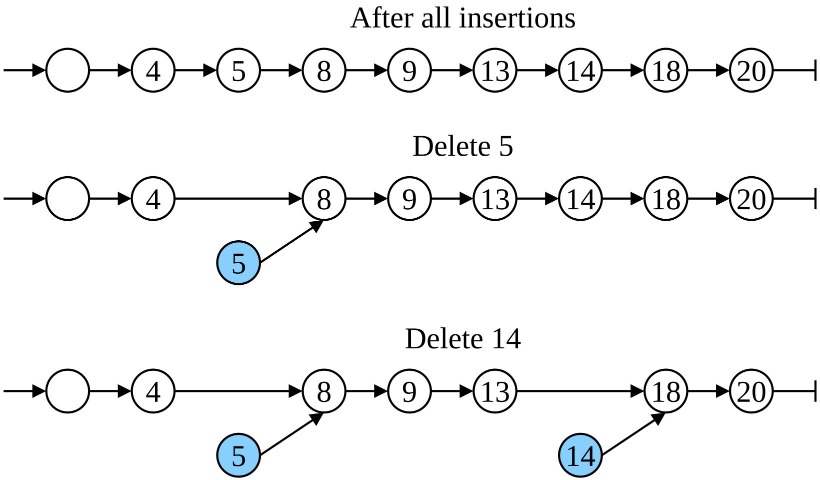
Delete without freeing: Example

```
node *ldel ( node *L, int x )
{
   node *p;

   p = L;
   while (p -> next) {
      if (p -> next -> data == x) {
         p -> next = p -> next -> next;
         return L;
      }
      if (p -> next -> data > x) break;
      p = p -> next;
   }
   return L;
}
```

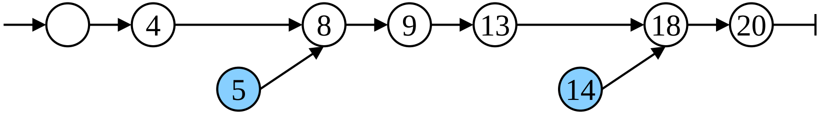
Delete without freeing: Example

```
sizeof(node) = 16
insert  5 : 5
insert 14 : 5 14
insert 18 : 5 14 18
insert  4 : 4 5 14 18
insert 13 : 4 5 13 14 18
insert 20 : 4 5 13 14 18 20
insert  8 : 4 5 8 13 14 18 20
insert  9 : 4 5 8 9 13 14 18 20
delete  5 : 4 8 9 13 14 18 20
delete 14 : 4 8 9 13 18 20
delete 18 : 4 8 9 13 20
delete 13 : 4 8 9 20
delete 20 : 4 8 9
==9837==
==9837== HEAP SUMMARY:
==9837==     in use at exit: 144 bytes in 9 blocks
==9837==   total heap usage: 10 allocs, 1 frees, 1,168 bytes allocated
==9837==
==9837== LEAK SUMMARY:
==9837==    definitely lost: 48 bytes in 3 blocks
==9837==    indirectly lost: 32 bytes in 2 blocks
==9837==      possibly lost: 0 bytes in 0 blocks
==9837==    still reachable: 64 bytes in 4 blocks
==9837==         suppressed: 0 bytes in 0 blocks
==9837== Rerun with --leak-check=full to see details of leaked memory
==9837==
==9837== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

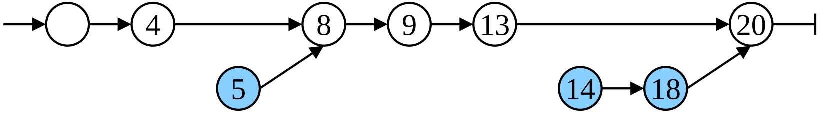
Memory leak example



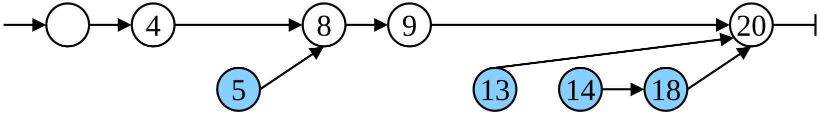
Memory leak example: Continued



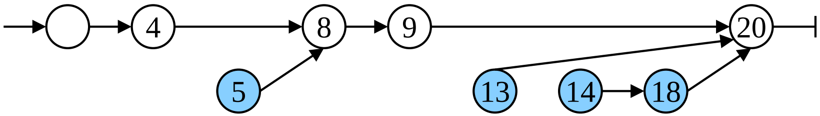
Delete 18



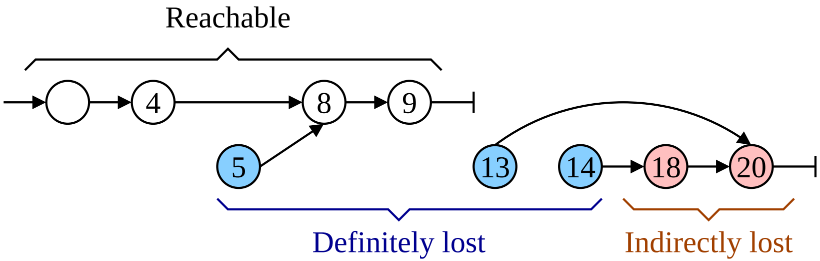
Delete 13



Memory leak example: Continued



Delete 20



Delete with freeing

```
node *ldel ( node *L, int x )
{
   node *p, *q;
   p = L;
   while (p -> next) {
      if (p -> next -> data == x) {
         q = p -> next; p -> next = q -> next; free(q);
         return L;
      }
      if (p -> next -> data > x) break;
      p = p -> next;
   }
   return L;
}
```

```
==10012==
==10012== HEAP SUMMARY:
==10012==     in use at exit: 64 bytes in 4 blocks
==10012==   total heap usage: 10 allocs, 6 frees, 1,168 bytes allocated
==10012==
==10012== LEAK SUMMARY:
==10012==    definitely lost: 0 bytes in 0 blocks
==10012==    indirectly lost: 0 bytes in 0 blocks
==10012==      possibly lost: 0 bytes in 0 blocks
==10012==    still reachable: 64 bytes in 4 blocks
==10012==         suppressed: 0 bytes in 0 blocks
==10012== Rerun with --leak-check=full to see details of leaked memory
==10012==
==10012== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

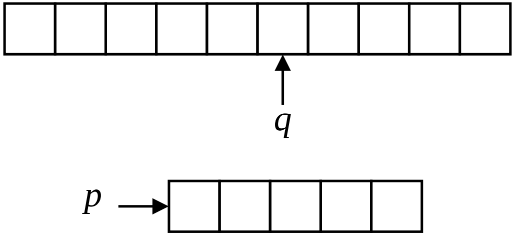
Free the reachable nodes at the end

```
void ldestroy ( node *L )
{
   node *p;

   while (L) {
      p = L -> next;
      free(L);
      L = p;
   }
}
```

```
==10160==
==10160== HEAP SUMMARY:
==10160==     in use at exit: 0 bytes in 0 blocks
==10160==   total heap usage: 10 allocs, 10 frees, 1,168 bytes allocated
==10160==
==10160== All heap blocks were freed -- no leaks are possible
==10160==
==10160== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Demonstrating possibly lost memory

```
int main ()
{
   int *p, *q;

   p = (int *)malloc(10 * sizeof(int));
   q = p + 5;

   p = (int *)malloc(5 * sizeof(int));

   exit(0);
}
```



Possibly lost memory: valgrind output

```
==4155== Memcheck, a memory error detector
==4155== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4155== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4155== Command: ./a.out
==4155==
==4155==
==4155== HEAP SUMMARY:
==4155==     in use at exit: 60 bytes in 2 blocks
==4155==   total heap usage: 2 allocs, 0 frees, 60 bytes allocated
==4155==
==4155== LEAK SUMMARY:
==4155==    definitely lost: 0 bytes in 0 blocks
==4155==    indirectly lost: 0 bytes in 0 blocks
==4155==      possibly lost: 40 bytes in 1 blocks
==4155==    still reachable: 20 bytes in 1 blocks
==4155==         suppressed: 0 bytes in 0 blocks
==4155== Rerun with --leak-check=full to see details of leaked memory
==4155==
==4155== For lists of detected and suppressed errors, rerun with: -s
==4155== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Overflow in arrays

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

int main ()
{
   int n = 16, i, *A;

   A = (int *)malloc(n * sizeof(int));
   printf("A starts at %p, and ends at %p\n", A, A+n-1);
   for (i=1; i<=n; ++i) A[i] = i * i;
   for (i=1; i<=n; ++i) printf("%d ", A[i]);
   printf("\n");

   free(A);

   exit(0);
}
```

Overflow in arrays: valgrind output

```
==13180== Memcheck, a memory error detector
==13180== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13180== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==13180== Command: ./a.out
==13180==
A starts at 0x4a5a040, and ends at 0x4a5a07c
==13180== Invalid write of size 4
==13180==    at 0x109240: main (in /home/abhij/IITKGP/course/lab/SPL/Spring22/prog/valgrind/a.out)
==13180== Address 0x4a5a080 is 0 bytes after a block of size 64 alloc'd
==13180==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==13180==    by 0x1091EC: main (in /home/abhij/IITKGP/course/lab/SPL/Spring22/prog/valgrind/a.out)
==13180==
==13180== Invalid read of size 4
==13180==    at 0x10926B: main (in /home/abhij/IITKGP/course/lab/SPL/Spring22/prog/valgrind/a.out)
==13180== Address 0x4a5a080 is 0 bytes after a block of size 64 alloc'd
==13180==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==13180==    by 0x1091EC: main (in /home/abhij/IITKGP/course/lab/SPL/Spring22/prog/valgrind/a.out)
==13180==
1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256
==13180==
==13180== HEAP SUMMARY:
==13180==     in use at exit: 0 bytes in 0 blocks
==13180==   total heap usage: 2 allocs, 2 frees, 1,088 bytes allocated
==13180==
==13180== All heap blocks were freed -- no leaks are possible
==13180==
==13180== For lists of detected and suppressed errors, rerun with: -s
==13180== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Buffer overflow: Example

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

int main ( int argc, char *argv[] )
{
   char *wnote = malloc(32);

   if (argc == 1) exit(1);

   printf("The input has size %ld\n", strlen(argv[1]));
   printf("wnote starts at %p and ends at %p\n", wnote, wnote + 31);
   sprintf(wnote, "Welcome to %s", argv[1]);
   printf("%s\n", wnote);

   free(wnote);

   exit(0);
}
```

- Compile and run as follows. There may be no errors.

  ```
  ./a.out "Systems Programming Laboratory"
  ```

- Run as follows to see the problems.

  ```
  valgrind ./a.out "Systems Programming Laboratory"
  ```

Buffer overflow: valgrind output

```
==12432== Command: ./a.out Systems\ Programming\ Laboratory
==12432==
The input has size 30
wnote starts at 0x4a5a040 and ends at 0x4a5a05f
==12432== Invalid write of size 1
==12432== Address 0x4a5a060 is 0 bytes after a block of size 32 alloc'd
==12432== Invalid write of size 1
==12432== Address 0x4a5a069 is 9 bytes after a block of size 32 alloc'd
==12432== Invalid read of size 1
==12432== Address 0x4a5a060 is 0 bytes after a block of size 32 alloc'd
==12432== Invalid read of size 1
==12432== Address 0x4a5a068 is 8 bytes after a block of size 32 alloc'd
==12432== Invalid read of size 1
==12432== Address 0x4a5a066 is 6 bytes after a block of size 32 alloc'd
==12432==
Welcome to Systems Programming Laboratory
==12432==
==12432== HEAP SUMMARY:
==12432==     in use at exit: 0 bytes in 0 blocks
==12432==   total heap usage: 2 allocs, 2 frees, 1,056 bytes allocated
==12432==
==12432== All heap blocks were freed -- no leaks are possible
==12432==
==12432== For lists of detected and suppressed errors, rerun with: -s
==12432== ERROR SUMMARY: 38 errors from 6 contexts (suppressed: 0 from 0)
```

Practice exercises

1. Consider the sorted linked list $4 \rightarrow 5 \rightarrow 8 \rightarrow 9 \rightarrow 13 \rightarrow 14 \rightarrow 18 \rightarrow 20$ with a dummy node at the beginning, as described in the slides. You delete all the eight keys stored in the list in some order. The deletion function does not free the node being removed from the list. What would the order of deletion be in the following two cases?

   (a) The amount of definitely lost memory is small as possible.
   (b) The amount of definitely lost memory is as large as possible.

   What are these amounts? Verify your answers by running valgrind on the actual list.

2. Create a *circular* linked list on the example of the previous exercise. Keep no dummy node at the beginning. The next pointer of the node 20 points to the node 4. An external (header) pointer `L` points to the node 4. After this list is prepared, do one of the following things, and exit.

   (a) `L = NULL;`
   (b) `L -> next = NULL;`
   (c) `free(L);`
   (d) `free(L -> next);`
   (e) `L = L -> next;`

   Run the code with valgrind in each case, and note the different types of memory leaks. Explain.

Practice exercises

3. Consider a BST with each node storing only an `int` data and two child pointers `L` and `R`. Write a function to insert in the BST using the standard algorithm (no height balancing). Inside the main() function, start with an empty (NULL) BST `T`. Insert the following data items in the given order using the insert function: 6, 9, 3, 2, 10, 7, 5, 8, 4. Then, do one of the following things, and exit.

   (a) `T = NULL;`
   (b) `free(T);`
   (c) `T -> L = T -> R = NULL;`
   (d) `free(T -> L); free(T -> R);`
   (e) `free(T -> L); free(T -> R); free(T);`

   Run your code with valgrind in each case, and note the different types of memory leaks. Explain.

4. Assume that in a machine each pointer is of size 4 bytes, and each `int` variable is also of size 4 bytes. Let `p` be a variable of type `int **`. A C program executes the following statements, and then exits. Calculate what valgrind would summarize at the end of the program about the memory leaks of types

   (a) still reachable,
   (b) definitely lost,
   (c) indirectly lost.

   Mention the leaked memory sizes in bytes and numbers of blocks. Do not report valgrind output, but clearly show/explain your calculations for different types of memory leaks that valgrind would detect.

Practice exercises

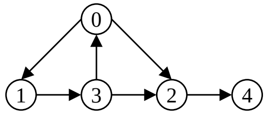
```
p = (int **)malloc(4 * sizeof(int *));
p[0] = (int *)malloc(9 * sizeof(int));
p[1] = (int *)malloc(8 * sizeof(int));
p[2] = (int *)malloc(7 * sizeof(int));
p[3] = (int *)malloc(5 * sizeof(int));
p[3] = (int *)malloc(6 * sizeof(int));

p = (int **)malloc(2 * sizeof(int *));
p[0] = p[1] = (int *)malloc(10 * sizeof(int));
free(*p);
```

5. Repeat the last exercise with the following `main()` function.

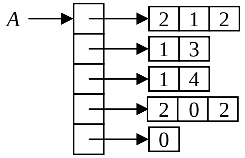
```
p = (int **)malloc(3 * sizeof(int *));
p[0] = p[1] = (int *)malloc(10 * sizeof(int));
p[1] = p[2] = (int *)malloc(11 * sizeof(int));
p[2] = p[0] = (int *)malloc(12 * sizeof(int));

p = (int **)malloc(3 * sizeof(int *));
*p = (int *)malloc(5 * sizeof(int));
*(p+1) = (int *)malloc(6 * sizeof(int));
*(p+2) = (int *)malloc(7 * sizeof(int));
p[0] = p[1] = p[2] = NULL;
```

Practice exercises

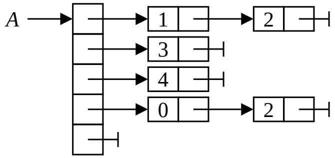
For the next two exercises, see the following figure. Part (a) of the figure shows a directed graph. Parts (b) and (c) show two representations of the adjacency list *A* of the graph.



(a)



(b)



(c)

6. *A* is an array of type `int **` (see Part (b) of the figure). For each node *u*, `A[u][0]` stores the number of neighbors of *u*. If that number is *d*, then the neighbors are stored in `A[u][1]`, `A[u][2]`, . . . , `A[u][d]`. Allocate each `A[u]` the exact amount of memory as needed for the given graph. Write a function `gengraph()` that creates and returns `A`. Also, write a function `prngraph(A,5)` to print the graph. After calling these functions, the `main()` function does one of the following things, and then exits.
   (a) `A = NULL;`
   (b) `free(A);`

   Run the code with valgrind in each case, and note the different types of memory leaks. Explain.
7. Repeat the previous exercise with the exception that the adjacency lists of the nodes are now implemented as linked lists (see Part (c) of the figure).

Practice exercises

8. Define a `struct stud` data type consisting of a character array `rollno` of size 16, a character array `name` of size 32, an integer `age`, and a floating-point value `cgpa`. Write a function `genstudarray(n)` that allocates an array of *n* records of type `struct stud`, and returns (a pointer to) that array. Now, consider the following two `main()` functions.

```
int main ()
{
   struct stud *S;
   struct stud *p;

   S = genstudarray(10);
   p = S + 5;
   S = genstudarray(5);

   exit(0);
}
```

```
int main ()
{
   struct stud *S;
   char *p;

   S = genstudarray(10);
   p = (S + 5) -> name;
   S = genstudarray(5);

   exit(0);
}
```

In each case, find the types of memory losses as reported by valgrind. Here, `p` is an internal pointer to the first allocation of `S`. Can you free the first allocation of `S` after the second allocation, using the internal pointer `p`?

Practice exercises

9. Consider the following code fragment which uses an $M \times N$ `int` array allocated as a single block.

```
#define M 5
#define N 7
typedef int row[N];
row *A = malloc(M * sizeof(row));
printf("A = %p\n", A);
for (i=0; i<M; ++i) {
   printf("i = %d\n", i);
   A[i][2*N+i] = 5;
}
```

Running the code with valgrind gives the following lines:

```
A = 0x4a5b040
i = 0
i = 1
i = 2
i = 3
==12345== Invalid write of size 4
==12345==    at 0x109200: main (in /home/foobar/a.out)
==12345== Address 0x4a5b0d8 is 12 bytes after a block of size 140 alloc'd
```

Explain why this happens, and the significance of “size 4” and “12 bytes” in the valgrind output.