

Recitation 4

Section 5

Topics for Today

1. Valgrind
2. Memory Allocation Strategies
3. Homework :(

Valgrind

1. Valgrind is a multipurpose memory debugging tool for Linux.
2. You can run your code in Valgrind's own environment, and this lets you monitor memory usage. For e.g. calls to malloc and free.
3. Suppose you use initialized memory, write off the end of an array or forget to free memory, then Valgrind can detect this.

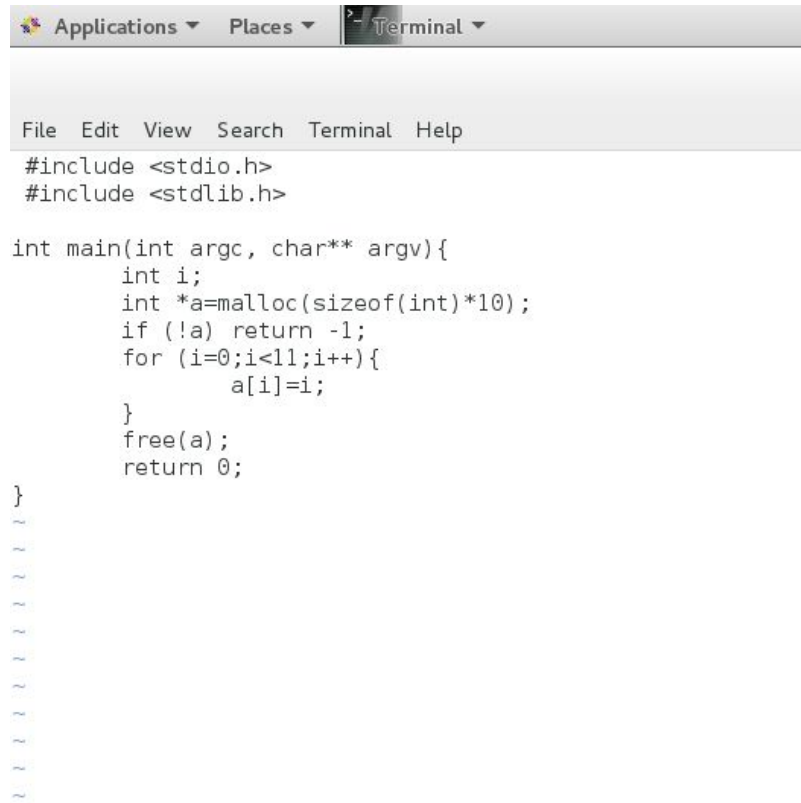
What else can Valgrind do?

1. Other than memory management, Valgrind can do a host of other things. It has four main modules.
2. **Memcheck:** This is the main module that provides memory leak detection.
3. **Massif:** Allows you to get information about memory leaks in different parts of the program.
4. **Callgrind:** Allows you to analyze function calls, build a tree of function calls, and analyze performance.
5. **Helgrind:** Analyse executed code for presence of synchronization errors.
6. **Cachegrind:** Analyse execution of code, collecting data about processor cache misses and code branching.

Using Valgrind

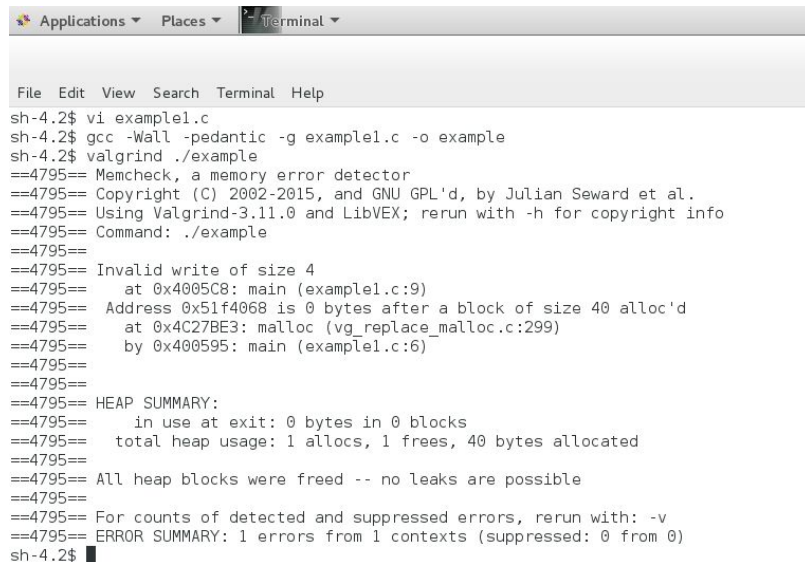
1. As is the case with gdb, compile your program with the `-g` flag.
2. To invoke it on an executable, say `a.out` simply run the command `valgrind ./a.out`, this is so that you can see the line numbers in the output.
3. Debug with optimization (`-O0`) turned off, since if you have them on, it will give you wrong line numbers and you may also occasionally have some false alarms.

Example 1



```
Applications ▾ Places ▾ Terminal ▾  
File Edit View Search Terminal Help  
#include <stdio.h>  
#include <stdlib.h>  
  
int main(int argc, char** argv){  
    int i;  
    int *a=malloc(sizeof(int)*10);  
    if (!a) return -1;  
    for (i=0;i<11;i++){  
        a[i]=i;  
    }  
    free(a);  
    return 0;  
}  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~
```

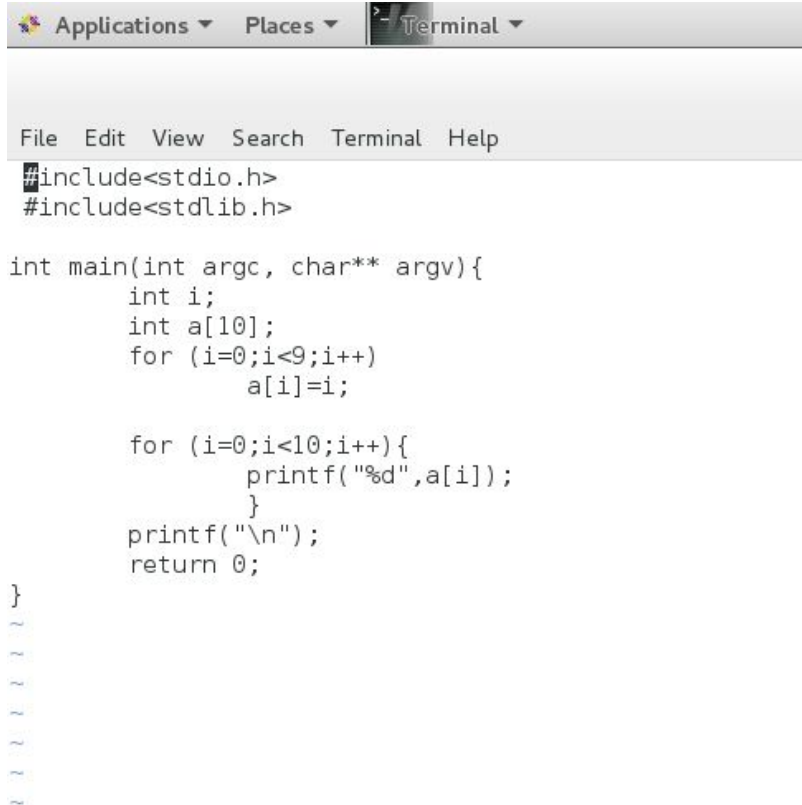
Example 1



```
sh-4.2$ vi example1.c
sh-4.2$ gcc -Wall -pedantic -g example1.c -o example
sh-4.2$ valgrind ./example
==4795== Memcheck, a memory error detector
==4795== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4795== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4795== Command: ./example
==4795==
==4795== Invalid write of size 4
==4795==    at 0x4005C8: main (example1.c:9)
==4795==    Address 0x51f4068 is 0 bytes after a block of size 40 alloc'd
==4795==    at 0x4C27BE3: malloc (vg_replace_malloc.c:299)
==4795==    by 0x400595: main (example1.c:6)
==4795==
==4795== HEAP SUMMARY:
==4795==    in use at exit: 0 bytes in 0 blocks
==4795==    total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==4795==
==4795== All heap blocks were freed -- no leaks are possible
==4795==
==4795== For counts of detected and suppressed errors, rerun with: -v
==4795== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
sh-4.2$
```

Examining the output, we know that there was one error. **Don't worry about suppressed errors.** Valgrind also tells you the type of error, the file number and the line number.

Example 2



```
Applications ▾ Places ▾ Terminal ▾  
File Edit View Search Terminal Help  
#include<stdio.h>  
#include<stdlib.h>  
  
int main(int argc, char** argv){  
    int i;  
    int a[10];  
    for (i=0;i<9;i++){  
        a[i]=i;  
  
        for (i=0;i<10;i++){  
            printf("%d",a[i]);  
        }  
        printf("\n");  
    }  
    return 0;  
}
```


Example 2

```
sh-4.2$ vi example2.c
sh-4.2$ gcc -Wall -pedantic -g example2.c -o example
sh-4.2$ valgrind ./example
==32161== Memcheck, a memory error detector
==32161== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==32161== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==32161== Command: ./example
==32161==
==32161== Conditional jump or move depends on uninitialised value(s)
==32161==   at 0x4E7A9F2: vfprintf (in /usr/lib64/libc-2.17.so)
==32161==   by 0x4E84878: printf (in /usr/lib64/libc-2.17.so)
==32161==   by 0x4005CD: main (example2.c:11)
==32161==
==32161== Use of uninitialised value of size 8
==32161==   at 0x4E79E8B: _itoa_word (in /usr/lib64/libc-2.17.so)
==32161==   by 0x4E7AF85: vfprintf (in /usr/lib64/libc-2.17.so)
==32161==   by 0x4E84878: printf (in /usr/lib64/libc-2.17.so)
==32161==   by 0x4005CD: main (example2.c:11)
==32161==
==32161== Conditional jump or move depends on uninitialised value(s)
==32161==   at 0x4E79E95: _itoa_word (in /usr/lib64/libc-2.17.so)
==32161==   by 0x4E7AF85: vfprintf (in /usr/lib64/libc-2.17.so)
==32161==   by 0x4E84878: printf (in /usr/lib64/libc-2.17.so)
==32161==   by 0x4005CD: main (example2.c:11)
==32161==
==32161== Conditional jump or move depends on uninitialised value(s)
==32161==   at 0x4E7AF54: vfprintf (in /usr/lib64/libc-2.17.so)
==32161==   by 0x4E84878: printf (in /usr/lib64/libc-2.17.so)
==32161==   by 0x4005CD: main (example2.c:11)
==32161==
==32161== Conditional jump or move depends on uninitialised value(s)
==32161==   at 0x4E7AABD: vfprintf (in /usr/lib64/libc-2.17.so)
==32161==   by 0x4E84878: printf (in /usr/lib64/libc-2.17.so)
==32161==   by 0x4005CD: main (example2.c:11)
==32161==
==32161== Conditional jump or move depends on uninitialised value(s)
==32161==   at 0x4E7AB40: vfprintf (in /usr/lib64/libc-2.17.so)
==32161==   by 0x4E84878: printf (in /usr/lib64/libc-2.17.so)
==32161==   by 0x4005CD: main (example2.c:11)
==32161==
01234567815
==32161==
==32161== HEAP SUMMARY:
==32161==   in use at exit: 0 bytes in 0 blocks
==32161==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==32161==
==32161== All heap blocks were freed -- no leaks are possible
==32161==
==32161== For counts of detected and suppressed errors, rerun with: -v
==32161== Use --track-origins=yes to see where uninitialised values come from
==32161== ERROR SUMMARY: 8 errors from 6 contexts (suppressed: 0 from 0)
sh-4.2$
```



[front slash - Google Search - Mo...



[Valgrind Tutorial.pdf - Valgrind ...



Terminal



Pictures

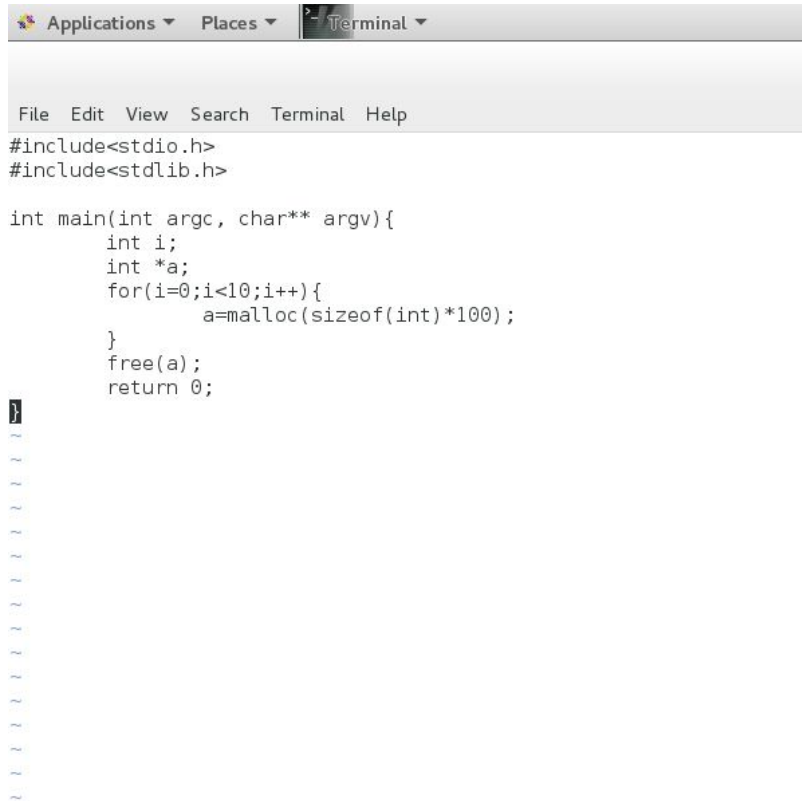
Example 2 contd.

```
Applications ▾ Places ▾ Terminal ▾

File Edit View Search Terminal Help

sh-4.2$ valgrind --track-origins=yes ./example
==4959== Memcheck, a memory error detector
==4959== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4959== Using Valgrind 3.11.0 and LibVEX; rerun with -h for copyright info
==4959== Command: ./example
==4959==
==4959== Conditional jump or move depends on uninitialised value(s)
==4959==    at 0x4E7A9F2: vfprintf (in /usr/lib64/libc-2.17.so)
==4959==    by 0x4E84878: printf (in /usr/lib64/libc-2.17.so)
==4959==    by 0x4085CD: main (example2.c:11)
==4959==    Uninitialised value was created by a stack allocation
==4959==    at 0x40857D: main (example2.c:4)
==4959==
==4959== Use of uninitialised value of size 8
==4959==    at 0x4E79EB8: _itoa_word (in /usr/lib64/libc-2.17.so)
==4959==    by 0x4E7AF05: vfprintf (in /usr/lib64/libc-2.17.so)
==4959==    by 0x4E84878: printf (in /usr/lib64/libc-2.17.so)
==4959==    by 0x4085CD: main (example2.c:11)
==4959==    Uninitialised value was created by a stack allocation
==4959==    at 0x40857D: main (example2.c:4)
==4959==
==4959== Conditional jump or move depends on uninitialised value(s)
==4959==    at 0x4E79E95: _itoa_word (in /usr/lib64/libc-2.17.so)
==4959==    by 0x4E7AF05: vfprintf (in /usr/lib64/libc-2.17.so)
==4959==    by 0x4E84878: printf (in /usr/lib64/libc-2.17.so)
==4959==    by 0x4085CD: main (example2.c:11)
==4959==    Uninitialised value was created by a stack allocation
==4959==    at 0x40857D: main (example2.c:4)
==4959==
==4959== Conditional jump or move depends on uninitialised value(s)
==4959==    at 0x4E7AF54: vfprintf (in /usr/lib64/libc-2.17.so)
==4959==    by 0x4E84878: printf (in /usr/lib64/libc-2.17.so)
==4959==    by 0x4085CD: main (example2.c:11)
==4959==    Uninitialised value was created by a stack allocation
==4959==    at 0x40857D: main (example2.c:4)
==4959==
==4959== Conditional jump or move depends on uninitialised value(s)
==4959==    at 0x4E7AABD: vfprintf (in /usr/lib64/libc-2.17.so)
==4959==    by 0x4E84878: printf (in /usr/lib64/libc-2.17.so)
==4959==    by 0x4085CD: main (example2.c:11)
==4959==    Uninitialised value was created by a stack allocation
==4959==    at 0x40857D: main (example2.c:4)
==4959==
==4959== Conditional jump or move depends on uninitialised value(s)
==4959==    at 0x4E7AB48: vfprintf (in /usr/lib64/libc-2.17.so)
==4959==    by 0x4E84878: printf (in /usr/lib64/libc-2.17.so)
==4959==    by 0x4085CD: main (example2.c:11)
==4959==    Uninitialised value was created by a stack allocation
==4959==    at 0x40857D: main (example2.c:4)
==4959==
01234567815
==4959==
==4959== HEAP SUMMARY:
==4959==    in use at exit: 0 bytes in 0 blocks
==4959==    total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==4959==
==4959== All heap blocks were freed -- no leaks are possible
==4959==
==4959== For counts of detected and suppressed errors, rerun with: -v
==4959== ERROR SUMMARY: 0 errors from 6 contexts (suppressed: 0 from 0)
sh-4.2$
```

Example 3



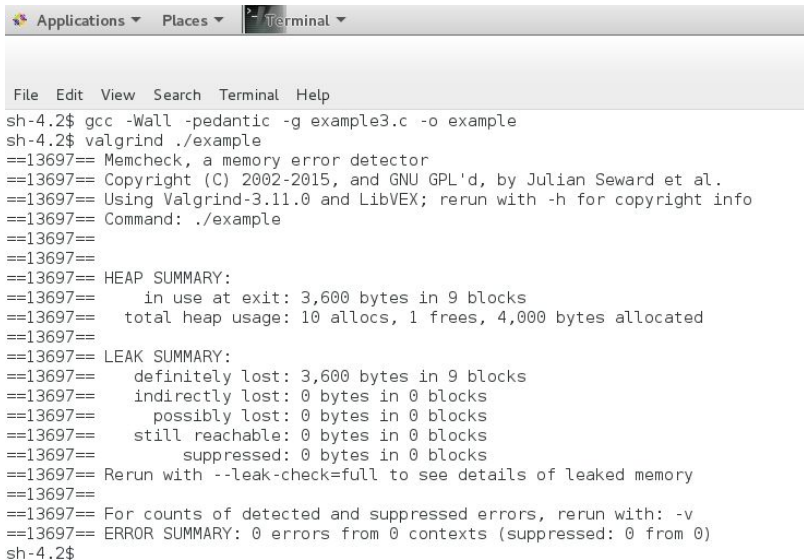
A screenshot of a terminal window with a menu bar at the top containing 'Applications', 'Places', and 'Terminal'. Below the menu bar is a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal displays the following C code:

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

int main(int argc, char** argv){
    int i;
    int *a;
    for(i=0;i<10;i++){
        a=malloc(sizeof(int)*100);
    }
    free(a);
    return 0;
}
```

Below the code, there are several tilde (~) characters, indicating a list of files or directories in the current directory.

Example 3



```
Applications ▾ Places ▾ Terminal ▾  
  
File Edit View Search Terminal Help  
sh-4.2$ gcc -Wall -pedantic -g example3.c -o example  
sh-4.2$ valgrind ./example  
==13697== Memcheck, a memory error detector  
==13697== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.  
==13697== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info  
==13697== Command: ./example  
==13697==  
==13697== HEAP SUMMARY:  
==13697==      in use at exit: 3,600 bytes in 9 blocks  
==13697==    total heap usage: 10 allocs, 1 frees, 4,000 bytes allocated  
==13697==  
==13697== LEAK SUMMARY:  
==13697==    definitely lost: 3,600 bytes in 9 blocks  
==13697==    indirectly lost: 0 bytes in 0 blocks  
==13697==    possibly lost: 0 bytes in 0 blocks  
==13697==    still reachable: 0 bytes in 0 blocks  
==13697==    suppressed: 0 bytes in 0 blocks  
==13697== Rerun with --leak-check=full to see details of leaked memory  
==13697==  
==13697== For counts of detected and suppressed errors, rerun with: -v  
==13697== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)  
sh-4.2$
```

Example 3

1. If you see leaks indicated as still reachable, this isn't a serious problem. These locations were probably still in use at the end of the program.
2. Leaks listed as 'definitely lost', 'indirectly lost', or 'possibly lost' should definitely be fixed.
3. E.g. for **indirectly lost**: Suppose you create a tree and free the root but don't free the other nodes.

Limitations of Valgrind

Consider the following program and its output:

```
Applications ▾ Places ▾ Terminal ▾  
  
File Edit View Search Terminal Help  
#include<stdio.h>  
#include<stdlib.h>  
  
int main(int argc, char** argv){  
    int i;  
    int x=0;  
    int a[10];  
    for(i=0;i<11;i++){  
        a[i]=i;  
    }  
  
    printf("%d ",x);  
  
    return 0;  
}
```

```
Applications ▾ Places ▾ Terminal ▾  
  
File Edit View Search Terminal Help  
sh-4.2$ vi example4.c  
sh-4.2$ gcc -Wall -pedantic -g example4.c -o example  
example4.c: In function 'main':  
example4.c:7:6: warning: variable 'a' set but not used [-Wunused-but-set-variable]  
    int a[10];  
    ^  
sh-4.2$ ./example  
10 sh-4.2$
```

What's Exactly Happening Here?



File Edit View Search Terminal Help

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

```
int main(int argc, char** argv){
    int i;
    int x=0;
    int a[10];
    for(i=0;i<11;i++){
        a[i]=i;
    }

    printf("Value of x is: %d ",x);
    printf("x is at: %p ",(void*)&x);
    printf("a[10]is at: %p ",(void*)&a[10]);
```

```
return 0;
}
```

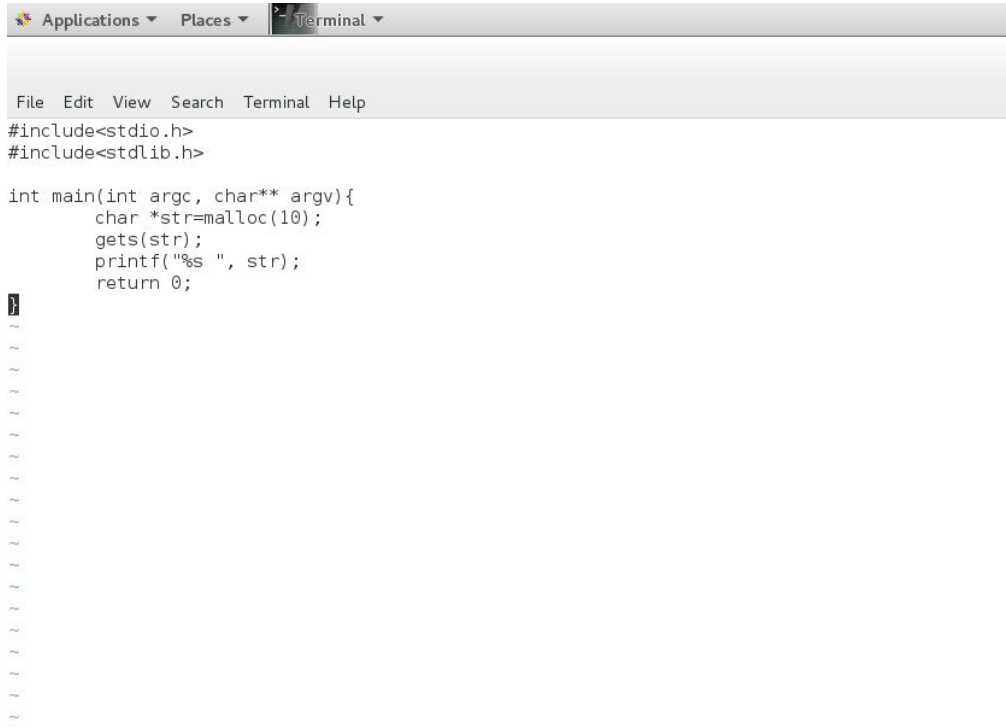
```
~
~
~
~
~
~
~
~
```



File Edit View Search Terminal Help

```
sh-4.2$ vi example4.c
sh-4.2$ gcc -Wall -pedantic -g example4.c -o example
sh-4.2$ ./example
Value of x is: 10 x is at: 0x7fff45a0d268 a[10]is at: 0x7fff45a0d268 sh-4.2$
```

Another Limitation



```
Applications ▾ Places ▾ Terminal ▾  
File Edit View Search Terminal Help  
#include<stdio.h>  
#include<stdlib.h>  
  
int main(int argc, char** argv){  
    char *str=malloc(10);  
    gets(str);  
    printf("%s ", str);  
    return 0;  
}
```


Lessons Learned

1. Valgrind does not do bound checking on static arrays i.e. those that are not allocated with malloc(). So, it is possible to have a program that is erroneous but does not generate any errors on Valgrind. Program 1 is an example of that.
2. Valgrind checks memory leaks dynamically. That is, it checks during the actual program run whether or not leaks occurred for that execution. So, it can happen that for some subset of inputs, the program works fine and Valgrind does not report errors. **But it is still buggy!**
3. If you want to be reasonably sure that a program is bug-free run it on a variety of inputs! Especially the corner cases.

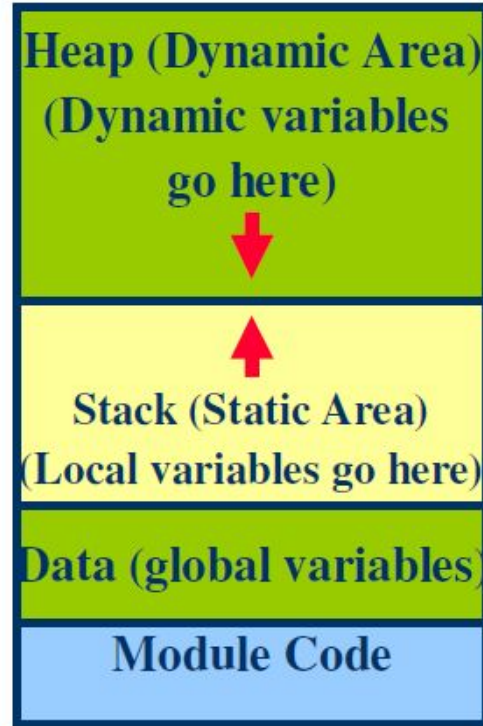
Memory Allocation Strategies

Almost every useful program has dynamic memory allocation. It has several benefits:

1. Allows for recursive procedures.
2. You can have data grow as a function of input size.

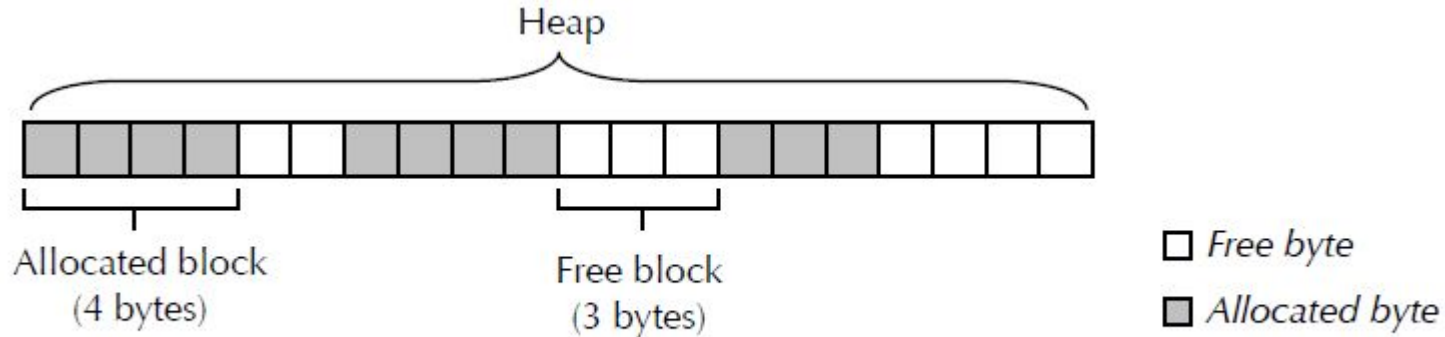
But, memory is not unlimited, we need to optimize the allocation as we will see in this lecture. But first we need to know how memory looks?

Logical View of a Process's Memory



The Heap

1. The heap is the region of a program's memory used for dynamic allocation.
2. Program can allocate and free blocks of memory within the heap.

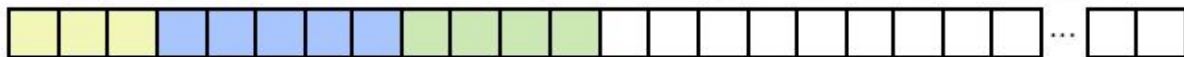


Free Blocks and Allocated Blocks

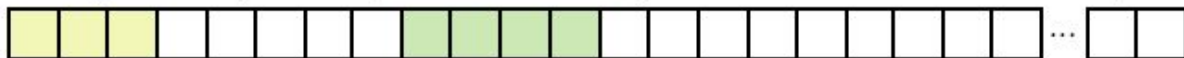
- Heap starts out as a single big “free block” of some fixed size (say a few MB)



- Program may request memory, which splits up the the free space.



- Program may free up some memory, which increases the free space

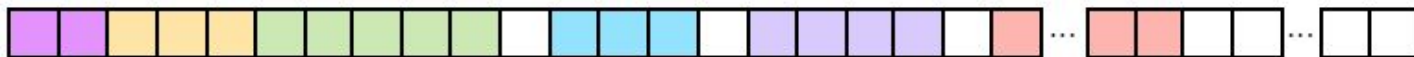


- Over time the heap will contain a mixture of free and allocated blocks.



- Heap may need to grow in size (but typically never shrinks)

- Program can grow the heap if it is too small to handle an allocation request.
- On UNIX, the `sbrk()` system call is used to expand the size of the heap.



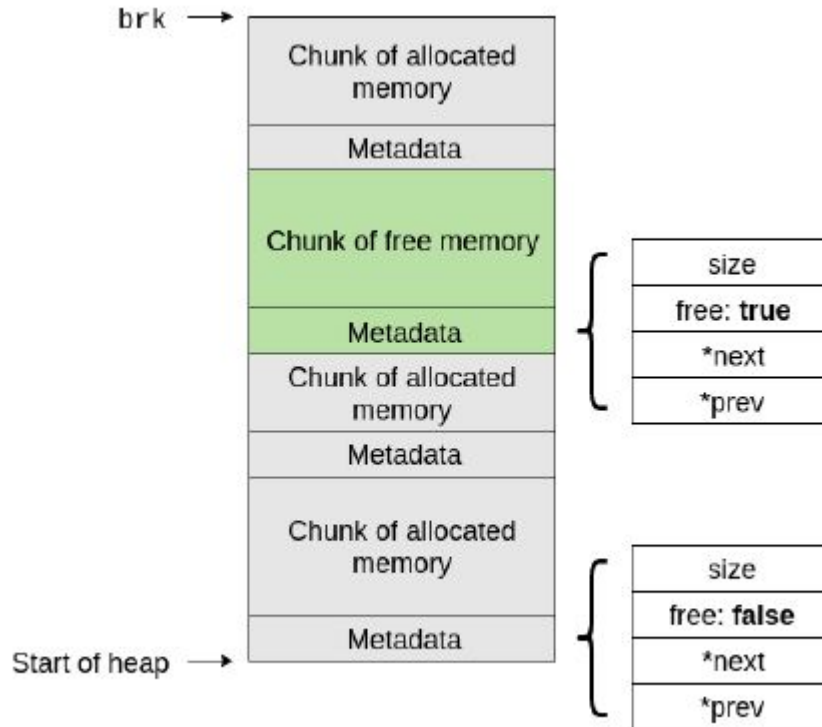
Metadata

Each block of memory is preceded by a header containing metadata. “Metadata” is basically data that describes other data.

For each block, we have the following metadata:

1. **prev, next:** Pointers to metadata describing the adjacent blocks.
2. **free:** A boolean describing whether or not this block is free.
3. **size:** the allocated size of the block of memory.

Metadata



Dynamic Memory Management

How do we manage allocating and freeing bytes on the heap? There are two broad schemes:

1. **Explicit Memory Management:** The application code is responsible for both **allocating** and **freeing** the memory. An example of this: **C!** This is what **malloc** and **free()** are for.
2. **Implicit Memory Management:** Application code can allocate memory, but **does not explicitly free memory**. There is **garbage collection** system to clean up memory objects no longer in use. E.g. **JAVA**

The Malloc() Package

- `#include <stdlib.h>`
- `void *malloc(size_t size)`
 - If successful:
 - Returns a pointer to a memory block of at least `size` bytes
 - If `size == 0`, returns `NULL`
 - If unsuccessful: returns `NULL`.
- `void free(void *p)`
 - Returns the block pointed at by `p` to pool of available memory
 - `p` must come from a previous call to `malloc` or `realloc`.
- `void *realloc(void *p, size_t size)`
 - Changes size of block `p` and returns pointer to new block.
 - Contents of new block unchanged up to min of old and new size.

So why is memory allocation hard?

1. You want to satisfy an arbitrary set of allocations and free's
2. Note that it is easy without free: Set a pointer to a big chunk of free memory (heap), and increment on each allocation.



So why is memory allocation hard?

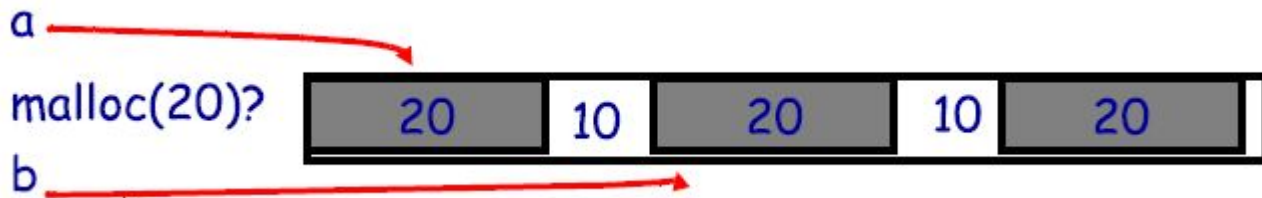
Problem: `free()` creates lots of holes. This is called fragmentation. So you have a lot of free space but how do you allocate it?



The job of a memory allocator

1. An allocator must keep track of which parts of memory are free and which are in use. Ideally, it should do it without wasting too much time or space.
2. However an allocator **cannot control the number and size of requested blocks**. Neither can it move around the allocated regions. This means bad placement decisions are permanent!

So, how do we fight defragmentation?



When does fragmentation happen?

1. Fragmentation is the inability to use memory that is free.
2. Two factors needed for fragmentation to happen:

Different timelines: If adjacent blocks die at different times, then we have fragmentation. Otherwise we don't.



When does defragmentation happen?

Different sizes: If all requests are of the same size, then there is no fragmentation



Some important decisions

1. Placement choice: Where in free memory should you put the requested block?

Note that you have the freedom to put the block anywhere. But, **ideally** you should put it in a manner that it doesn't cause fragmentation later (you need future knowledge, impossible).

Some Important Decisions

2. Split free blocks to satisfy smaller requests?

This can help fight fragmentation. But how do we choose the right block if we have the freedom to choose any block? Best fit! Choose the block with the smallest remainder.

3. Coalescing free blocks to yield larger blocks?

When should you coalesce. Coalescing at the right time can save you work!



Pathological Examples

- Given allocation of 7 20-byte chunks



- What's a bad stream of frees and then allocates?
-
- Given a 128-byte limit on malloced space
- What's a really bad combination of mallocs & frees?

Pathological Examples

- **Given allocation of 7 20-byte chunks**



- What's a bad stream of frees and then allocates?
- Free every other chunk, then alloc 21 bytes
- **Given a 128-byte limit on malloced space**
 - What's a really bad combination of mallocs & frees?

Pathological Examples

- **Given allocation of 7 20-byte chunks**



- What's a bad stream of frees and then allocates?
 - Free every other chunk, then alloc 21 bytes
- **Given a 128-byte limit on malloced space**
 - What's a really bad combination of mallocs & frees?
 - Malloc 128 1-byte chunks, free every other
 - Malloc 32 2-byte chunks, free every other (1- & 2-byte) chunk
 - Malloc 16 4-byte chunks, free every other chunk...

Two Allocators: Best Fit & First Fit

Best Fit

- **Strategy:** minimize fragmentation by allocating space from block that leaves smallest fragment

- Data structure: heap is a list of free blocks, each has a header holding block size and pointers to next



- Code: Search freelist for block closest in size to the request. (Exact match is ideal)
 - During free (usually) coalesce adjacent blocks
- **Problem: Sawdust**
 - Remainder so small that over time left with “sawdust” everywhere
 - Fortunately not a problem in practice

Best Fit Gone Wrong

- Simple bad case: allocate n, m ($n < m$) in alternating orders, free all the n s, then try to allocate an $n + 1$
- Example: start with 100 bytes of memory

- alloc 19, 21, 19, 21, 19



- free 19, 19, 19:



- alloc 20? Fails! (wasted space = 57 bytes)

First Fit

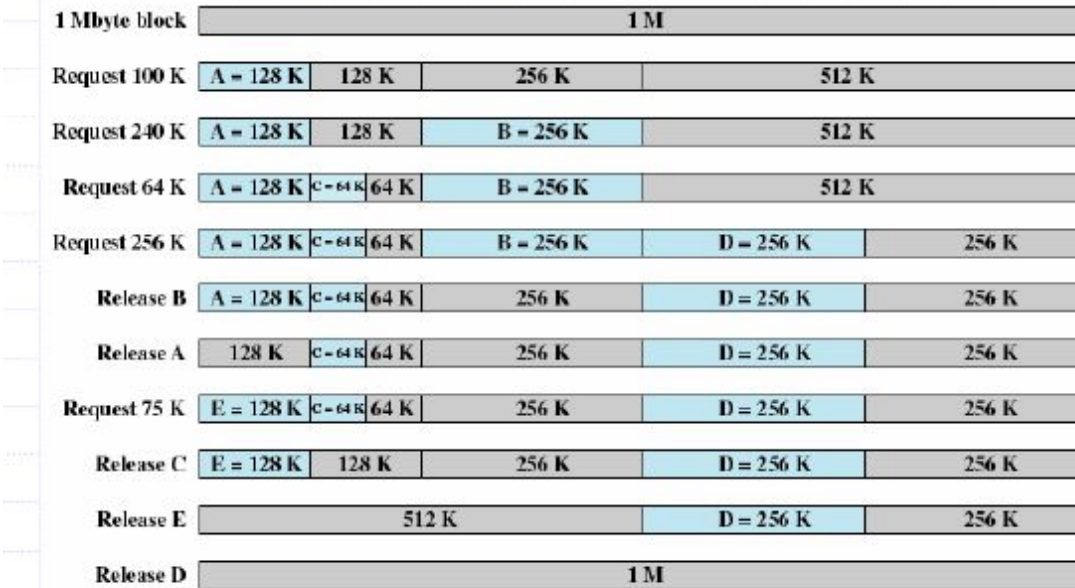
A20	20		108				
A15	20		15	93			
A10	20		15	10	83		
A25	20		15	10	25	58	
D20	20		15	10	25	58	
D10	20		15	10	25	58	
A8	8	12	15	10	25	58	
A30	8	12	15	10	25	30	28
D15	8	37			25	30	28
A15	8	15	22		25	30	28

Some worse ideas

1. **Next Fit:** This is exactly like first fit, except you start search from where you last left. This is faster than first fit, but leads to higher fragmentation,
2. **Worst Fit:** Fight sawdust by picking the largest block to satisfy the request.
Issues: You still have to search the entire list. It turns out be worse than best fit, and there are still fragmentation issues.
3. **Buddy System:** Explained in the next slide.

Buddy System: Example

Initial block size 1 MB; First request A is for 100 KB



Homework Questions

1. What is the difference between `strlen()` and `sizeof()` a string in C? Why? Write a code to find out!
2. Write a short program to read two lines of text, and concatenate them using `strcat`. Since `strcat` concatenates in-place, you'll have to make sure you have enough memory to hold the concatenated copy. For now, use a char array which is twice as big as either of the arrays you use for reading the two lines. Use `strcpy` to copy the first string to the destination array, and `strcat` to append the second one.

Homework Questions

3. Write the function `replace(char string[], char from[], char to[])` which finds the string `from` in the string `string` and replaces it with the string `to`. You may assume that `from` and `to` are the same length. For example, the code

```
char string[] = "recieve"; replace(string, "ie", "ei");
```

should change `string` to `"receive"`.

References:

1. <https://www.cs.columbia.edu/~junfeng/12sp-w4118/lectures/l18-alloc.pdf>
2. <https://www.cs.drexel.edu/~bls96/excerpt3.pdf>
3. <http://cs.ecs.baylor.edu/~donahoo/tools/valgrind/>