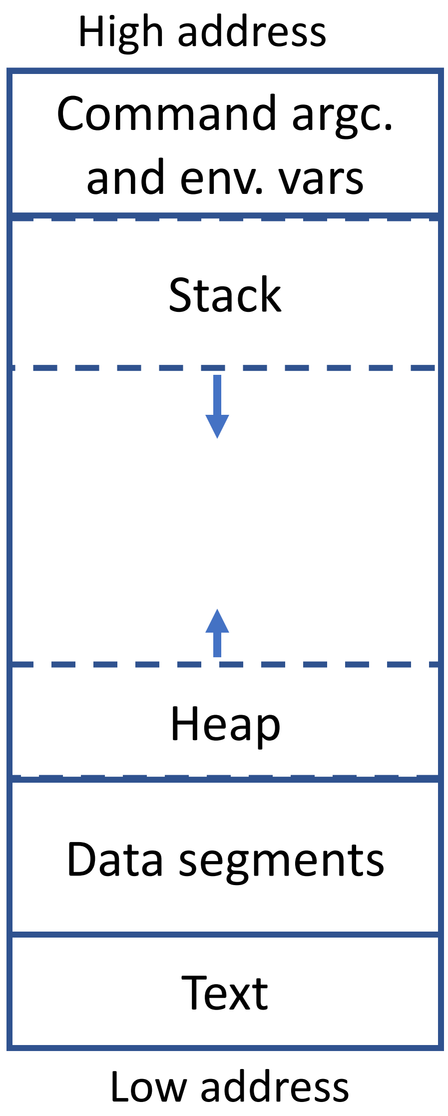
Dynamic Memory Management

Mohammed Bahja
School of Computer Science
University of Birmingham

Recap: Memory layout of C program

- Local variables use Stack ‘temporarily’. They are active only within their functions.
- Static variables are stored in the Data segments. They preserve their values.
- Global variables are stored in the Data segments. They are accessible to all functions of the C program.



[This figure is symbolic. Heap section need not start from the address just after data segment]

Use of Heap will be covered in Dynamic Memory Management.

Application scenario

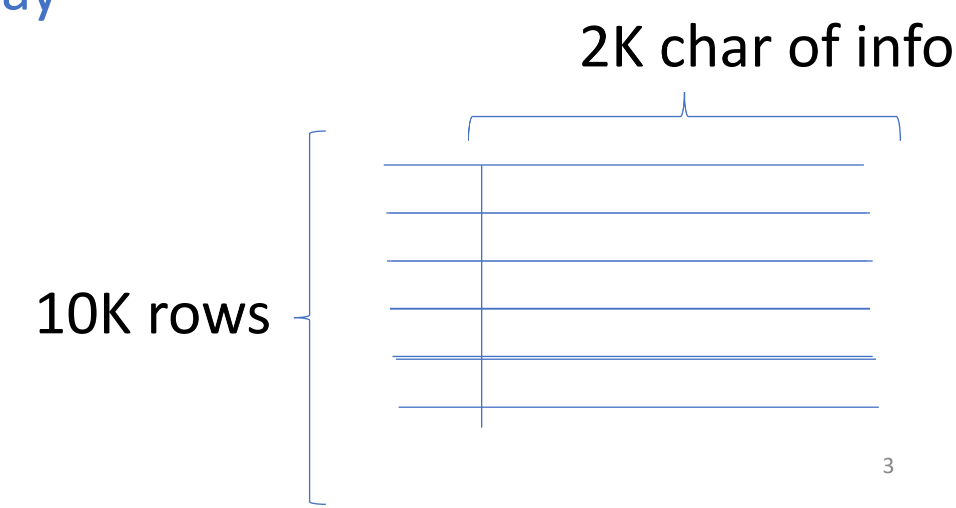
Consider an application where the volume of data is not known beforehand

Example: Store info of the people you meet during office hour

- The number of people is not known
- Each person may provide different amount of info

Solution 1: Allocate a large array

```c
char info[10000][2000];
```



Application scenario

Consider an application where the volume of data is not known beforehand

Example: Store info of the people you meet during office hour

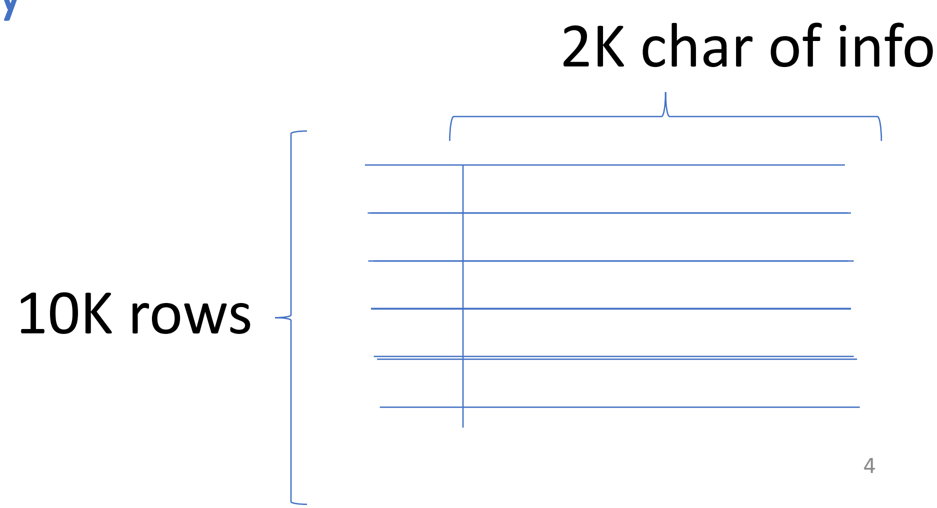
- The number of people is not known
- Each person may provide different amount of info

Solution 1: Allocate a large array

```
char info[10000][2000];
```

Problems:

1. May be wasteful
2. May be insufficient
3. Tiny computes can't afford large amount of space



Application scenario

Consider an application where the volume of data is not known beforehand

Example: Store info of the people you meet during office hour

- The number of people is not known
- Each person may provide different amount of info

Best solution: Allocate memory at **runtime** as per **demand**

Advantages: Optimal memory consumption depending on the current state of the application

Dynamic memory allocation in C

Dynamic memory allocation functions

C provides dynamic memory management functions in `stdlib.h`

- `malloc()`
  1. Allocates requested number of bytes of **contiguous** memory from the **Heap.**
  2. Returns a pointer to the first byte of the allocated space.
  3. If memory allocation fails, then `malloc()` returns NULL pointer.

```
Syntax:

T *p;
p = (T *) malloc(number of bytes);
```

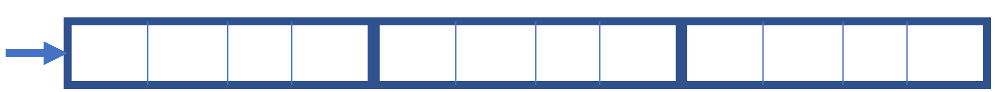
Dynamic memory allocation functions

C provides dynamic memory management functions in `stdlib.h`

- `malloc()`
  1. Allocates requested number of bytes of **contiguous** memory from the **Heap.**
  2. Returns a pointer to the first byte of the allocated space.
  3. If memory allocation fails, then `malloc()` returns NULL pointer.

```
Example: Allocate space for 3 int

int *p;
p = (int *) malloc(3*4); // int is 4 bytes
```

p


Allocates 12 bytes of contiguous memory in Heap.
p points to the first byte

Dynamic memory allocation functions

C provides dynamic memory management functions in `stdlib.h`

- `malloc()`
  1. Allocates requested number of bytes of **contiguous** memory from the **Heap.**
  2. Returns a pointer to the first byte of the allocated space.
  3. If memory allocation fails, then `malloc()` returns NULL pointer.

```
Example: Allocate space for 3 int

int *p;
p = (int *) malloc(3*sizeof(int));
// Allocates memory for 3 integers
```

Use `sizeof()` for convenience.

It returns the size of a `data_type` in bytes.

Dynamic memory allocation functions

C provides dynamic memory management functions in `stdlib.h`

- `malloc()`
  1. Allocates requested number of bytes of **contiguous** memory from the **Heap.**
  2. Returns a pointer to the first byte of the allocated space.
  3. If memory allocation fails, then `malloc()` returns NULL pointer.

```
Example: Allocate space for 3 int
int *p;
if((p= (int *) malloc(3*sizeof(int)))==NULL){
      printf("Allocation failed");
      exit(-1);
}
```

Good practice: check if a NULL pointer is returned by `malloc()`

Releasing allocated memory using free()

- Dynamically allocated block of memory can be released using free().
- After free(), the block of memory is returned to the Heap.

```
int *p;
// Block of memory is allocated
if((p= (int *) malloc(3*sizeof(int)))==NULL){
      printf("Allocation failed");
      exit(-1);
}

// [Some statements involving p]

// Block of memory pointed by p is released
free(p);
```

Example: Array of variable length

```
#include<stdio.h>
#include<stdlib.h>
int main(){
	int N, i;
	int *p;
	printf("Provide array size:");
	scanf("%d", &N);
	// Allocate memory in Heap for
	// array pointed by p
	if((p= (int *) malloc(N*sizeof(int)))==NULL){
		printf("Allocation failed");
		exit(-1);
	}
	printf("Provide %d integers\n", N);
	for(i=0; i<N; i++)
		scanf("%d", p+i);

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

Memory leak

If memory allocated using malloc() is not free()-ed, then the system will "leak memory"

- Block of memory is allocated, but not returned to Heap
- The program therefore grows larger over time
- In C, it is **your responsibility** to prevent memory leak

Example: Memory leak

```
int main(){
   int N, i,*p;

   while(1){
        printf("Provide array size:");
        scanf("%d", &N);
        if((p= (int *) malloc(N*sizeof(int)))==NULL){
              printf("Allocation failed");
              exit(-1);
        }
        printf("Provide %d integers\n", N);
        for(i=0; i<N; i++)
              scanf("%d", p+i);

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

Each iteration of while loop allocates memory. But that memory is never freed. This causes memory leak.

Valgrind: a tool for memory leak detection

- We will use the valgrind tool to detect memory leaks
- Quick start info: http://valgrind.org/docs/manual/quick-start.html

Steps to follow:

1. Compile your C code using gcc
2. Then use valgrind to run the compiled code:
   `valgrind --leak-check=full ./a.out`

Example: valgrind output

```
int main()
{
        int *pt = malloc(1*sizeof(int));
        scanf("%d", pt);
        printf("%d\n", *pt);
        return 0;
}
```

Output of valgrind --leak-check=full ./a.out

```
==1057== Memcheck, a memory error detector
==1057== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1057== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==1057== Command: ./a.out
==1057==
```

... continued

```
==1057== HEAP SUMMARY:
==1057==     in use at exit: 4 bytes in 1 blocks
==1057==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==1057==
==1057== LEAK SUMMARY:
==1057==    definitely lost: 0 bytes in 0 blocks
==1057==    indirectly lost: 0 bytes in 0 blocks
==1057==      possibly lost: 0 bytes in 0 blocks
==1057==    still reachable: 4 bytes in 1 blocks
==1057==         suppressed: 0 bytes in 0 blocks
```

Valgrind reports a memory leak of 4 bytes. ☹

Example: valgrind output

```c
int main()
{
        int *pt = malloc(1*sizeof(int));
        scanf("%d", pt);
        printf("%d\n", *pt);
        free(pt);
        return 0;
}
```

Allocated memory is freed here

Output of valgrind --leak-check=full ./a.out

```
==2308== Memcheck, a memory error detector
==2308== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2308== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2308== Command: ./a.out
==2308==
5
5
==2308==
==2308== HEAP SUMMARY:
==2308==     in use at exit: 0 bytes in 0 blocks
==2308==   total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==2308==
==2308== All heap blocks were freed -- no leaks are possible
==2308==
```

No memory leak reported! ☺

Consequences of memory leak

- Memory leaks slowdown system performance by reducing the amount of available memory.

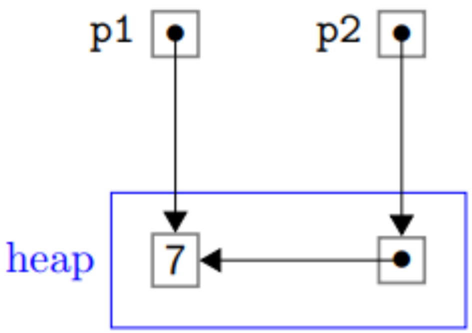
- In modern operating systems, memory used by an application is released when the application terminates
  - If a program (with leak) runs for a short duration, then no serious problem
  - Will be a serious problem when a leaking program runs for long duration

- Memory leak will cause serious issues on resource-constrained embedded devices

Overall, your program should not contain memory leaks

Another memory leak example(1)

```c
int main(){
   int *p1, **p2;
   p1 = malloc(sizeof(int));
   *p1=7;
   p2 = malloc(sizeof(int*));
   *p2=p1;
   return 0;
}
```

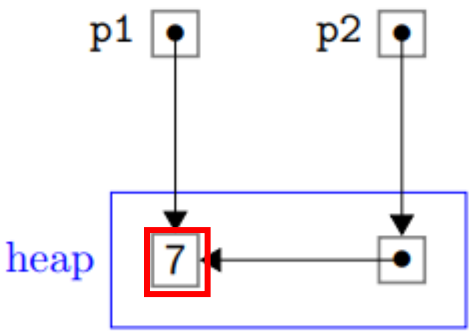


p2 is a pointer to a pointer variable

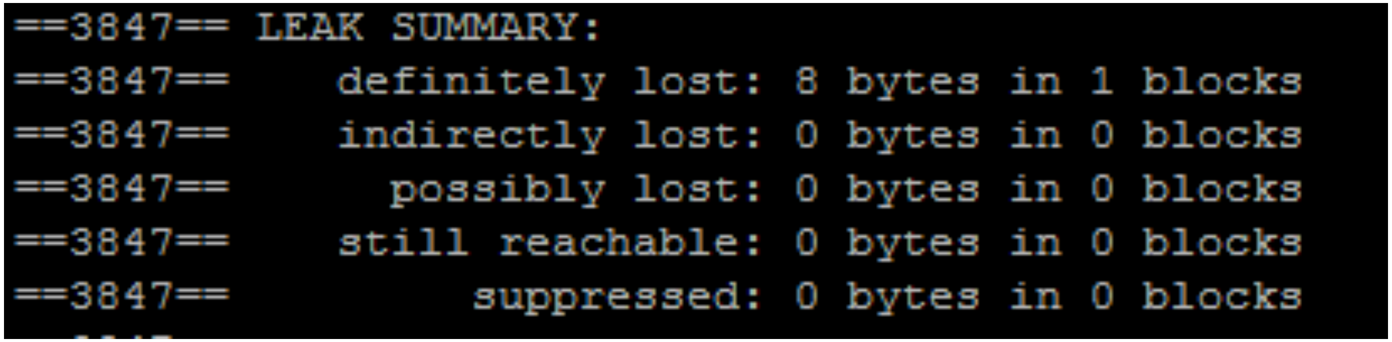
```
==3600== HEAP SUMMARY:
==3600==     in use at exit: 12 bytes in 2 blocks
==3600==   total heap usage: 2 allocs, 0 frees, 12 bytes allocated
==3600==
==3600== 12 (8 direct, 4 indirect) bytes in 1 blocks are definitely lost in loss record 2 of 2
==3600==    at 0x4C29BC3: malloc (vg_replace_malloc.c:299)
==3600==    by 0x400546: main (in /users/cosic/ssinharo/dynamicmem/a.out)
==3600==
==3600== LEAK SUMMARY:
==3600==    definitely lost: 8 bytes in 1 blocks
==3600==    indirectly lost: 4 bytes in 1 blocks
==3600==      possibly lost: 0 bytes in 0 blocks
==3600==    still reachable: 0 bytes in 0 blocks
==3600==         suppressed: 0 bytes in 0 blocks
==3600==
```

Another memory leak example(2)

```c
int main(){
   int *p1, **p2;
   p1 = malloc(sizeof(int));
   *p1=7;
   p2 = malloc(sizeof(int*));
   *p2=p1;
   free(p1);
   return 0;
}
```



p2 is a pointer to a pointer variable

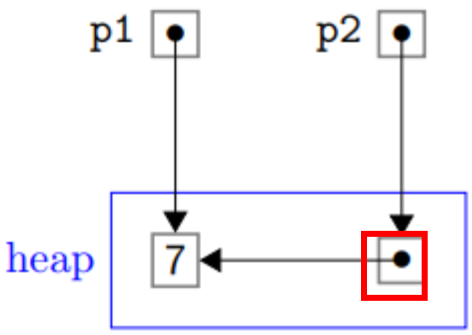


```
==3847== LEAK SUMMARY:
==3847==    definitely lost: 8 bytes in 1 blocks
==3847==    indirectly lost: 0 bytes in 0 blocks
==3847==      possibly lost: 0 bytes in 0 blocks
==3847==    still reachable: 0 bytes in 0 blocks
==3847==         suppressed: 0 bytes in 0 blocks
```

Still 8 bytes are leaked!

Another memory leak example(3)

```c
int main(){
   int *p1, **p2;
   p1 = malloc(sizeof(int));
   *p1=7;
   p2 = malloc(sizeof(int*));
   *p2=p1;
   free(p2);
   return 0;
}
```



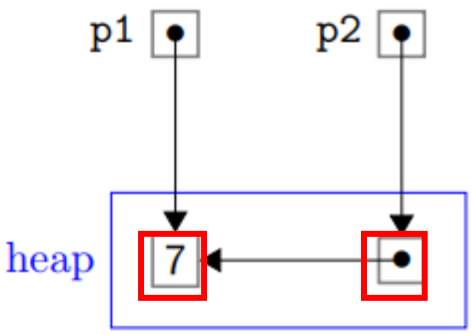
p2 is a pointer to a pointer variable

```
==6994== LEAK SUMMARY:
==6994==    definitely lost: 4 bytes in 1 blocks
==6994==    indirectly lost: 0 bytes in 0 blocks
==6994==      possibly lost: 0 bytes in 0 blocks
==6994==    still reachable: 0 bytes in 0 blocks
==6994==         suppressed: 0 bytes in 0 blocks
```

4 bytes are leaked!

Another memory leak example(4)

```
int main(){
   int *p1, **p2;
   p1 = malloc(sizeof(int));
   *p1=7;
   p2 = malloc(sizeof(int*));
   *p2=p1;
   free(p1);
   free(p2);
   return 0;
}
```



p2 is a pointer to a pointer variable

No memory leak!

Accessing memory after free( )

- After free(p), the memory is no longer owned by the program.

  - Some other program may be using the memory! Data is lost

- Re-accessing the memory pointed by p causes undefined behaviour

```
int main(){
      int sum, *p;
      p = (int *) malloc(4*sizeof(int));
      // [some code here]
      free(p);
      sum = sum + *p; //Use after free issue
      // [some code here]
}
```

Memory pointed by p was freed and then used.
Program will cause undefined behaviour.
Valgrind reports memory error.

Don’t free something that did not come from malloc()

```
foo(int a){
   int b;
   int c[]={1,2,3};
   free(&a);
   free(&b);
   free(c);
}
```

- Only, dynamically created objects are stored in heap
- gcc gives warnings: “attempt to free a non-heap object”
- However, program crashes with segmentation fault

Valgrind reports memory errors

Double free problem

- Double free errors occur when free() is called more than once with the same memory address as an argument.

```
int main(){
        char *p1=malloc(1);
        char *p2=malloc(1);
        free(p1);
        free(p1);
}
```

- When free() is called twice with the same argument, the program's memory management data structures become corrupted

- Consequences: Program malfunction including security vulnerabilities

At runtime, modern OS detects double-free operation.
Program is aborted.

How to think about double free

```c
int main(){
      char *p1=malloc(1);
      char *p2=malloc(1);
      free(p1);
      free(p1);
}
```

Heap manager

Available blocks
…
…
p1
p1

- Every time free() is called, the address is added in the list of available memory blocks
- The last address added to the list can by the next malloc() call
- Since p1 is present twice in the list, next two memory allocations will have the same address (which is a problem)
- Valgrind reports memory error