COMP281
Principles of C and memory management

lecture 6

Dr. Frans Oliehoek
Department of Computer Science
University of Liverpool

Last Time

- Pointers
  - A pointer is denoted with a *
  - * is used to dereference a pointer
  - The address of variable can be given by &variable
- Arrays & Pointers
  - "*pointer* ***arithmetic*** and *array* ***indexing*** are equivalent in C, *pointers* and *arrays* are different."
- Start Heap Memory allocation
  - malloc, free

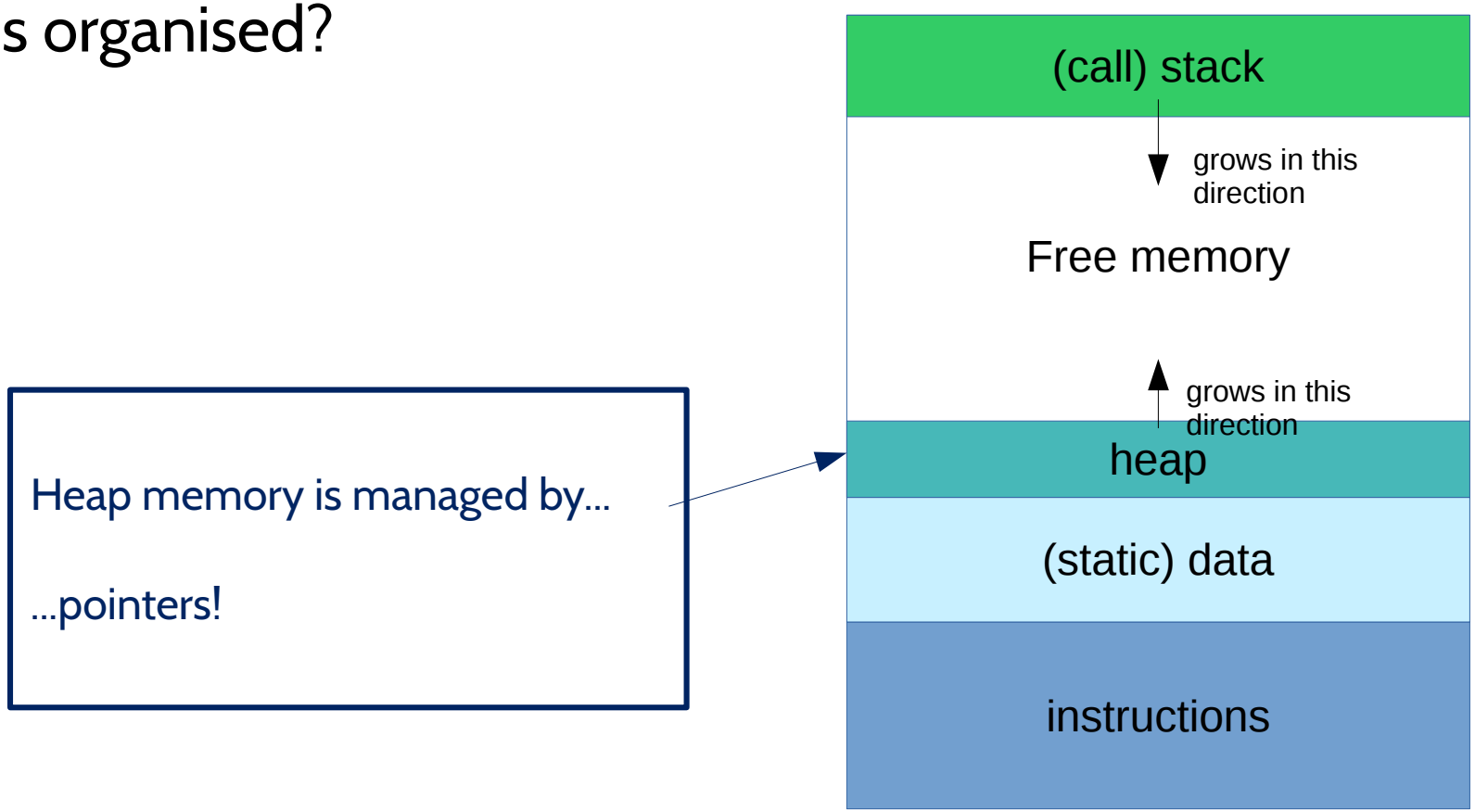
Today

- Wrap up malloc/free
- Debugging heap memory allocation: `valgrind`
- Multi-dimensional arrays

The Heap – Part II

Refresher: memory organization

- Remember the way memory is organised?



malloc example

```
#define NUM_ARRAYS 300
#define ARRAYSIZE 100000000 //100 million ints - 0.4 gb
void processArray(int * array)
{
     int j=0, sum=0;
     for (j=0; j < ARRAYSIZE; j++)
          sum += array[j];
     printf("%d\n",sum);
}
main()
{
     int* store_numbers;
     int i,j;
     for (i=0; i < NUM_ARRAYS;i++)
     {
          store_numbers = malloc(ARRAYSIZE*sizeof(int));
          store_numbers[0] = i;
          processArray(store_numbers);
          free(store_numbers);
     }
}
```

File:e20.c

malloc is defined in "stdlib.h"

← free the memory...!

Memory allocation functions

- `void* malloc(size)`
  - void* is a 'void' pointer
  - it is a pointer to memory, but we don't know what is stored there
  - e.g., `int* a=malloc(sizeof(int)*10);`
  - allocation may fail: then `NULL` pointer returned
- `void* realloc(old_pointer, newsize)`
  - old_pointer must point to a previously allocated area.
  - contents remain unchanged up min(newsize, oldsize).
  - If expanded, the contents of the new part are undefined.
  - After realloc, **old_pointer may become invalid**.
- `void* calloc(number, size);`
  - Guaranteed to **zero initialise** the memory
- `void free(pointer);`
  - Deallocates the space allocated by malloc(), calloc() or realloc().

Memory allocation functions

- `void* malloc(size)`
  - void* is a 'void' pointer
  - it is a pointer to memory, but we don't know what is stored there
  - e.g., `int* a=malloc(sizeof(int)*10);`
  - allocation may fail: then `NULL` pointer returned
- `void* realloc(old_pointer, newsize)`
  - old_pointer must point to a previously allocated area.
  - contents remain unchanged up min(newsize, oldsize).
  - If expanded, the contents of the new part are undefined.
  - After realloc, **old_pointer may b**ecome invalid.
- `void* calloc(number`
  - Guaranteed to **zero initialise** the
- `void free(pointer);`
  - Deallocates the space allocated by malloc(), calloc() or realloc().

but how about those 'void' pointers?

Casting types

- Casting (as in Java) converts one type of variable to another type
- In C, most primitive types can be moved without being cast
  - Even to smaller types
  - This would usually cause a compile error in Java
  - It is usually considered good practice to explicitly cast type conversions

```
main()
{
     int intvar;
     char charvar;
     long long longvar =  (1024*1024+1) ;
     longvar = longvar * 4096;
     charvar = (char) longvar;
     intvar =  (int)longvar;

     printf("%d %d %lld \n",charvar,intvar,longvar);
}
```

File: casting.c

Casting Example

```
double db = 7 / 5;
printf("%f \n",db);
```

- prints 1.000000
- (7/5 is treated as an integer calculation)

```
double db = (double) (7 / 5);
printf("%f \n",db);
```

- Is exactly the same
- (7/5) is converted to a double after the calculation

```
double db = (double) 7 / 5;
printf("%f \n",db);
```

- prints 1.400000
- as 7 is treated as a double, hence 7/5 is no longer an integer calculation

```
double db = (double) (7 / 5.0);
printf("%f \n",db);
```

- prints 1.400000
- as 5 is treated as floating point, hence 7/5.0 is not an integer calculation

Casting `void*`

- malloc returns a void*...
- void* cannot be dereferenced!
  - this will not work:

```
int main(void)
{
    void* p=NULL;
    p = malloc(2*sizeof(int));
    printf("%d", *p);
}
```

void_ptr.c

- To dereference it: change to a different type of pointer!
  - So, cast it!
  - `int* a = (int*) malloc(sizeof(int)*10);`
- Why no problem before?
  - `int* a = malloc(sizeof(int)*10);`

Casting General Pointers

- Pointers that are not void may also be explicitly cast to another type
  - e.g.,

```
float number = 1.0;
float* number_pointer = &number;
int* int_pointer = (int*) number_pointer;
*int_pointer += 30;
printf("%f \n",number);
```

File:pointer_mixing.c

- This is a dangerous thing to do…

Arrays, Pointers & Malloc

- Arrays and pointers are mostly interchangeable…
- An **array** is a single, preallocated chunk of contiguous elements (all of the same type), fixed in size and location.
- A **pointer** is a reference to any data element (of a particular type) anywhere. A pointer must be assigned to point to space allocated elsewhere, but it can be reassigned (**and the space, if derived from malloc, can be resized**) at any time. A pointer can point to an array, **and can simulate (along with malloc) a dynamically allocated array**.
  - a pointer to a block of memory assigned by `malloc` is frequently treated (and can be referenced using `[]`) exactly as if it were a true array.

Arrays, Pointers & Malloc

- Arrays and pointers are mostly interchangeable…
- An **array** is a single, preallocated chunk of contiguous elements (all of the same type), fixed in size and location
- A **pointer** is a reference to any
  anywhere. A pointer must be a
  elsewhere, but it can be reassig
  from malloc, can be resized) a
  array, **and can simulate (along
  allocated array**.
  - a pointer to a block of memory assigned by malloc is frequently treated (and can be referenced using `[]`) exactly as if it were a true array.

Possible due to equivalence between pointer arithmetic and array indexing

Heap or Stack?

- Where should you store data?
  - How much data?
  - How long do you need it for?
- Small amounts of data, that are only needed temporarily, should go on the stack (as local variables).
  - They are faster to use (no malloc to call, no pointer to dereference).
  - In some architectures, they may also be in faster memory (such as a pre-defined data cache).
- If you need to keep the data around for a while, or if it is a large amount, it needs to go on the heap.
  - But it must be freed after use

Heap: common errors

- Not checking for allocation failures:
  - Memory allocation is not guaranteed to succeed. If there's no check for successful allocation implemented, this usually leads to a crash of the program or the entire system.
- Memory leaks:
  - Failure to deallocate memory using free leads to buildup of memory that is non-reusable memory, which is no longer used by the program. This wastes memory resources and can lead to allocation failures when these resources are exhausted.
- Logical errors:
  - All allocations must follow the same pattern:
  - **allocation** using malloc, **usage** to store data, **deallocation** using free.
  - Failures to adhere to this pattern, such as memory usage after a call to free or before a call to malloc, calling free twice, etc., usually leads to a crash of the program

Example – 1

- Problem 1057

Title	Reverse input
Description	Reverse the input of a sequence of integers
Input	an integer *n*, followed by *n* integers
Output	the *n* integers in reverse order
Sample Input	3 1 2 3
Sample Output	3 2 1
Hint	*n* may be very large (up to 28 bits), only allocate the memory actually required

Example – 1

```
#include <stdio.h>
#include <stdlib.h>
main()
{

  int i, array_size;
  int* array;
  scanf("%d",&array_size); // read the input size
  array = malloc(sizeof(int) * array_size); // allocate space for it

  for  (i =0; i < array_size; i++)  // read all the input
    scanf("%d",&array[i]);

  for  (i =0; i < array_size; i++) // reverse the order
    printf("%d ",array[array_size - 1 - i]);

  free(array);
}
```

File:reverse_malloc.c

Review Heap Memory

- Memory allocation
  - Small, local, variables should go on the stack
  - Otherwise, you need to allocate heap memory at runtime
  - Use malloc(size) to allocate memory
  - malloc returns a pointer, or NULL if it can't be allocated
    - may need to **cast**
  - don't forget to free the memory after you have finished with it

Heap debugging: Valgrind

Using valgrind

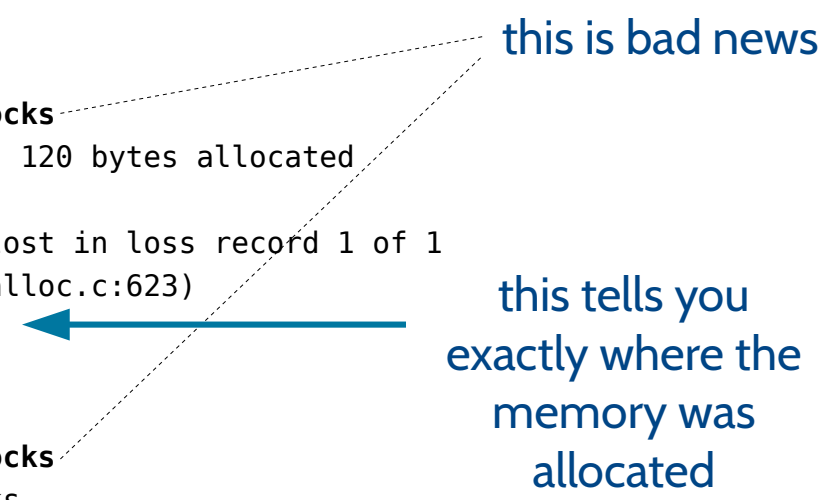
- Valgrind can check for memory leaks!
- Compile with debug symbols (-g) and turn of optimization (-O0) for accurate reporting:
  - gcc -o mem_leak -g -O0 -Wall mem_leak.c
  - valgrind --tool=memcheck --leak-check=full --show-leak-kinds=all ./mem_leak

  (demo)
- If you don't like all that typing... create a **Makefile**!

  (demo)
- More info:
  - memcheck: http://valgrind.org/docs/manual/quick-start.html
  - other tools: http://valgrind.org/info/tools.html

Reference: Interpreting Valgrind

```
valgrind --tool=memcheck  --leak-check=full  --show-leak-kinds=all   ./mem_leak
==10927== Memcheck, a memory error detector
==10927== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==10927== Command: ./mem_leak
<OUTPUT OF PROGRAM>
==10927==
==10927== HEAP SUMMARY:
==10927==     in use at exit: 120 bytes in 3 blocks
==10927==   total heap usage: 3 allocs, 0 frees, 120 bytes allocated
==10927==
==10927== 120 bytes in 3 blocks are definitely lost in loss record 1 of 1
==10927==    at 0x4C2AD10: calloc (vg_replace_malloc.c:623)
==10927==    by 0x4005C1: main (mem_leak.c:25)
==10927==
==10927== LEAK SUMMARY:
==10927==    definitely lost: 120 bytes in 3 blocks
==10927==    indirectly lost: 0 bytes in 0 blocks
==10927==      possibly lost: 0 bytes in 0 blocks
==10927==    still reachable: 0 bytes in 0 blocks
==10927==         suppressed: 0 bytes in 0 blocks
==10927==
==10927== For counts of detected and suppressed errors, rerun with: -v
==10927== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```



Reference: Makefile

- contents of "Makefile"

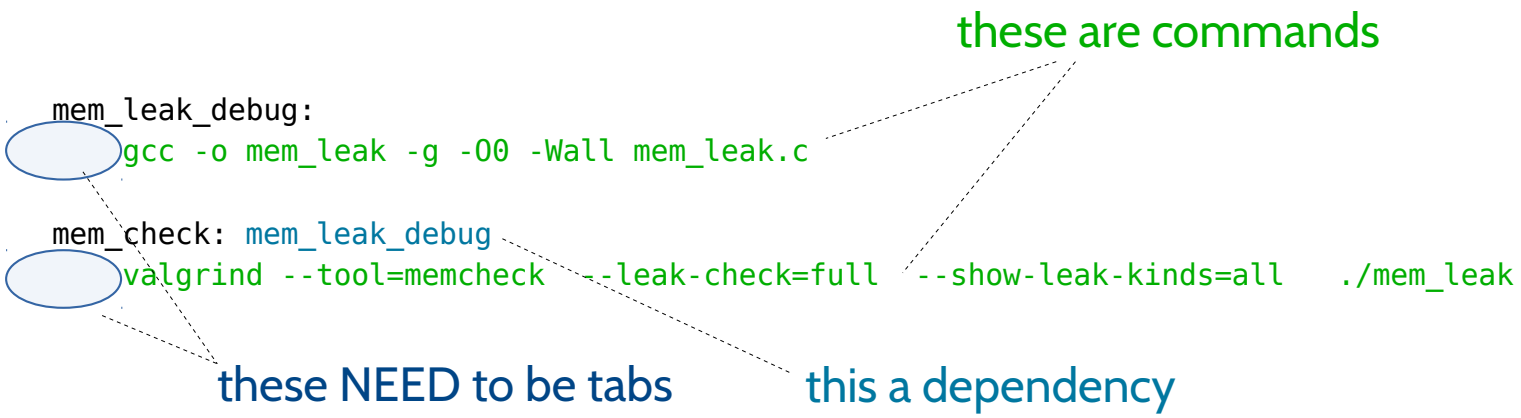
```
mem_leak_debug:
    gcc -o mem_leak -g -O0 -Wall mem_leak.c

mem_check: mem_leak_debug
    valgrind --tool=memcheck  --leak-check=full  --show-leak-kinds=all   ./mem_leak
```

- In terminal

  `$ make mem_check`
- Many other things makefiles can do!
  - ask google about "makefile tutorial"

Reference: Makefile

- contents of "Makefile"



- In terminal

  `$ make mem_check`

- Many other things makefiles can do!
  - ask google about "makefile tutorial"

Review

- Heap memory allocation
  - use malloc etc.
  - and don't forget to free!
- Heap debugging... valgrind!
  - (yes, we all forget to free ;-))
- You now know how to...
  - ...deal with variable size memory requirements
  - ...decide between heap and stack
  - ...trace memory leaks
  - ...avoid typing long commands at the command line