COMP281
Principles of C and memory management

Lecture 5

Dr. Frans Oliehoek
Department of Computer Science
University of Liverpool

Last Week

- All kinds of C syntax
- OJ and Debugging
  - in short:
    - compilation error → read the message, try gcc on linux
    - presentation error → replicate OJ by piping input
    - wrong answer → think about the assumptions your code makes
    - runtime error → gdb
    - it works on my computer → try gcc on linux
  - always: read the messages!
- a first few words about pointers...

This week

- Pointers, pointers, pointers!
  - declaration and dereferencing
  - pointers and arrays
  - Heap memory (and pointers!)

Pointers

Pointers

- Program code and data are stored in memory

- Every location (e.g., each byte) in memory has an **address**
  - This is just a number telling the processor how to find it.

- In C we can **access** and **manipulate these addresses** directly
  - (similar to assembly language)
  - the variables that store such addresses are called **pointers**
  - Declared using '*'
- E.g.,:

```
int a = 42;
int * ptr_to_int = NULL;
ptr_to_int = &a;
```

- If you want to pass a function a large amount of data, it is much easier to just **pass a pointer to that data**

Pointers

- A **pointer** is a variable that contains the memory address of some item
- `*type` denotes 'a pointer to type'
- `&` denotes 'the address of'

```
int* pointer;  //the variable will contain a pointer to an integer
```

- We can use this, as follows:

```
int variableA;
int* pointer = &variableA;
```

- Pointer now contains the address of variableA…
  - so, how do we access its contents?
  - We also use the * notation for this:

```
int value = *pointer;
*pointer = 8;
```

Pointer – example

```
void set_to_10(int* ptr)
{
    *ptr = 10;
}
main()
{
    int v = 5;
    int* pointer = &v;
    int a[5] = {1,2,3,4,5};

    printf("v = %d \n", v);
    set_to_10(pointer);
    printf("v = %d \n", v);

    printf("a[0] = %d \n", a[0]);
    set_to_10(a);
    printf("a[0] = %d \n", a[0]);
    return(0);
}
```

Pointer – example

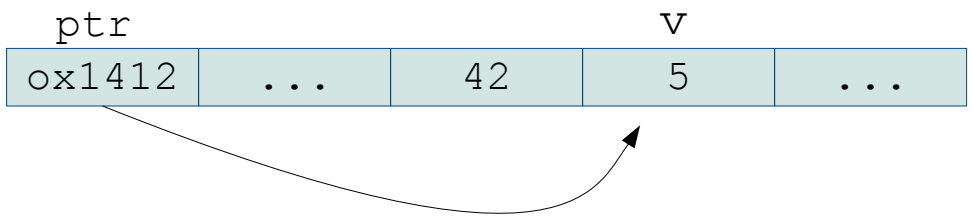
```
void set_to_10(int* ptr)
{
    *ptr = 10;
}
main()
{
    int v = 5;
    int* pointer = &v;
    int a[5] = {1,2,3,4,5};

    printf("v = %d \n", v);
    set_to_10(pointer);
    printf("v = %d \n", v);

    printf("a[0] = %d \n", a[0]);
    set_to_10(a);
    printf("a[0] = %d \n", a[0]);
    return(0);
}
```

pointer is **dereferenced**

i.e., the **value** of the address to which the pointer points is set (or retrieved)



Pointer – example

```
void set_to_10(int* ptr)
{
    *ptr = 10;
}
main()
{
    int v = 5;
    int* pointer = &v;
    int a[5] = {1,2,3,4,5};

    printf("v = %d \n", v);
    set_to_10(pointer);
    printf("v = %d \n", v);

    printf("a[0] = %d \n", a[0]);
    set_to_10(a);
    printf("a[0] = %d \n", a[0]);
    return(0);
}
```

Output.:
v = 5
v = 10
a[0] = 1
a[0] = 10

this is called "(C style) pass by reference"

Pointer – example

```
void set_to_10(int* ptr)
{
    *ptr = 10;
}
main()
{
    int v = 5;
    int* pointer = &v;
    int a[5] = {1,2,3,4,5};

    printf("v = %d \n", v);
    set_to_10(pointer);
    printf("v = %d \n", v);

    printf("a[0] = %d \n", a[0]);
    set_to_10(a);
    printf("a[0] = %d \n", a[0]);
    return(0);
}
```

Output.:
v = 5
v = 10
a[0] = 1
a[0] = 10

this is called "(C style) pass by reference"

- an array's name **is a pointer**
  - points to first element
  - i.e. `a == &a[0]`

Pointer / References

- In Java, you refer to objects by *'reference'*; same idea!
- except that a **C pointer refers to the actual memory address** used by the system:
  - can do pointer arithmetic (moving the pointer)
  - Java reference cannot be used in that way

- 'reference' tends to mean something slightly different in different languages...
  http://stackoverflow.com/questions/40480/is-java-pass-by-reference-or-pass-by-value

Moving the pointer

- You can also add to a pointer to move where it points to
  - When you add to a pointer, the number of bytes it moves depends on the type of the pointer
  - e.g.:

```
int array[3] = {1,2,3};
int* pointer = array;  //<-could also write &array[0]
pointer++;             //<- increments the pointer value to the next piece of memory
```

  - Actually moves the pointer along 4 bytes (1 int is usually 4 bytes)

```
double array[3] = {1.0, 2.2, 3.6};
double* pointer = array;
pointer++;             //<-moves pointer 'sizeof(double)' bytes along (typically 8 bytes)
```

- Watch out for precedence:

```
*pointer++;            //better to write *(pointer++);
```

vs

```
(*pointer)++;
```

Moving the pointer

- This is particularly useful when looping in arrays

```
int array [] = {2,3,4};
int size = 3;
int* pointer = array;
for (i = 0; i < size; i++)
{
  sum = sum + *pointer;
  pointer++;  //move to the next element in the array
}
```

File: moving_pointer.c

Multiple return arguments

- Passing by reference (i.e., via pointer) can be useful if your function should do more than just return a value

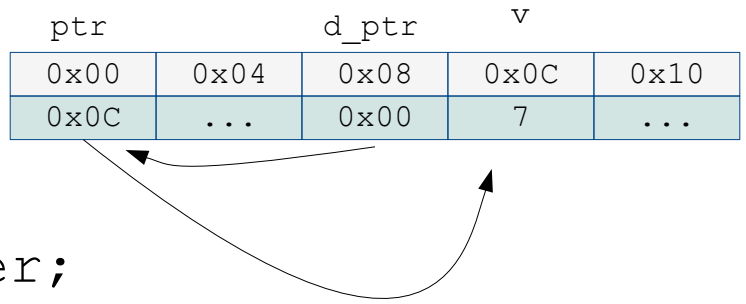
- E.g.,: how to compute the area and circumference of a circle at the same time…?

`multiple_return_vals.c`

Pointers to Pointers

- Pointers can also point to other pointers

```
int value = 7;
int* pointer = &value;
int** double_pointer = &pointer;
```



- Each deference needs a * operator

```
printf("%d \n",**double_pointer );
```

- Ways to help the reader of the code to understand:

```
int new_value = *(* double_pointer);
```

- or

```
int* new_pointer = *double_pointer;
int new_value = *new_pointer;
```

Pointers Summary

- pointers are variables
  - whose value is a memory address
  - whose type depends on the type to which it points

- Usage:
  - Declared using '*'

```
int my_int = 42;
int * my_ptr = &my_int;
```

  - **Dereferenced** *also* using '*'

```
int b = *my_ptr; //b == 42
```

make sure you understand the difference between these two!

(see suggested readings)

- Used to implement (c-style) pass by reference
  - for arrays, or (upcoming lectures) large data structures...!

Pointers vs Arrays

Array names can be used as pointers

- Remember, we saw that an array's name **is a synonym for a pointer to its first element**
  - i.e., an array name `a` is equivalent to `&a[0]`
  - http://c-faq.com/aryptr/aryptrequiv.html

```
void set_to_10(int* ptr)
{
    *ptr = 10;
}
main()
{
    int a[5] = {1,2,3,4,5};
    set_to_10(a);
    printf("a[0] = %d \n", a[0]);
    return(0);
}
```

- this allows for a unified treatment of arrays and pointers
- "**equivalence of pointers and arrays**"
  - slightly misleading name!

Arrays and Pointers

- An array is a (pre-defined) block of memory containing a number of elements
  - elements are indexed by []
- A pointer points to a block of memory (which may also contain a number of elements)
  - a pointer can also be indexed by []
- Hence you can easily point to the beginning of an array

```
main()
{
    int array[] ={1,2,3,4,5,6};
    int* pointer;

    pointer = array;
    printf("%d \n",array[2]);
    printf("%d \n",pointer[2]);

    pointer = &array[0];
    printf("%d \n",pointer[2]);
}
```

Arrays and Pointers

- An array is a (pre-defined) block of memory containing a number of elements
  - elements are indexed by []
- A pointer points to a block of memory (which may also contain a number of elements)
  - a pointer can also be indexed by []
- Hence you can easily point to the beginning of an array

```
main()
{
     int array[] ={1,2,3,4,5,6};
     int* pointer;

     pointer = array;
     printf("%d \n",array[2]);
     printf("%d \n",pointer[2]);

     pointer = &array[0];
     printf("%d \n",pointer[2]);
}
```

[] notation is **defined** as

```
pointer[i] == *(pointer + i)
```

for arrays this is the same:

```
array[i] == *(array + i)
```

but additionally we have:

```
== *( &array[0] + i)
```

Passing Arrays

- If you pass an array as a parameter, you actually pass a pointer to the array
  - `int get_average(int numbers[])` becomes `int get_average(int* numbers)`

```
int get_average(int numbers[])
{
  int sum = 0;
  int count = 0;
  int i = 0;
  while (numbers[i] !=0)
  {
    count++;
    sum += numbers[i++];
  }
  return sum/count;
}
main()
{
  int numbers []= {1,3,5,6,8,9,0};
  int average=get_average(numbers);
  printf("%d\n",average);
}
```

Passing Arrays

- If you pass an array as a parameter, you actually pass a pointer to the array
  - `int get_average(int numbers[])` becomes `int get_average(int* numbers)`

```
int get_average(int numbers[])
{
  int sum = 0;
  int count = 0;
  int i = 0;
  while (numbers[i] !=0)
  {
    count++;
    sum += numbers[i++];
  }
  return sum/count;
}
main()
{
  int numbers []= {1,3,5,6,8,9,0};
  int average=get_average(numbers);
  printf("%d\n",average);
}
```

Note that C has no way of knowing the size of the array...!

- here: encoded by 0
- in general: also pass in the size!

Arrays and Pointers

- Notation can largely be mixed:

```
int get_average(int* numbers)
{
   int sum = 0;
   int count = 0;
   while (numbers[0] !=0)
   {
       count++;
       sum += *numbers;
       numbers++;
   }
   return sum/count;
}
```

file:passing_arrays.c

- numbers[0] and *numbers both access the same array element
- You can also declare a pointer as const
  - the pointer itself can be amended, but the data it points to cannot be written to!

Arrays

- So-called **'equivalence of arrays and pointers':** arrays and pointers often seem interchangeable.
  - but think about what you mean,
  - particularly for multiple dimensions, this can get messy...!

- Don't forget
  - array: a single, preallocated chunk of contiguous elements
  - pointer: a reference to any data element (of a particular type) anywhere.
    - must be assigned to point to space allocated elsewhere,
    - but it can be reassigned

- Proper interpretation of 'equivalence':

"*pointer **arithmetic*** and *array **indexing*** are equivalent in C,
pointers and *arrays* are different."

– http://c-faq.com/~scs/cgi-bin/faqcat.cgi?sec=aryptr

Arrays

- So-called **'equivalence of arrays and pointers':** arrays and pointers often seem interchangeable.
  - but think about what you mean,
  - particularly for multiple dimensions, this can get messy…!

- Don't forget
  - array: a single, preallocated chunk of contiguous elements
  - pointer: a reference to any data element of appropriate type
    - must be assigned to point to space allocated elsewhere,
    - but it can be reassigned

`pointer[i]` **is defined as** `*(pointer + i)`

- Proper interpretation of 'equivalence':

"*pointer* ***arithmetic*** and *array* ***indexing*** are equivalent in C, *pointers* and *arrays* are different."

– http://c-faq.com/~scs/cgi-bin/faqcat.cgi?sec=aryptr

Arrays vs Pointers – Summary

- an array's name **is a pointer**, that points to first element
  - `a` **is defined as** `&a[0]`

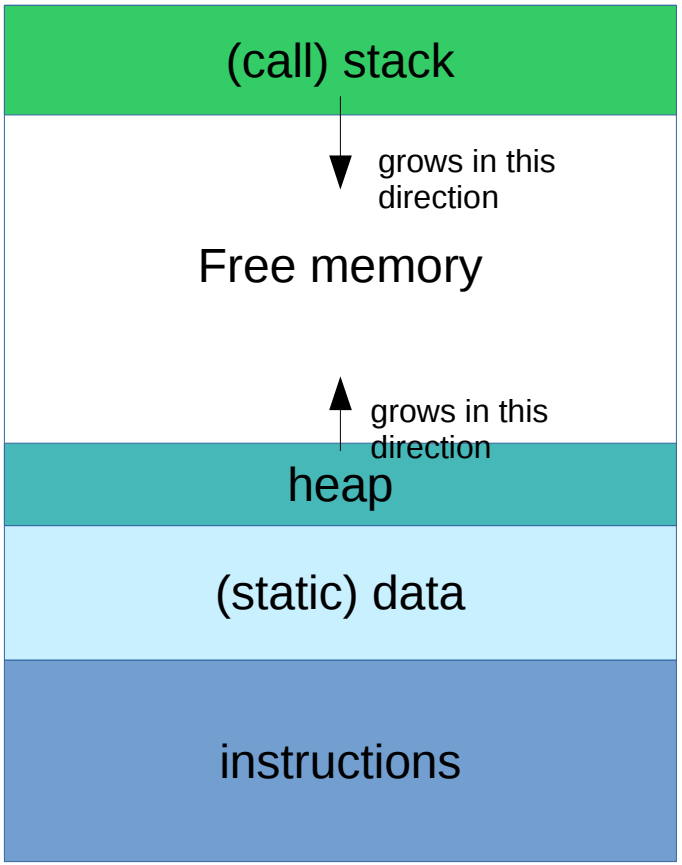
- bracket notation:
  - `pointer[i]` **is defined as** `*(pointer + i)`

"*pointer* ***arithmetic*** and *array* ***indexing*** are equivalent in C, *pointers* and *arrays* are different."

The Heap – Part I

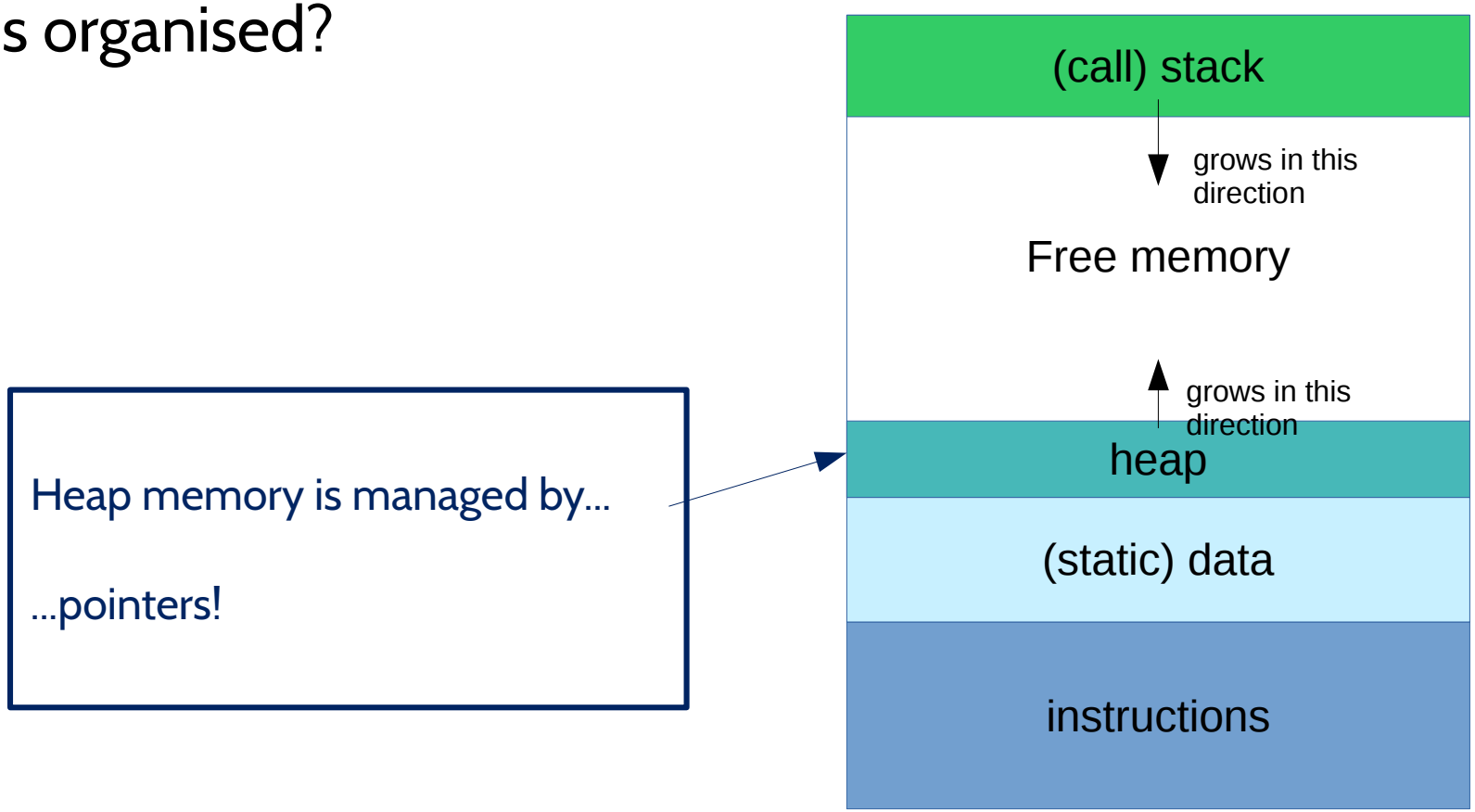
Refresher: memory organization

- Remember the way memory is organised?



Refresher: memory organization

- Remember the way memory is organised?



Heap memory allocation: malloc/free

- Memory is allocated on request
- This comes from the memory '**heap**'
  - requested from the operating system
  - has nothing to do with stack memory

- If you allocate lots of memory at once, there is less available for other programs

- When you have finished using memory, you should 'free' it again
  - otherwise cannot re-use that memory
  - Unused memory is NOT collected for you
    - No 'garbage collection'
  - Not freeing memory is often called a '**memory leak**'
    - every time you run the code, a small piece of memory is 'lost'
    - eventually, you may use all the system's memory, and the program will fail

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);

     }
}
```

Problem?

File: malloc_free.c

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: malloc_free.c

← free the memory...!

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: malloc_free.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().

Review

- Pointers
  - A pointer is **declared** with a `*`
  - `*` is used to **dereference**
  - The address of variable can be given by `&variable`

suggested reading for this week

Again (!):

- pointers and arrays: Lu Ch4 / K&R Ch5 / Bradley Ch1,3

New:

- heap — Lu Ch8,9
- valgrind/makefiles — Lu Ch5
- multi-dim. arrays — Lu Ch8,9, K&R Ch5

- Pointers and Arrays
  - definition of the "[]" notation
  - "*pointer* ***arithmetic*** and *array* ***indexing*** are equivalent in C, *pointers* and *arrays* are different."

- Heap 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
  - don't forget to free the memory after you have finished with it