COMP281
Principles of C and memory management

lecture 7

Dr. Frans Oliehoek
Department of Computer Science
University of Liverpool

Last Time

- Using heap memory: `malloc`
  - and `free`!
- Tracing down memory leaks with `valgrind`
- Using `Makefile`s

Today

- Multidimensional Arrays

- Custom data types (structs, etc.)

Multi-dimensional Arrays
and
Arrays of Arrays

Multi-dimensional arrays

- Like Java, C supports multi-dimensional arrays. E.g.:

```
int grid_locations[4][4];
int x,y;
for(x=0; x<4; x++)
  for(y=0; y<4; y++)
    grid_locations[x][y]=0;
```

Multi-dimensional arrays

- You can also have arrays of pointers.
  - eg. `int* matrix[16];`

```
main()
{
   int* matrix[16];
   printf("%d",matrix[0][0]);
}
main()
{
   int matrix[16][16];
   printf("%d",matrix[0][0]);
}
```

Question: What do the following do?

multidimensional_arr.c

multi-dim. array vs. “array of arrays”– 1

- A *multi-dimensional array* is **NOT** exactly the same as *an array of arrays*…!

- If we define a 2D array
  `int matrix[2][4] = {{1,2,3,4},{5,6,7,8}};`
  - matrix contains all 16 separate ints, in a single block of memory
  - matrix[0][1] is simply element number 1 (the 2nd element) in the memory
  - matrix[1][1] is simply element number 5.

- You **cannot** change an entire row
  - e.g., cannot do: `matrix[0] = new_row;`

- you **can** reference it as a whole row,
  - e.g. `int* rowpointer = matrix[0];`

multi-dim. array vs. "array of arrays"– 2

- In Java, multidimensional arrays are actually **arrays of arrays** (also "jagged arrays")
- Can do that in C too using an **array of pointers**:

```
main()
{
    int row1[4] = {1,2,3,4};
    int row2[4] = {5,6,7,8};

    //an array of 'int*' (i.e., an array of pointers)
    int* matrix[2] = {row1, row2};

    printf("matrix[1,2]: %d\n",matrix[1][2]);
}
```

- Clearly, we can still reference a whole row, e.g.: `int* rowpointer = matrix[0];`
- but now, we **can also replace an entire row**, e.g.:

```
int new_row[2] = {1,2}; //new row of different size
matrix[0] = new_row;    //no problem, but keep track of #cols
                        //yourself!
```

multi-dim. array vs. "array of arrays"– 2

- In Java, multidimensional arrays are actually **arrays of arrays** (also "jagged arrays")
- Can do that in C too using an **array of pointers**:

```
main()
{
    int row1[4] = {1,2,3,4};
    int row2[4] = {5,6,7,8};

    //an array of 'int*' (i.e., an array of pointers)
    int* matrix[2] = {row1, row2};

    printf("matrix[1,2]: %d\n",matrix[1][2]);
}
```

- Clearly, we can still reference a whole row, e.g.: `int* rowpointer = matrix[0];`
- but now, we **can also replace an entire row**, e.g.:

```
int new_row[2] = {1,2}; //new row of different size
matrix[0] = new_row;    //no problem, but keep track of #cols
                        //yourself!
```

Note:

this 'multidimensional array simulation' is **not** guaranteed to be in a contiguous part of memory...

- the int* in 'matrix' are stored next to each other...
- ...but they may point to quite different places!
  (depends on where compiler made the space for row1 and row 2)

malloc for arrays of arrays

- You can now dynamically allocate two dimensional ‘arrays’
  - actually, a pointer to a block of pointers which each point to a block of ints
  - (a ‘block’ is a one-dimensional array, just a consecutive list in memory)

```
int** allocateArray(int rows,int columns)
{
  int **array; int i;
  array =  malloc(rows*sizeof(int*));
  for(i=0;i<rows;i++)
  {
    array[i] = malloc(columns*sizeof(int));
  }
  return array;
}
main ()
{
  int** array = allocateArray(10,10);
  array[5][5] = 1;
  printf("%d \n",array[5][5]);
  printf("%d \n",array[15][5]);
}
```

malloc for arrays of arrays

- You can now dynamically allocate two dimensional 'arrays'
  - actually, a pointer to a block of pointers which each point to a block of ints
  - (a 'block' is a one-dimensional array, just a consecutive list in memory)

```
int** allocateArray(int rows,int columns)
{
  int **array; int i;
  array =  malloc(rows*sizeof(int*));
  for(i=0;i<rows;i++)
  {
    array[i] = malloc(columns*sizeof(int));
  }
  return array;
}
main ()
{
  int** array = allocateArray(10,10);
  array[5][5] = 1;
  printf("%d \n",array[5][5]);
  printf("%d \n",array[15][5]);
}
```

Flexible:

- allocate memory as needed
- could store different #elements in each row

Drawbacks:

- could be slower than multidimensional array
  - overhead malloc
  - consecutive pointer dereferences
- no 'navigational support'
  - you are responsible of keeping track of #rows, and # elements

Pointers to multi-dimensional arrays

- Multi-dimensional arrays: compiler knows their size
  - gives 'navigation support' for pointers
  - but syntax gets a bit more complex...

```
int matrix_a[2][4] = {{1,2,3,4},{5,6,7,8}};

//this defines a pointer to length-4 int arrays:
int (*row_ptr)[4] = matrix_a;
printf("%d", row_ptr[1][2]); //1 row ahead, element '2'
row_ptr++; //jump to next row
```

`multidim_pointer_arithmetic.c`

- Nice tool: cdecl
  - type natural language, gets transformed to C declaration

cdecl> declare test as pointer to array 4 of array 3 of int
> int (*test)[4][3]

Arrays of Strings

- A string is an array of chars: `char str [] = "hello";`
- So an array of strings, is an array of arrays of chars…

```
int** allocateArray(int rows, int columns)
{
  char **array; int i;
  array =  malloc(rows*sizeof(char*));
  for(i=0;i<rows;i++)
    array[i] = malloc(columns*sizeof(char));
  return array;
}
main ()
{
  //5 strings of length 10 (9!)
  char** array = allocateArray(5,10);
  strcpy(array[3],"test123");
  printf("%s", array[3]);
}
```

Example – 2

- Problem 1058

Title	Reverse all input
Description	Receive a number of sequences of integers Print all the sequences in reverse order and also reverse the ordering of each sequence
Input	an integer *n*, followed by *n* lines each line contains an integer m followed by m integers
Output	n lines, in reverse order. each line contains m integers, also in reverse order
Sample Input	2 3 1 2 3 4 5 6 7 8
Sample Output	8 7 6 5 3 2 1
Hint	only allocate the memory actually required

Example – 2

```
main()
{
    int num_rows,i;
    scanf("%d",&num_rows);

    int** arrays = malloc(sizeof(int*) * num_rows);

    for ( i=0; i < num_rows; i++)
        get_input(&arrays[i]);

    for ( i=1; i <= num_rows; i++)
        do_output(arrays[num_rows - i]);
}
```

Example – 2

```
get_input(int** arrays) //<- receive pointer to an array pointer (&arrays[i])
{
  int i, array_size;
  int* array;              // array that will store the next row
  int* use_pointer;
  scanf("%d",&array_size); // find out how big this array is

  // allocate enough for the whole array, plus 1 to store the size:
  array = malloc(sizeof(int) * (array_size+1));

  // store the address of the array so the main function knows where it is:
  *arrays = array;
  //store the size of the array at position 0:
  *array = array_size;
  // start storing at position 1
  use_pointer = array+1;
  for  (i =0; i < array_size; i++)
    scanf("%d",use_pointer++);
}
```

Example – 2

```
void do_output(int* array)
{
    int i;
    //get the size of the array (remember *array == array[0])
    int array_size = *array;
    array++;            // we can move a pointer along to
                        // ignore the first entry

    //print the array in reverse order:
    for  (i =0; i < array_size; i++)
    {
        printf("%d ",array[array_size -1 - i]);
    }
    printf("\n");
}
```

Initialising Arrays

- You can initialise arrays when they are declared

```
int array[]={0,1,2,3};
```

  - This sets the size automatically

- You can also declare the size, and partially initialise
  - remaining elements initialized to 0

```
int array[100]={8};
int array[100]={8,1,2};
```

Initialising Multi-dimensional Arrays

- You can also initialise 2-dimensional arrays
  - but compiler needs to know about 'inner' dimensions

```
int array[2][2] = { { 0,1}, {2,3} };
int array[][2] = { { 0,1}, {2,3},{4,5} };
```

- You can also do this with strings
  - But you need to specify the (max.) size of the strings!

```
char wordarray[][101]={ "one word",
                        "second word"};
```

Custom Data Types: structs

Data types

- We have only used primitive data types
  - ints, chars and pointers, etc.
- This is obviously inconvenient for more advanced structures
  - e.g., date – may be stored as year, day, month
  - We could store this as an array - e.g., `int date[3];`
  - But what if we want to mix types?
- We need a composite data structure - a **`struct`**
  - define as:

```
struct structurename
{
//member definitions go here
};
```

  - and reference as:

```
struct structurename
```

struct example

- Members of a struct are referenced by `structurename.member`

- Note the initialisation
  - The fields are allocated in the same order
  - This is not a 'constructor'; no other code is called
  - As this is a local variable, all data is created **on the stack**

```
struct address
{
   int number;
   char street[100];
};

main()
{
   struct address myAddress =
     {108,"Anywhere Road"};
   printf("%d %s\n",
     myAddress.number,
     myAddress.street
   );
}
```

Pointers to Structs

- When accessed via a pointer, the following syntax is used: `pointer->member` (instead of `variable.member`)

```
struct address {
  int number;
  char street[1000000];
};
struct address myAddress={108,"Anywhere Road"};

void nextDoor(struct address *any_address) {
  any_address->number += 2;
}

main() {
  struct address processAddress = myAddress;
  nextDoor(&processAddress);
  printf("%d %s \n",processAddress.number, processAddress.street);
}
```

- As only a pointer is passed, the large struct is not copied onto the stack

Pointers to Structs

- When accessed via a pointer, the following syntax is used: `pointer->member` (instead of `variable.member`)

```
struct address {
  int number;
  char street[1000000];
};
struct address myAddress={108,"Anywhere Road"};

void nextDoor(struct address *any_address) {
  any_address->number += 2;
}

main() {
  struct address processAddress = myAddress;
  nextDoor(&processAddress);
  printf("%d %s \n",processAddress.number, processAddress.street);
}
```

any_adress->number == (*any_address).number

- As only a pointer is passed, the large struct is not copied onto the stack

More Structs

- A struct can **contain another struct**

```
struct address {
  int number;
  char street[100];
};
struct record {
  char name[100];
  struct address home_address;
};
struct record a_record = { "Tony",{108,"Anywhere Road"}};
main(){
  printf("%d %s \n",record.home_address.number, record.name);
}
```

- Structs and variables may have the same name, but this can become confusing...

More Structs

- A struct can **contain another struct**

```
struct address {
  int number;
  char street[100];
};
struct record {
  char name[100];
  struct address home_address;
};
struct record record = { "Tony",{108,"Anywhere Road"}};
main(){
  printf("%d %s \n",record.home_address.number, record.name);
}
```

- Structs and variables may have the same name, but this can become confusing…

structs pointing to structs

- A struct can **contain a pointer to another struct**

```
struct address
{
  int number;
  char street[100];
};

struct record
{
  char name[100];
  struct address *address;
};
```

```
struct address
  my_address={108,"Anywhere Road"};
struct address
  my_new_address={12,"Anywhere Lane"};
struct record record =
  { "Tony",&myAddress};

main()
{
  record.address = &my_new_address;
  printf("%d %s \n",
    record.address->number,
    record.name
  );
}
```

File:struct_2.c

structs pointing to structs

- A struct can **contain a pointer to another struct**

```
struct address
{
  int number;
  char street[100];
};

struct record
{
  char name[100];
  struct address *address;
};
```

```
struct address
  my_address={108,"Anywhere Road"};
struct address
  my_new_address={12,"Anywhere Lane"};
struct record record =
  { "Tony",&myAddress};

main()
{
  record.address = &my_new_address;
  printf("%d %s \n",
    record.address->number,
    record.name
  );
}
```

File:struct_2.c

Also possible: partially initialisation

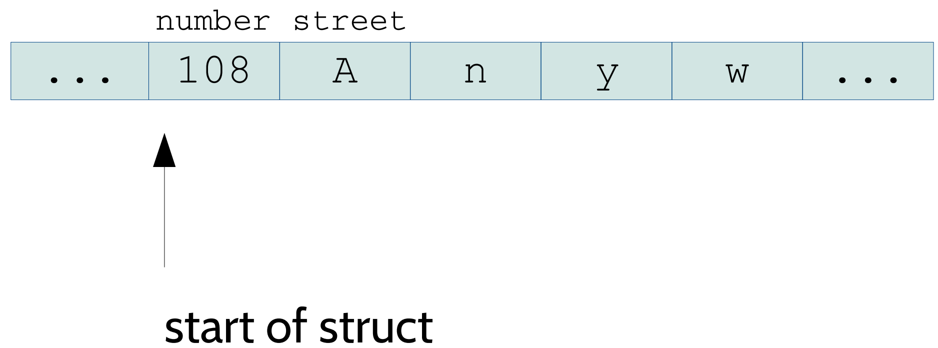
```
struct record record =
{ "Tony"};
```

The rest will follow the usual rules for uninitialised memory

structs and memory

- A ‘struct’ is stored as a **continuous section** of memory

```
struct address
{
  int number;
  char street[100];
};
```



- Imagine that where you use
  `struct address myaddress;`
  you are actually inserting:
  ```
  int myaddress.number;
  char myaddress.street[100];
  ```

struct and memory – example

```
#include<stdio.h>
struct address
{       int number;
        char street[10000000];
};
void useStruct(struct address anaddress)
{ printf("%d \n",anaddress.number); }

struct address myAddress={108,"Anywhere Road"};

main()
{ useStruct(myAddress); }
```

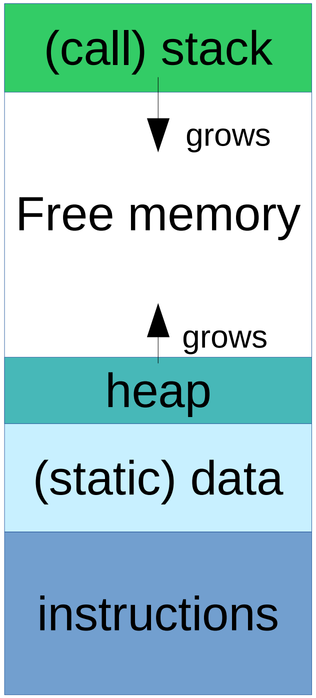
File:q_struct_mem.c

Q:what happens here?

structs and memory

Cf. Java:
-Primitive variables represent the data
-Object variable always represent a reference

- In C, we have **data** and **pointers**
  - If a variable is not explicitly a pointer, it is treated as data
  - even if it's not a primitive type – e.g., an array or struct



- Hence....
  - a definition reserves memory for the whole struct
    - As a local variable, on the stack
    - As a global variable, on the static data segment
    - (or malloc to dynamically allocate on the heap)
  - **passing** a struct to a function is *by value*

struct and memory – example

```
#include<stdio.h>
struct address
{       int number;
        char street[10000000];
};
void useStruct(struct address anaddress)
{ printf("%d \n",anaddress.number); }

struct address myAddress={108,"Anywhere Road"};

main()
{ useStruct(myAddress); }
```

File:q_struct_mem.c

Q:what happens here?

struct and memory – example

```
#include<stdio.h>
struct address
{       int number;
        char street[10000000];
};
void useStruct(struct address ana
{ printf("%d \n",anaddress.number

struct address myAddress={108,"Anywhere Road"};

main()
{ useStruct(myAddress); }
```

A: A large struct is stored in main memory

- no problem, yet...
- ...but this call is *by value.*
  - i.e., it copies the entire struct
  - i.e., involves copying ALL of the struct's data onto the stack.

  → The stack probably isn't large enough!

File:q_struct_mem.c

Q:what happens here?

Other Custom 'Data Types'

Unions

- Sometimes you may wish to store **different types of data in the same space**
  - e.g., some assignments are given a mark, but some of them are given a grade
  - can be done with a `union`

```
typedef union
{
   int raw_mark ;
   char letter_grade ;
} assignment;

main ( )
{
   assignment assignment_result;
   assignment_result.raw_mark =100;
   assignment_result.letter_grade='A' ;
   printf("%d \n",assignment_result.raw_mark);
}
```

Question: What does this code produce?

union.c

Unions Q&A

- How much memory is taken up?
  - Enough space for the largest of the possible elements
- How does the compiler / program know which type of data is stored?
  - it doesn't
- Does that mean I may be treating a float as an int, for example?
  - Yes, you may be, if you aren't careful
- Isn't this a bit dangerous, can't unpredictable things happen?
  - Yes – particularly if you make a mistake with a pointer
- So, I could end up dereferencing a pointer when it is actually just a numerical value?
  - Yes – your program will probably crash
- How do I find out which type is being represented?
  - You have to do that yourself!
- Why should I bother with unions?
  - There are reasons why this may be sensible
  - E.g., limited memory on some embedded systems (robots)

Review

- Multi-dimensional arrays
  - are **not** *arrays of arrays*
  - main point of difference: contiguous or not
  - pros and cons follow from that
- User defined 'datatypes': Structs / Unions
  - you can create composite data types with struct and union
  - passing a whole struct as a function parameter may be inefficient
  - unions allow variables to share a memory space – be careful!