Multidimensional arrays in C
Lists in C
Some other C programs
Multidimensional arrays in C

- The language has no concept of multidimensional arrays.
- Instead you simply use arrays of arrays.

```c
double m[3][4];
double x[2][3][4][5];
```

- So `m` is an array with three elements, where each element is an array of four doubles.
- `x` has two elements.
For an array of `char`, eg `char v[20]`, twenty bytes are needed and they are in consecutive array elements.

An `int` typically is stored in four bytes and a double in eight bytes.

You might wonder how arrays of arrays are organised in memory.

Since `double m[3][4]` is an array with three elements of arrays with four elements, the four element arrays are in consecutive bytes.

So, one row of four elements comes at a time.

This is called **row-major order**.
double m[3][4]

<table>
<thead>
<tr>
<th>address</th>
<th>element</th>
</tr>
</thead>
<tbody>
<tr>
<td>m+0</td>
<td>m[0][0]</td>
</tr>
<tr>
<td>m+8</td>
<td>m[0][1]</td>
</tr>
<tr>
<td>m+16</td>
<td>m[0][2]</td>
</tr>
<tr>
<td>m+24</td>
<td>m[0][3]</td>
</tr>
<tr>
<td>m+32</td>
<td>m[1][0]</td>
</tr>
<tr>
<td>m+40</td>
<td>m[1][1]</td>
</tr>
<tr>
<td>m+48</td>
<td>m[1][2]</td>
</tr>
<tr>
<td>m+56</td>
<td>m[1][3]</td>
</tr>
<tr>
<td>m+64</td>
<td>m[2][0]</td>
</tr>
<tr>
<td>m+72</td>
<td>m[2][1]</td>
</tr>
<tr>
<td>m+80</td>
<td>m[2][2]</td>
</tr>
<tr>
<td>m+88</td>
<td>m[2][3]</td>
</tr>
</tbody>
</table>
Computers are optimised for accessing array elements in sequence.

Reading $m[0][0]$, $m[0][1]$ and then $m[0][2]$ is usually faster than reading $m[0][0]$ and then $m[2][2]$.

Hardware designers (or computer architects) have observed that programmers almost always access arrays in sequence and not randomly.

Hardware can be faster if this behaviour is exploited.

On the other hand, since hardware is built to optimise such behaviour, programmers should write their programs so that they behave like that.

More about this will follow when we talk about cache memories.
Consider now an array allocated with calloc.

double* a = calloc(10, sizeof(double));

Calloc has no idea about what we are going to do with 80 bytes of memory.
But we do get some 80 bytes of memory.
When we write a[i] = 3.14; it is the compiler which fixes so that we write to element number i of these 80 bytes.
That is, not byte number i but 8-byte element number i.
Suppose we want an \( m \times n \) matrix from calloc. How do we do?

A one-dimensional array was declared as: `double* a`.

Here `a` is a pointer which points to the start of the calloc-ed memory.

So a two-dimensional matrix, can be declared as `double** m`.

But how can we allocate memory for it???

First allocate an \( m \) array which can hold pointers to the rows,
and then allocate memory for each row.
double** make_matrix(int m, int n) {
    double** a;
    int i;

    a = calloc(m, sizeof(double*));
    for (i = 0; i < m; i += 1)
        a[i] = calloc(n, sizeof(double));
    return a;
}

- Now we can write `double** m = make_matrix(3, 4);`
- We can access the elements as `m[i][j].`
Alternatives

- Instead of doing $m + 1$ calls to calloc, we can make one big:

  ```c
  double* a = calloc(m * n, sizeof(double));
  ```

- Now we cannot use it as a two-dimensional matrix. Assume we want $a[i][j]$:

  ```c
  for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
          a[i * n + j] = ...;
  ```

- The row number is determined by $i$ and each row has $n$ elements.
- We cannot write $a[i][j]$ since the type of $a[i]$ is a double and not an array.
What is really double** a?

- Can we really know what double** a is? No!
- It can be a two-dimensional array, but we don’t know the dimensions!

```c
double a;
double* b = &a;
double c;
f(a); void f(double x) { } // no effect.
g(b); void g(double* x) { *x = 3.14; } // changes A.
h(&b); void h(double** x) { *x = &c; } // changes B.
```

- In g, a becomes 3.14 since double* x points to a.
- In h, double** x can modify a pointer and has nothing to do with matrices.
The data allocated by `void* calloc(size_t count, size_t size)` is initialised to zeroes.

There is an alternative function `void* malloc(size_t size)` which leaves the data uninitialised.

Using `malloc` but forgetting to initialise the data leads to painful bugs.

You will often notice that the data is already zeroed by `malloc` but that is only by accident (by chance).

The function `void* realloc(void* ptr, size_t size)` tries to extend (or shrink) the memory area pointed to by `ptr`. If that is not possible it allocates new memory and copies the old content. Why can that be dangerous?
There are of course various kinds of lists, eg:

- Single linked,
- Single linked, with header pointing to the end (instead of having data).
- Null terminated double linked,
- Circular double linked.
An example circular double linked list

typedef struct list_t list_t;

struct list_t {
    list_t* succ;
    list_t* pred;
    void* data;
};

- Without the typedef we must write `struct list_t` everywhere.
- By circular is meant that the head’s predecessor points to the last node and the successor of the last node points to the head.
Making a list node

```c
list_t* new_list(void* data)
{
    list_t* list;

    list = malloc(sizeof(list_t));

    if (list == NULL)
        error("out of memory");

    (*list).succ = list;
    (*list).pred = list;
    (*list).data = data;

    return list;
}
```
Using the \( \rightarrow \) syntax

```c
list_t* new_list(void* data)
{
    list_t* list;

    list = malloc(sizeof(list_t));

    if (list == NULL)
        error("out of memory");

    list->succ = list; // (*list).succ = list;
    list->pred = list; // (*list).pred = list;
    list->data = data; // (*list).data = data;

    return list;
}
```

The arrow is a shorthand for \((\ast list)\). and was added to C very early.
void free_list(list_t** head)
{
    list_t* h = *head; // better than using *list below.
    list_t* p;
    list_t* q;
    if (h == NULL)
        return;
    p = h->succ;
    while (p != h) {
        q = p->succ;
        free(p);
        p = q;
    }
    free(p);
    *head = NULL;
}
Remarks about free

```c
int* a;
int* b;
a = malloc(sizeof(int));
b = a;
free(a);
*a = 12; // invalid
a; // invalid
b; // invalid
```

- After you have freed an object, any mention of that object is wrong, and the behaviour is undefined. Anything is permitted to happen according to the C standard.
Why using `new_list` as a separate function?

- Reason against: function calls take longer time so write more code, be happy, and think your program gets faster.
- Reasons for:
  - a good compiler will make the code equally fast (in fact, faster),
  - cleaner code is easier to read,
  - you might want to add a check that `malloc` did not report out of memory,
  - you might not want to call `malloc` but rather a function which calls `malloc`, and if `malloc` does report out of memory, it can perhaps do some cleaning or write some data structures to disk, or
  - you might not want to use `malloc` directly but other tricks which we will talk about later.
Iterating through a circular list

```c
#include <stddef.h>

size_t length(list_t* head)
{
    size_t count;
    list_t* p;

    if (head == NULL)
        return 0;
    p = head;
    do {
        count += 1;
        p = p->succ;
    } while (p != head);
    return count;
}
```
A comment on whether to use list_t or not

- We set the data pointer to an object which we want to put in a list.
- Now, why don’t we just add the succ and pred variables to the object type directly?
- That would avoid the waste of allocating memory for the void* data.
- The answer is that if there is one obvious list that our objects should be put in, then that is a good idea, since it avoids calling malloc/free when dealing with lists.
- The list_t should be used when eg an object should be put in two lists at the same time.
- Another aspect is that it may be more convenient to waste that small amount of memory and just use the list_t type. Make your own priorities.
Strings in C

- Strings are adjacent characters terminated with a 0.
- ’’C is fun’’ is a string and consists of 9 bytes.
- Eg char v[10] can hold a string.
- Eg char* s can point to a string — but it is no string.
- If we also do s = malloc(10); it is still no string.
- However, s points to memory which can hold a string.
- If we now do s = ’’C is fun’’; — what will happen?
Effect of \texttt{s = "C is fun"}'

- When we write something like \texttt{"C is fun}, besides being wise, we also declare a \textit{string literal}.

- A string literal is a constant string which will be part of the program.

- It is essentially an anonymous array of characters of sufficient size which will be present in the program from start to end, and initialised to the string we wrote.

- But what happens at the assignment \texttt{s = "C is fun"}?

- Hypothesis: the characters somehow jump down into the memory previously allocated with \texttt{s = malloc(10)}.
Memory leaks

- The hypothesis is wrong. The string literal, being an array of characters, lives at a certain place, or address, in memory, and the assignment simply writes that address into \( s \).
- That means the old value of \( s \), the address of the memory allocated with malloc is lost.
- In fact that memory is lost until the program terminates.
- That is a bug which may or may not be serious. It is called a memory leak.
- A small memory leak every twenty minutes will let the program run for a long time before running out of memory, but more frequently and in eg a mobile phone or an Airbus, they are very unpleasant. Valgrind is a tool which will help you detect leaks.
To make a copy of a string, we can use the following function.

```c
#include <stdlib.h>
#include <string.h>

char* copy_string(char* s)
{
    size_t length;
    char* t;

    length = strlen(s);
    t = malloc(length + 1);  // why + 1 ???
    if (t != NULL)
        strcpy(t, s);
    return t;
}
```
size_t strlen(const char* s);

- The type size_t is an unsigned integer of some suitable size, and const means this function promises not to modify what s points to.

```c
size_t strlen(const char* s)
{
    size_t length = 0;
    while (*s != 0) { // have we reached the zero yet?
        length += 1; // one more char found.
        s += 1; // step to the next character.
    }
    return length;
}
```
size_t strlen(const char* s) {
    const char* s0 = s;
    while (*s != 0)
        s += 1;
    return s - s0;    // length is difference in addresses
}

- An implementation defined type is a type which the compiler decides what is should be. For instance size_t can be defined to be unsigned long in one compiler and unsigned int in another.
- The type of the difference between two pointers is an implementation defined type called ptrdiff_t
- ptrdiff_t is a signed integer type, whose size is unrelated to size_t.
A faster portable size_t strlen(const char* s);

#include <string.h>
#include <stdint.h>

size_t strlen(const char* s) {
    const char* s0 = s;  // uintptr_t is an implementation
    uintptr_t t0;       // defined unsigned integer type
    uintptr_t t;        // which can hold a pointer value.

    while (*s != 0)
        s += 1;

    t0 = (uintptr_t)s0;
    t = (uintptr_t)s;

    return t - t0;      // length is difference in addresses
}