Recall: Pointers

```
int x = 12;
int *p;
int main()
{
        p = &x;
        *p = 13;
        return x * 2;
}
```

- A pointer is just a variable and it can hold the address of another variable.
- When p points to x, writing `*p` accesses `x`.

Recall: Memory layout

	instruction/data	**Java**	**comment**
0	STORE 6 at 7	MEMORY[7] = 6	&x is put in element 7, ie p
1	READ from 7 into R	R = MEMORY[7]	read data in p: R=6
2	STORE 13 at R	MEMORY[R] = 13	*p = 13
3	READ 6 into R	R = MEMORY[6]	fetch the value of x
4	MUL 2	R = R * 2	multiply x and R
5	RETURN	return R	
6	12		x lives here
7	0		p lives here

Function calls and local variables

- When you call a function or method, all the local variables must be stored somewhere.
- It is a convention to put them at the end of the memory array.
- The local variables of the main function are put at the very end of the array.
- When main calls a function, its local variables are put just before main's.
- In general, when a new function starts running, it puts its local variables at the last (highest index) unused memory array elements.
- This works like a stack of plates: main is at the bottom and you put newly called functions on the plate at the top.

The stack

```
int main()
{
    int x = 12;
    return f(x);
}
```

```
int f(int a)
{
    int b = a+1;
    return g(b+2);
}
```

```
int g(int a)
{
    return a + 3;
}
```

1073741820	15	a in g lives here.
1073741821	13	b in f lives here.
1073741822	12	a in f lives here.
1073741823	12	x in main lives here.

- When a function returns, it deallocates its memory space.
- This is managed by the compiler which uses a register for holding the current free memory index, called the **stack pointer**.

More about pointers

- In Java, you have used pointers all the time, but they are called object references.
- Suppose you have `Link p`, then `p` is a pointer.
- In Java, pointers can only point at objects.
- The address of some object is, as you might know, the location in memory where that object lives, ie just an integer number.
- In Java, `new` returns the address of a newly created object.
- In C, `new` is a normal function and is called `malloc`.

More about pointers

- In C, but not in Java, the programmer can ask for the address of almost anything and thus get a pointer to that object (or function).
- To change the value of a variable in a function, you need to pass the address of the variable as a parameter to the function:

```
void f(int* a)
{
        *a = 12;
}
```

```
void g()
{
        int     b;

        f(&b);
}
```

More about pointers

- If the type of the variable is a pointer, then you will need two stars:

```
void f(int** a)
{
        *a = NULL;
}
```

```
void g()
{
        int*     b;

         f(&b);
}
```

Class in Java vs Struct in C 1(4)

- C has no classes!
- C has structs which are Java classes with everything public and no methods.

```
struct a {  // a is a tag.
        int     b;
        int     c;
} d;        // d is a variable identifier.
```

- Struct names have a so called **tag** which is a different name space than variables and functions: so the above declares a `struct a` which is a type and a variable `d`.
- If we write `Link p` in Java we declare `p` to be a reference but not the object itself whereas `s` above is the *real* object, or data.

Class in Java vs Struct in C 2(4)

- In Java we can declare a List class something like this:

```
class List {
        List     next;   // Next is a reference
        int      a;
        int      b;
};
```

- `next` above only holds the address of another object but `next` *is not a List object itself*. The list does not contain a list.
- Java let's you use pointers conveniently without giving you too much head ache.
- C does not.

Class in Java vs Struct in C 3(4)

- We cannot write the following in C:

```
struct list_t {
        struct list_t    next;   // Compilation error!!
        int              a;
        int              b;
};
```

- It is impossible to allocate a list within the list!
- We really want to declare `next` to simply hold the address of a list object.
- In C this is done as: `struct list_t* next;` which makes `next` a pointer.

Class in Java vs Struct in C 4(4)

- The following is correct in C:

```
struct list_t {
        struct list_t*    next;
        int               a;
        int               b;
};
```

- After going into pointers in more detail we will see how to avoid typing `struct list_t` more than twice using `typedef`.

More about pointers

- To return multiple values in Java, you create and return an extra object.
- Option 1 in C: use a plain struct which is allocated on the stack.
- Option 2 in C: Pass additional arguments as pointers (preferable).

```
struct s f()
{
        struct s a;
        a.x = ...;
        a.y = ...;
        a.u = ...;
        return a;
}
```

```
void g(int* x, int* y, int* u)
{

        *x = ...;
        *y = ...;
        *u = ...;
}
```

Creating another name of a type

- The `typedef` command creates another name for the specified type:

```
typedef int integer;
integer a,b;

typedef struct list_t  list_t; // list_t is a type.

struct list_t {
        list_t*     next;
        int         a;
        int         b;
};
```

More about typedef

```
typedef struct _list_t  list_t;
struct _list_t {
        list_t*     next;
        int         a;
        int         b;
};
```

- Two errors: starting a tag (or identifier) with an underscore is permitted **only for compiler and library implementors**.
- There is no need to invent two identifiers here: call both the typedef name and the tag the same thing!

C has row-major matrix memory layout

```
int c[3][4] = { { 1, 2, 3, 4}, { 5, 6 }, { 7 } };
int i, j;
for (i = 0; i < 3; i++)
        for (j = 0; j < 4; j++)
                x += c[i][j];
```

- In a two-dimensional array, one row is layed out in memory at a time, ie row-major.
- Could also be called "rightmost index varies fastest".
- The elements `c[i][j]` and `c[i][j+1]` are next to each other.

Arrays as parameters

```
int fun(int c[3][4])
{
        printf("%zu %zu\", sizeof c, sizeof c[0]);
}
```

- If the output is ”4 16”, what conclusions can we draw about the size of a pointer and the size of an int?
- Answer: both are four bytes.
- The variable `c` in the function is a pointer: `int (*c)[4]`.

Representation of array references

- a[i] is represented as `*(a+i)`

```
int main(void)
{
        int     a[10], *p, i = 3;

        /* the following are equivalent: */

        &a[i] == a+i;

        p = a; p[i] == a[i];

        p = a+i; p[0] == *p;

        return 0;
}
```

Multidimensional arrays in C

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

Arrays of arrays

```
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.

Multidimensional arrays with calloc

- Suppose we want an $m \times n$ matrix from calloc. How do we do?
- A one-dimensional array is declared as: `double* a`.
- Here `a` is a pointer which points to the start of the calloc-ed memory.
- A two-dimensional matrix, can be declared as `double** m`.
- But how can we allocate memory for it???
- First allocate an array which can hold m pointers to the rows,
- and then allocate memory for each row.

More from previous slide

```
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:

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

- Unfortunately, we cannot use it as a two-dimensional matrix. Assume we want `a[i][j]`:

```
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.
- We will return to matrices in a later lecture and explain Examples 1.4.10 and 1.4.11 which are more advanced.

malloc/calloc/realloc/free

- The data allocated by `void* calloc(size_t count, size_t size)` is initialized to zeroes.
- There is an alternative function `void* malloc(size_t size)` which leaves the data uninitialised.
- Using malloc but forgetting to initialize 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, and if that is not possible it allocated new memory and copies to old content. Why can that be dangerous ?

Lists in C

- 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

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

        list = malloc(sizeof(list_t));

        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 `(*list).` and was added to C very early.

Freeing of a list

```
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;
}
```