Pointers again

- A pointer is an *address* to a memory location.
- Addresses can be saved in *pointer variables*.
- Pointer variables are declared using the *unary* operator `*`.

  Examples:
  ```
  int *m, *n;
  double *x;
  ```
- We can get the address of a variable using the unary operator `&` .

  Examples:
  ```
  int b = 0;
  int *ip = &b;
  ```

Pointers cont.

- The *dereference operator* `*` is used to access the pointed place.

  Examples:
  ```
  int b = 0;
  int *ip = &b;
  *ip = *ip + 1;
  ```
  Same as `n = n + 1;`

- There is one *pointer constant* `NULL` meaning *"pointer to nothing"*. `NULL` has actually the integer value 0 so it is regarded as *false* in tests.

- It is illegal to dereference a `NULL`-pointer.

Example of usage

Suppose we want a function that solves the quadratic equation

$$x^2 + px + q = 0.$$

The solution is given by the formula: $x_{1,2} = -\frac{p}{2} \pm \sqrt{(\frac{p}{2})^2 - q}$

Since a function only can return ONE value, we use two *pointer parameters* to tell the function were to store the result.

The function value is used to indicate if there are complex roots (which we can't handle) or real roots.

Exmaple: A quad function

```
int quad(double p, double q, double *x1, double *x2) {

   double d = p*p/4. - q;

   if (d < 0) {
      return 0;        // Complex roots
   }
   d = sqrt(d);
   *x1 = -p/2 + d;
   *x2 = -p/2 - d;
   return 1;           // Real roots
}
```

Usage of quad function

```
int main() {
    double p, q;
    double x1, x2;
    while (1) {
        printf("Give p and q: ");
        if (scanf("%lf %lf)", &p, &q) != 2) {
            break;
        }
        if (quad(p, q, &x1, &x2) == 1) {
            printf(" x1: %lg\n x2: %lg\n", x1, x2);
        } else {
            printf("Complex roots\n");
        }
    }
}
```

Link to quadEquation.c

Dynamically allocated memory

There are two types of memory area available to the C-programmer:

- the *stack* and
- the *heap*.

The stack is used for all local variables i.e. variables declared inside functions (and not declared `static`).

Almost all variables we have used so far have been allocated on the stack.

The local variables are automatically allocated when the execution enters a function and deallocated when the function returns.

Thus, no values are saved between calls.

Using the heap

There are several ways to allocate memory on the heap:

- Put the declaration in the file but outside all functions (as `myDigits`)
  Such variable retains its value during the complete execution of the program. It is also accessible from all functions in the file.
- Declare a variable in a function as `static`.
  Such variables will keep their values between different calls to the function but can not be accessed from other functions.
- Use one of the memory allocation functions: `malloc`, `calloc` and `realloc`. These functions take the desired amount of memory as parameter and return a pointer to the first memory location in the allocated area.

Example

Suppose we want to simulate a lottery where we can pull numbered tickets in such way that each number only occurs once.

We would like to be able to run a program like this:

```
int main() {
  int m;
  printf("Number of tickets: ");
  scanf("%d", &m);
  setUp(m);
  for (int i = 0; i < m; i++) {
    printf("%d  ", pull());
  }
  printf("\n");
}
```

Example cont.

Algorithm

- Use an array `int theTickets[`m`]` for storing the tickets.
  It should be initialised with the numbers 1, 2, 3, 4, ... , m.
- Use a variable n to keep track of the number of tickets left.
- To pull a ticket we generate a random number r in the interval `[0,` n`-1]`. Take the ticket at index r and move the last ticket to index r and decrease n.
  Return the taken ticket.

Example

Thus two functions:

- `void setUp(int size)` for initialising the tickets and
- `int pull()` for getting a ticket

(Remark: `size` is a better variable name than `m`).

The array with tickets as wells the number of tickets left (n) must be accessible from both functions so we place them outside the functions.

Since the number of tickets is not known at compile time we must allocate the array using `malloc` or `calloc`.

Example cont.

```
int *theTickets;
int ticketsLeft;
```

"Global" variables. Will exist during the complete execution.

```
void setUp(int size) {
  theTickets =
     (int *) malloc(size*sizeof(int));
  for (int i = 0; i<size; i++) {
    theTickets[i] = i + 1;
  }
  ticketsLeft = size;
}
```

Allocate using `malloc`

Array notation!

Note:

- `sizeof(`*data type*`)`
- The typecast `(int *)`

Example cont.

```
int pull() {
  int nbr = rand()%ticketsLeft;
  int ticket = theTickets[nbr];
  theTickets[nbr] =
     theTickets[ticketsLeft-1];
  ticketsLeft--;
  return ticket;
}
```

`rand` returns an integer random number

Link to urn.c

What happen when we run the program several times?

A new data type: the `struct`

Suppose we want to represent persons in a program. Every person should have a *name* and an *age*.

Such an object can be defined using a `struct`:

```
struct Person {
    char name[10];
    int age;
};
```

We can then declare variables of this type:

```
struct Person p, persons[100];
```

We can access the different fields in a `struct` using a "dot" notation:

```
strcpy(p.name, "Eva");
p.age = 42;
```

Type aliases: typedef

```
typedef
  struct Person {
     char name[10];
     int age;
  } Person, *pLink;
```

We can then declare variables of this type:

```
Person p, persons[100];
```

and

```
pLink pl;   //pointer to a Person-struct
```

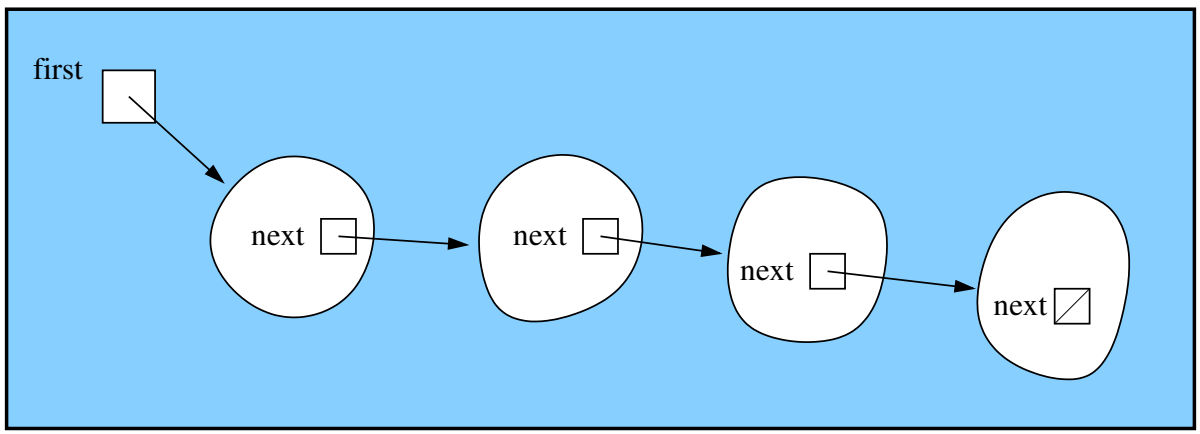
if we want a pointer to a `Person`.

A linked list

Suppose we want to store an unknown number of person objects. Instead of using an array we will make a *linked list*.

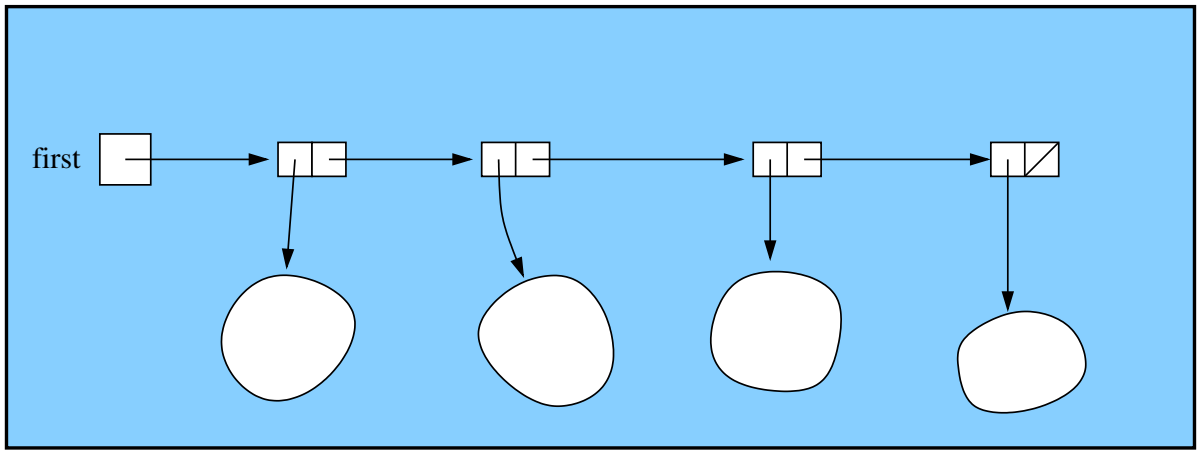
We can add a *pointer field* in each person struct:

```
struct Person {
    char name[10];
    int age;
    struct Person *next;
}
```



Alternative

Since "next" is an unnatural property for a person we could do like this:




```
typedef
  struct Person {
     char name[10];
     int age;
  } Person, *pLink;
```

```
typedef
   struct Node {
      pLink thePerson;
      struct Node *next;
   } Node, *link;
```

Linked lists cont.

Since we do not know the length of the list we allocate both person- och node-structs using `malloc`

```
pLink makePerson(char * name, int age) {
  pLink res = (pLink) malloc(sizeof(Person));
  strcpy((*result).name, name);
  (*res).age = age;
  return res;
}

link makeNode(pLink p, link next) {
  link res = (link) malloc(sizeof(Node));
  (*res).thePerson = p;
  (*res).next = next;
  return res;
}
```

Linked list cont.

A print function can be conveniently to have:

```
void printPerson(pLink p) {
  printf("%s \tof age %d\n", (*p).name, (*p).age);
}
```

And a demonstration program

```
int main() {
  link list = NULL;
  list = makeNode(makePerson("Sue",12), NULL);
  list = makeNode(makePerson("Kim",9),list);
  list = makeNode(makePerson("Eva",9),list);
  list = makeNode(makePerson("Don",7),list);
  list = makeNode(makePerson("Pete",7),list);
  for (link p = list; p!=NULL; p = (*p).next) {
      printPerson((*p).thePerson);
  }
}
```

Link to personList.c

A new operator: ->

The expression `(*p).f` where

- *p* is a pointer to a `struct` and
- *f* is a field in that `struct`

can be written *p* `->` *f*

Using that in, for example, `makeNode` gives:

```
link makeNode(pLink p, link next){
  link result = (link) malloc(sizeof(Node));
  result->thePerson = p;
  result->next = next;
  return result;
}
```