CS205

C / C++

Stéphane Faroult
faroult@sustc.edu.cn

Wang Wei (Vivian) vivian2017@aliyun.com

String conversion

These functions aren't too safe can be used alternatively to sscanf()

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

Old fashioned functions

```
int atoi(char *str);
long atol(char *str);
```

Better to use strtol() (more complicated)

```
double atof(char *str);
```

Better to use strtod()

Even if function returns a special code for errors (int functions often return -1 for errors, char * functions often return NULL), it's often a bit short to diagnose an error, especially as some functions such as fgets() return NULL if they fail to read or if they have reached the end of a file. errno.h contains helpful functions.

Error Handling

```
#include <errno.h>
```

fgets() → returns NULL if error

SOMETHING WRONG HAPPENED

```
#include <errno.h>
```

```
int errno
```

Global variable

Set by all system calls - reset to 0 by all successful system calls

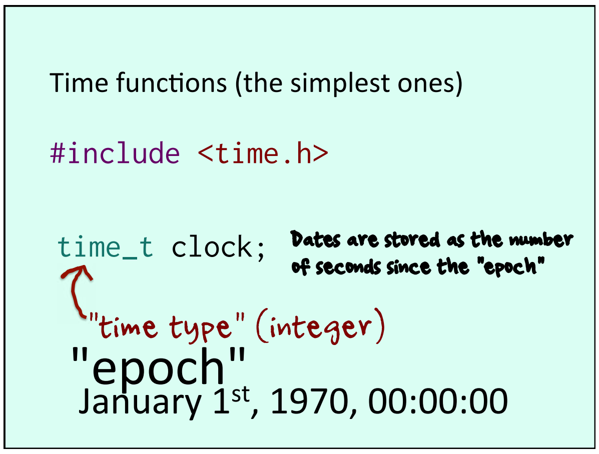
```
char *strerror(int errnum )
```

Returns the text of the associated error message

```
perror("your message")
       your message + strerror(errno )
```

→ stderr

Most functions before returning an error store a more precise error code into a variable called errno. The corresponding message may be retrieved.



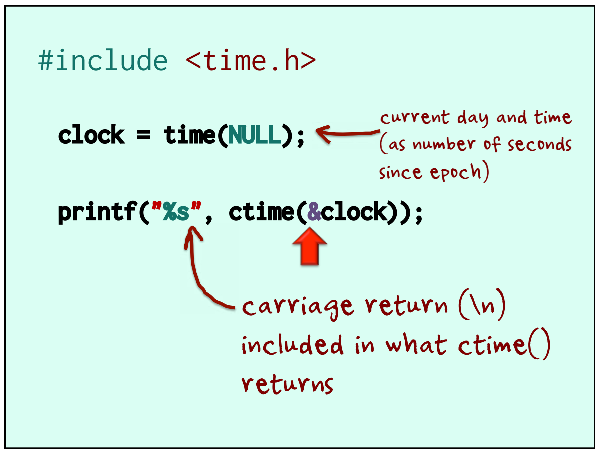
Time functions (the simplest ones)

```
#include <time.h>

time_t clock;
```

Dates are stored as the number of seconds since the "epoch"

"time type" (integer)

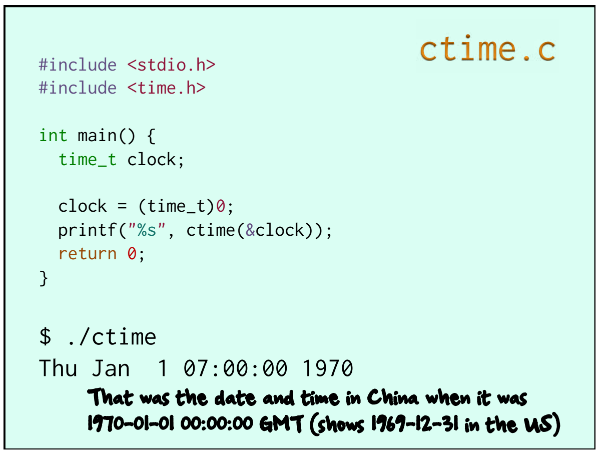
"epoch"
January 1st, 1970, 00:00:00



```
#include <time.h>

  clock = time(NULL);

  printf("%s", ctime(&clock));
```

current day and time (as number of seconds since epoch)

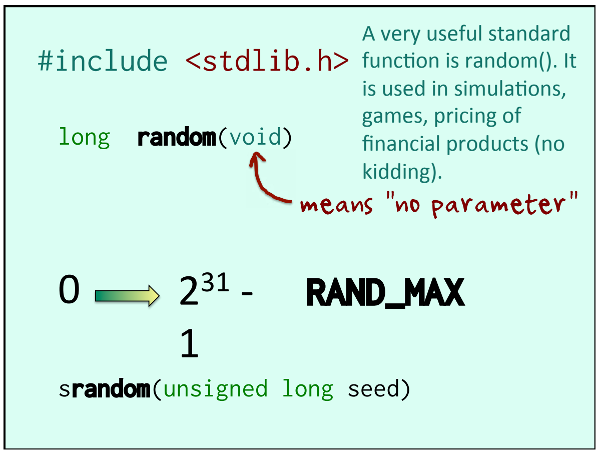
carriage return (\n) included in what ctime() returns



ctime.c

```
#include <stdio.h>
#include <time.h>

int main() {
  time_t clock;

  clock = (time_t)0;
  printf("%s", ctime(&clock));
  return 0;
}

$ ./ctime
Thu Jan  1 07:00:00 1970
```

That was the date and time in China when it was 1970-01-01 00:00:00 GMT (shows 1969-12-31 in the US)



```
#include <stdlib.h>

  long  random(void)
```

A very useful standard function is random(). It is used in simulations, games, pricing of financial products (no kidding).

means "no parameter"

$0 \longrightarrow 2^{31} - 1$ RAND_MAX

```
  srandom(unsigned long seed)
```

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

Random number between a and b

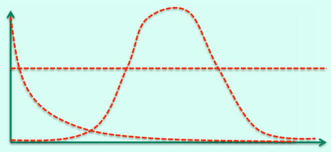
```
num = (data type)(a + (b – a)
            * random() / (double)RAND_MAX);
```

Random integer between 0 and *n* - 1

```
num = random() % n;
```

Yes but ...

Distribution issue



The problem with random() is that it returns a uniform distribution (same odds of getting any value between min and max). Real life distribution are different (bell curve, exponential ...)

Exponential distribution: use log()

Normal distribution: "Box-Müller transform"

There are some transformations that turn a uniform distribution into something else. Applying the log() function will turn the distribution into exponential. For obtaining a normal distribution (the bell curve) there is a more complicated transform (which uses C techniques that we will see later) that allows to get it from a uniform distribution as well.

Localization

```
#include <locale.h>

setlocale(LC_ALL, "zh_CN.UTF-8");
```

ctime() unchanged

As a reminder, `setlocale()` allows some internationalization (often called "i18n" for short) of programs. It affects some time functions (because date formats differ by country), but not ctime().

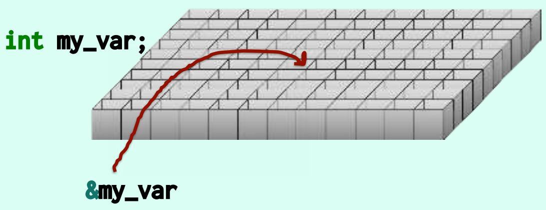
More on pointers

You don't need to know the exact memory location

(may change between executions)

Reminder:

& returns the address of a variable



You need to know what you are pointing at.

How many bytes are we interested in

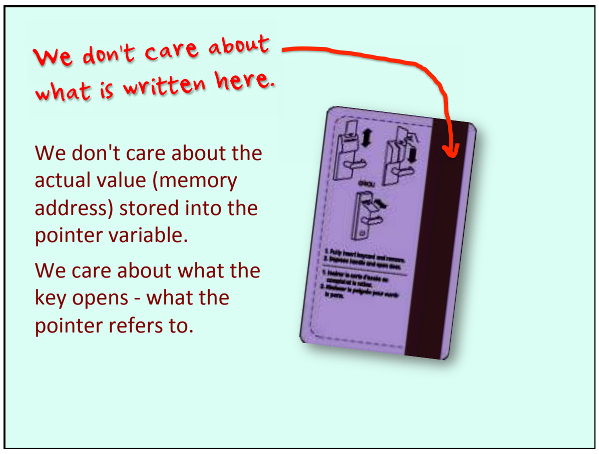
How should we **interpret** these bytes (encoding)

Four bytes will not be interpreted in the same way if they store an int, a float, or four chars.

A pointer is very much like a plastic hotel key card. The memory location is like a room in the hotel.

The card may give access to a budget room.

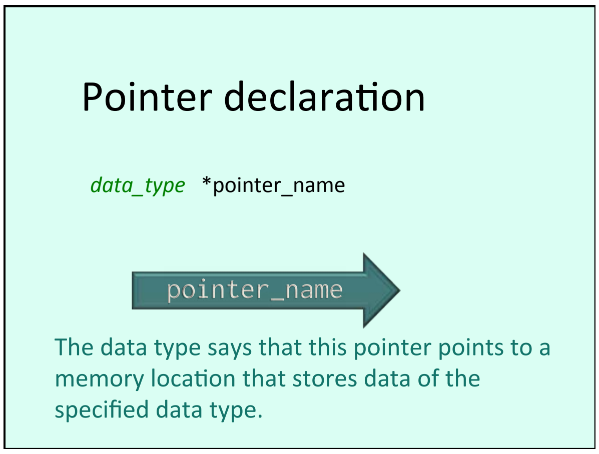
Or to a luxurious suite.
It will always look like the same card.



We don't care about what is written here.

We don't care about the actual value (memory address) stored into the pointer variable.

We care about what the key opens - what the pointer refers to.

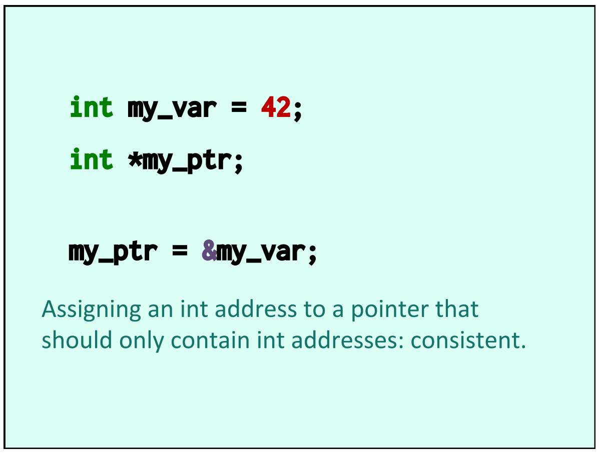


Pointer declaration

data_type *pointer_name

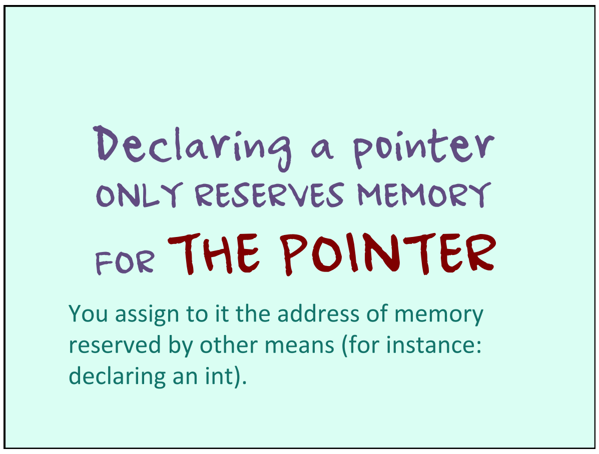
pointer_name

The data type says that this pointer points to a memory location that stores data of the specified data type.



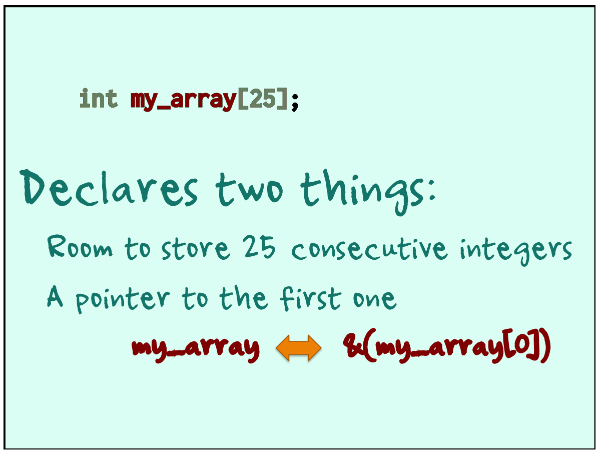
```
int my_var = 42;
int *my_ptr;

my_ptr = &my_var;
```

Assigning an int address to a pointer that should only contain int addresses: consistent.



Declaring a pointer
ONLY RESERVES MEMORY
FOR THE POINTER

You assign to it the address of memory reserved by other means (for instance: declaring an int).



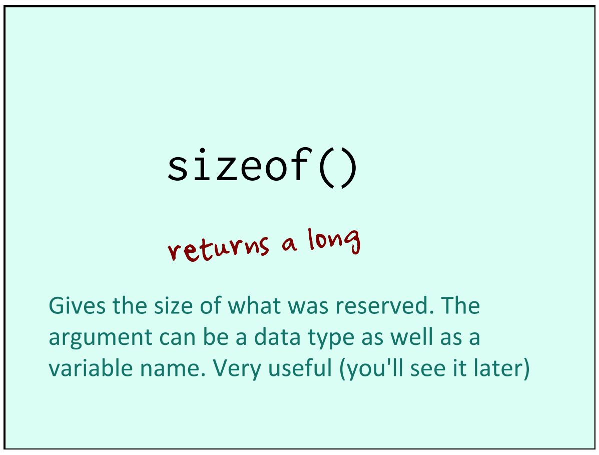
```
int my_array[25];
```

Declares two things:

Room to store 25 consecutive integers

A pointer to the first one

my_array ⇔ &(my_array[0])



```
sizeof()
```

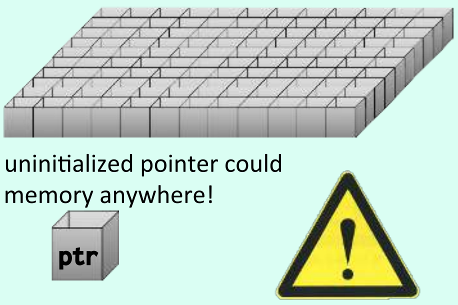
returns a long

Gives the size of what was reserved. The argument can be a data type as well as a variable name. Very useful (you'll see it later)

NULL

Initialize pointers to a variable address or to NULL!

NULL is a constant which represents a pointer with all bits set to zero (it's to pointers what \0 is to chars ...) Functions that return a pointer return NULL when they fail or are done. Note that it's in uppercase, not in lowercase as in Java (or Javascript).



An uninitialized pointer could hit memory anywhere!

A pointer holds a reference to a variable value.

A bit of jargon ...
This is why Java talks about references as well (the big difference in C is that you can handle references explicitly).

Dereferencing

= using the pointer to get the value

* operator

```
int my_var = 42;
int *my_ptr;

my_ptr = &my_var;
printf("%d\n", *my_ptr);
```

It will print 42

(int) value found at the address stored in my_ptr

CAUTION

DON'T CONFUSE POINTER INITIALIZATION AND DEREFERENCING

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

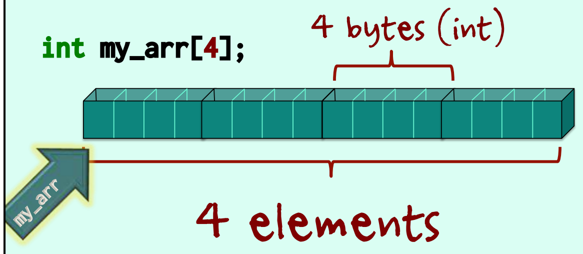
"what my_ptr points to is an int"

"this is what I store in my_ptr - the address of an int"

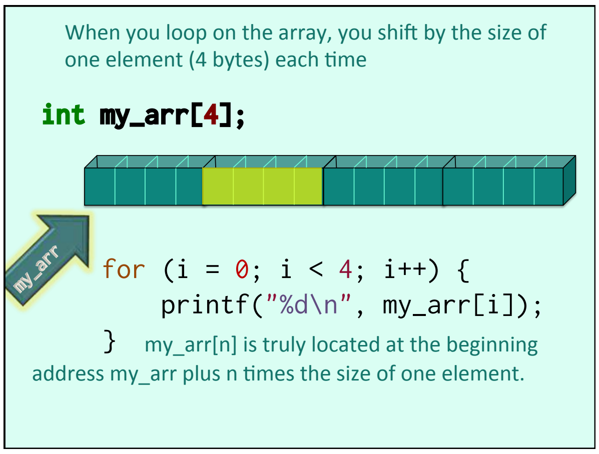
```
my_ptr = &my_var;
```

Pointers and arrays

```
int my_arr[4];
```



When you declare an array of 4 ints, the compiler reserves 16 bytes (4 x 4).

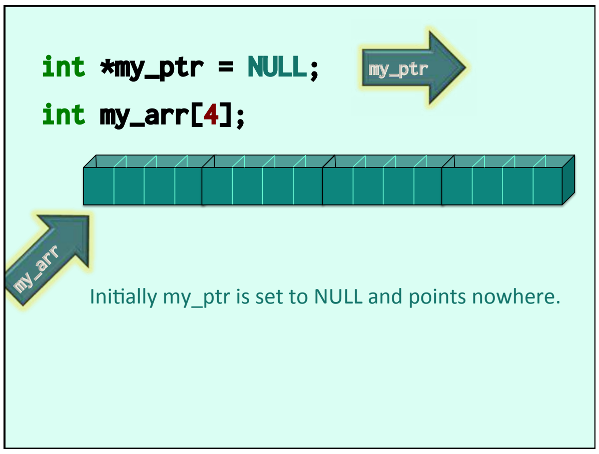


When you loop on the array, you shift by the size of one element (4 bytes) each time

```
int my_arr[4];
```

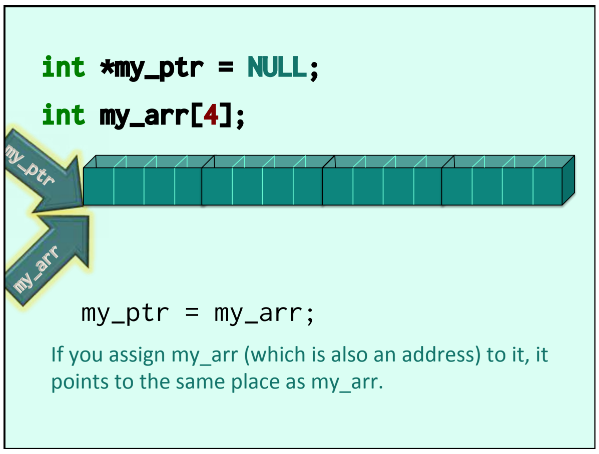
```
for (i = 0; i < 4; i++) {
    printf("%d\n", my_arr[i]);
}
```

my_arr[n] is truly located at the beginning address my_arr plus n times the size of one element.



```
int *my_ptr = NULL;
int my_arr[4];
```

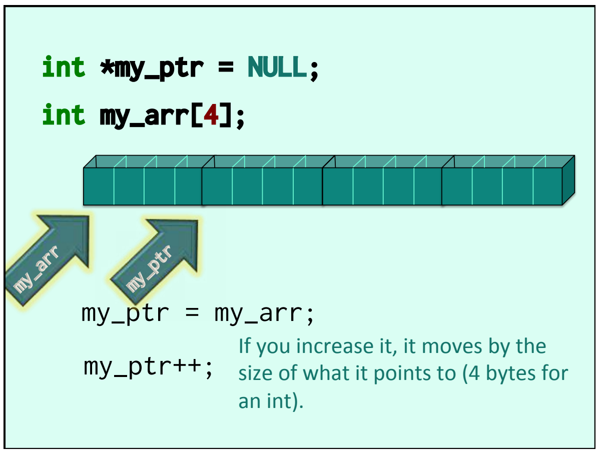
Initially my_ptr is set to NULL and points nowhere.



```
int *my_ptr = NULL;
int my_arr[4];
```

```
my_ptr = my_arr;
```

If you assign my_arr (which is also an address) to it, it points to the same place as my_arr.



```
int *my_ptr = NULL;
int my_arr[4];
```

```
my_ptr = my_arr;

my_ptr++;
```

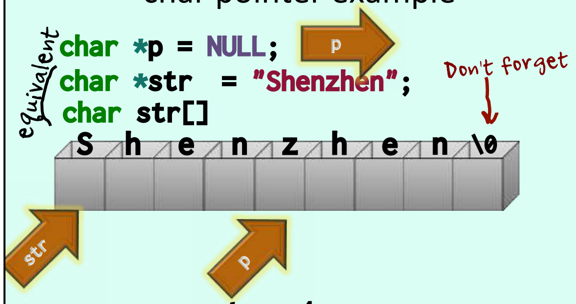
If you increase it, it moves by the size of what it points to (4 bytes for an int).

Pointer arithmetic

pointer + *n*	Makes the pointer point to the nth element - like array indices.

p++	Increments p by as many bytes as the type p points to
***p++**	Returns the value pointed to by p then increases p (in an assignment)
(*p)++	Increases by one what p points to (COMPLETELY DIFFERENT)

char pointer example

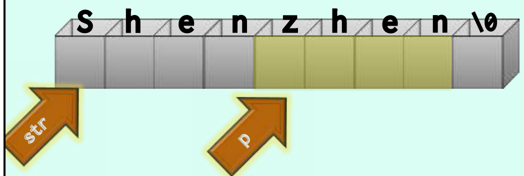


```
char *p = NULL;
char *str  = "Shenzhen";
char str[]
```

```
p = str + 4;
```

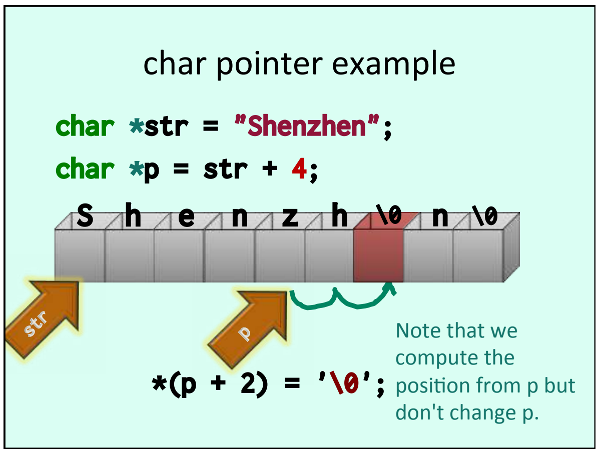
char pointer example

```
char *p = NULL;
char *str  = "Shenzhen";
```



```
printf("%s\n", p);
```

Prints everything from p up to \0



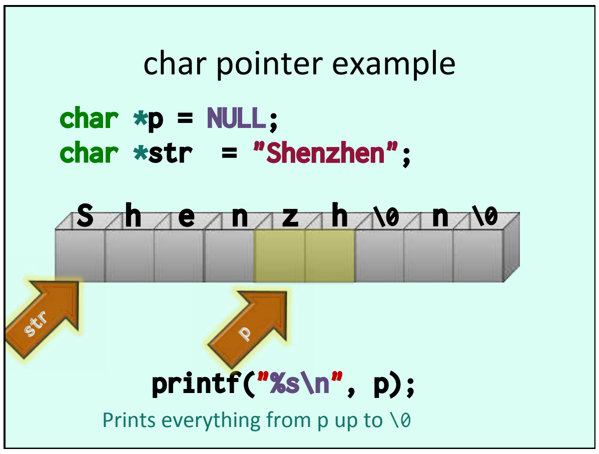
char pointer example

```
char *str = "Shenzhen";
char *p = str + 4;
```

S h e n z h \0 n \0
str
p

```
*(p + 2) = '\0';
```

Note that we compute the position from p but don't change p.



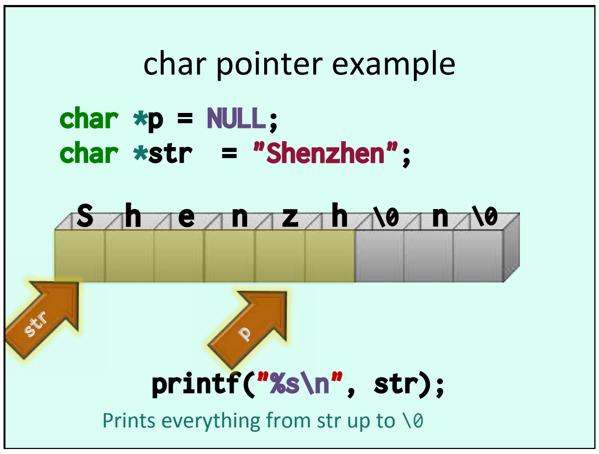
char pointer example

```
char *p = NULL;
char *str  = "Shenzhen";
```

S h e n z h \0 n \0
str
p

```
printf("%s\n", p);
```

Prints everything from p up to \0



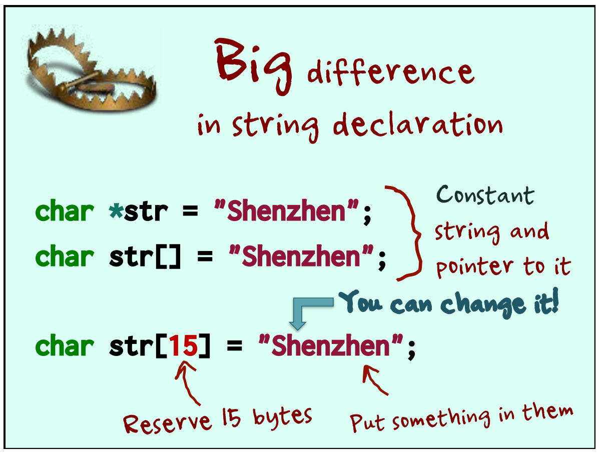
char pointer example

```
char *p = NULL;
char *str  = "Shenzhen";
```

S h e n z h \0 n \0
str
p

```
printf("%s\n", str);
```

Prints everything from str up to \0



Big difference in string declaration

```
char *str = "Shenzhen";
char str[] = "Shenzhen";
```

Constant string and pointer to it

You can change it!

```
char str[15] = "Shenzhen";
```

Reserve 15 bytes

Put something in them

Note: because arrays BOTH reserve memory and define a pointer to this memory area, operations on pointers and arrays aren't completely symmetrical.
You can say:

```
    ptr = array_name;
```

to make ptr point to the memory area pointed to by array_name, but you CAN'T say

```
    array_name = ptr;   // Invalid
```

You would lose the location of the area reserved by the array.

The main difference between a C array and a Java array is that when you declare a C array there is an implicit memory reservation. There is none in Java, which is why a new is required (we'll see shortly the C equivalent). Java only declare a reference, in fact.

Arrays of strings

Array element = **string** = array of characters

Array of arrays

Bi-dimensional array

Arrays of strings

```
char simple_string[] = "A string";
```

element type

An array (compiler sets the size)

CONSTANT

Can also be written:

```
char *simple_string  = "A string";
```

"string type"

Arrays of strings

```
char *simple_string  = "A string";

char *string_array[]  = {"Welcome",
                         "Bienvenidos",
                         "Bienvenue",
                         "Willkommen",
                         "Bemvindos",
                         NULL};
```

If you let the compiler compute the size, you should have a NULL to know where the array stops when you loop.

(Another method is to use function sizeof() to get the size of the array and divide by the size of one element, here a char *)

Arrays of strings

```
char *simple_string  = "A string";

char **string_array   = {"Welcome",
                         "Bienvenidos",
                         "Bienvenue",
                         "Willkommen",
                         "Bemvindos",
                         NULL};
```

Can also be written that way. What you find at `string_array` is the address of the W of "Welcome"

(confused? It's normal)

A multi-dimensional array can also be given explicit sizes:

```
double matrix[ROW_COUNT][COL_COUNT]
```

Not much used with char strings.

Command line parameters

This is the true, complete definition of main()

```
int main(int argc, char *argv[]) {
```

Passed to the program by the system

```
  return 0;
}
```

Command line parameters

You will also find it written like this; it's exactly the same thing, argv is an array of strings.

```
int main(int argc, char **argv) {



    return 0;
}
```

Command line parameters

Beware it's different from Java! Two parameters, and there is always one element in argv[].

`int argc` C is for count; always >= 1

`char *argv[]` V is for value; argv[0] always contains the name of the program

echo.c

```
#include <stdio.h>

int main(int argc, char **argv) {
  int   repeat = 1;
  int   i;
  char *w = NULL;

  if ((argc < 2) || (argc > 3)) {
    fprintf(stderr, "Usage: %s [times] word\n", argv[0]);
    return 1;  <—— Quit
  }
  if (argc == 3) {
    if (!sscanf(argv[1], "%d", &repeat)) {
      fprintf(stderr,"First parameter must be a number of times\n");
      return 1;
    }
    w = argv[2];
  } else {
    w = argv[1];
  }
```

echo.c

```
    w = argv[1];
  }
  for (i = 0; i < repeat; i++) {
    printf("%s ... ", w);
  }
  putchar('\n');
  return 0;
}
```

```
$ ./echo
Usage: ./echo [times] word
$ ./echo hello
hello ...
$ ./echo 3 hello
hello ... hello ... hello ...
$ ./echo hi folks
First parameter must be a number of times
$
```

Passing flags to commands is very common in Unix systems. There is in `stdlib.h` the prototype of a function called `getopt()` that is very convenient for parsing flags.

There is a very interesting function called getopt() for managing flags passed to commands – such as gcc -c mycode.c -o mycode

STRUCTURES

Structures are the closest C comes to the idea of an object. Structures are like objects with only attributes and no methods (although you can store pointers to functions in structures - but pointers to functions are relatively advanced C usage). Everything is public.

This is how you can declare a variable named my_place which is a structure.

Declaration

```
struct {
        char    place_name[NAME_LEN];
        double  latitude;
        double  longitude;
      } my_place;
```

(nameless) user-defined data type

You can also have arrays of structures.

```
struct {
        char   place_name[NAME_LEN];
        double latitude;
        double longitude;
       } my_place[PLACE_COUNT];
```

You refer to the attributes (or fields) in a structure by a dot notation (like in Java but you'll see that the meaning isn't quite the same as in Java).

```
struct {
        char   place_name[NAME_LEN];
        double latitude;
        double longitude;
       } my_place;


strncpy(my_place.place_name, "Shenzhen", NAME_LEN);
my_place.latitude = 22.25;
my_place.longitude = 114.1;
```

You can also initialize a structure with a syntax close to array initialization.

```
struct {
        char   place_name[NAME_LEN];
        double latitude;
        double longitude;
       } my_place = {"Shenzhen",
                     22.25,
                     114.1};
```

Consecutive places in memory - like an array

Naming a structure

Two (non-exclusive) ways

```
#include <stdio.h>
#include <string.h>

#define NAME_LEN 50

struct my_struct {
                  char   place_name[NAME_LEN];
                  double latitude;
                  double longitude;
                 };

int main() {
  struct my_struct my_place;

  strncpy(my_place.place_name, "Shenzhen", NAME_LEN);
  my_place.latitude = 22.25;
  my_place.longitude = 114.1;
  return 0;
}
```

You can have a blue-print of the structure (very loosely like a class).

Declaration is like an instantiation - but there is no new here (important)

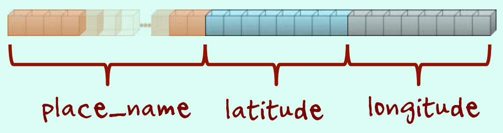
```
#include <stdio.h>
#include <string.h>

#define NAME_LEN 50

typedef struct my_struct {
                  char   place_name[NAME_LEN];
                  double latitude;
                  double longitude;
                 } PLACE_T;

int main() {
  PLACE_T my_place;

  strncpy(my_place.place_name, "Shenzhen", NAME_LEN);
  my_place.latitude = 22.25;
  my_place.longitude = 114.1;
  return 0;
}
```

Typedef gives a user-defined name to the structure.

typedef isn't necessarily used with a struct - it can be used with any data type, eg

```
typedef int my_type;
```

```
typedef struct my_struct {
                     char   place_name[NAME_LEN];
                     double latitude;
                     double longitude;
                   } PLACE_T;
```



Sequential storage reserved in memory.

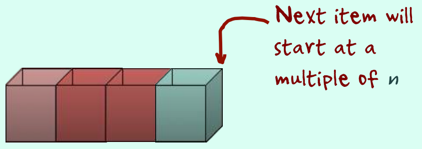
Structures aren't necessarily "packed"

There may be additional (unused) bytes in the structure.

struct.c

```c
#include <stdio.h>

struct my_struct {
                  short a_short;
                  char  a_char;
                 };

int main() {
  printf("size of a short is %ld bytes\n",
         sizeof(short));
  printf("size of a char is %ld bytes\n",
         sizeof(char));
  printf("size of a struct my_struct is %ld bytes\n",
         sizeof(struct my_struct));
  return 0;
}
```

```
$ ./struct
size of a short is 2 bytes
size of a char is 1 bytes
size of a struct my_struct is 4 bytes
$
```



Alignment

Depends on architecture (i.e. type of computer)

Some computers want everything to start at even addresses, so there may be "padding"

What about POINTERS ?

Don't rely on pointer arithmetic to point to the various components of a structure

Because of "padding" and because elements are usually of different types.

```
typedef struct my_struct {
                  char   place_name[NAME_LEN];
                  double latitude;
                  double longitude;
                 } PLACE_T;


PLACE_T   my_place;

struct my_struct *ptr;
PLACE_T   *ptr;
```

Equivalent declarations

Dereferencing?

~~*ptr~~

Not one value

Dereferencing?

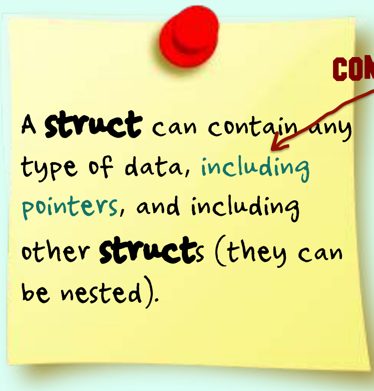
Use the "arrow notation" with a pointer to a structure.

```
ptr->place_name
ptr->latitude
ptr->longitude
```

How you access attributes is significantly different from Java. In Java, a composite type is always an object, and when you declare an object it's always a pointer; you "instantiate" it by calling a constructor. In C, you can have structure variables that are chunks of memory like simple variables (you can also reserve memory dynamically as you'll see later). The dot notation is used to access attributes of "plain variable" structures. The arrow notation is actually equivalent to the dot notation of Java (which only accesses atributes through a reference) and means access by a pointer.

```
struct_variable.fieldname

pointer_to_struct->fieldname
```

Same as a simple (non composite) variable

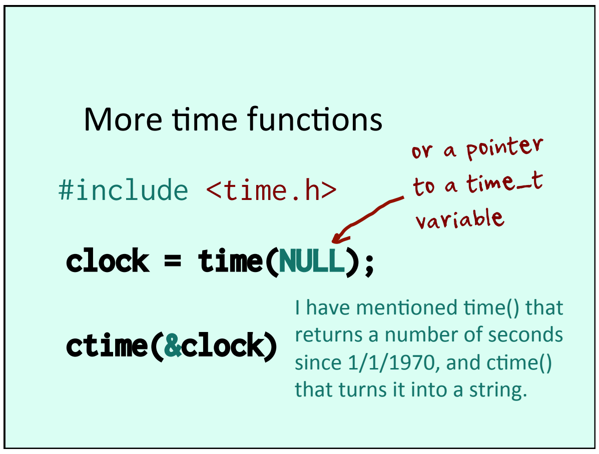


COMING SOON ...

A **struct** can contain any type of data, including pointers, and including other **structs** (they can be nested).

Practical use of structures

We are going to see a practical example of structures with time functions. We have seen very few time functions so far, because most time functions use structures that break a date and time into its components (year, month, day, and so forth). Many C functions take structures as arguments, or return pointers to structures.

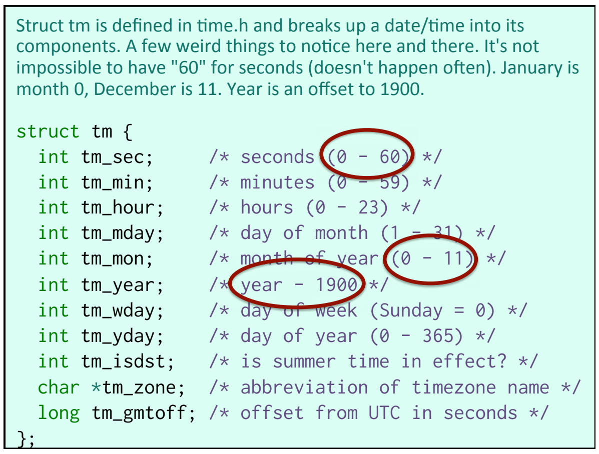


More time functions

```
#include <time.h>

clock = time(NULL);

ctime(&clock)
```

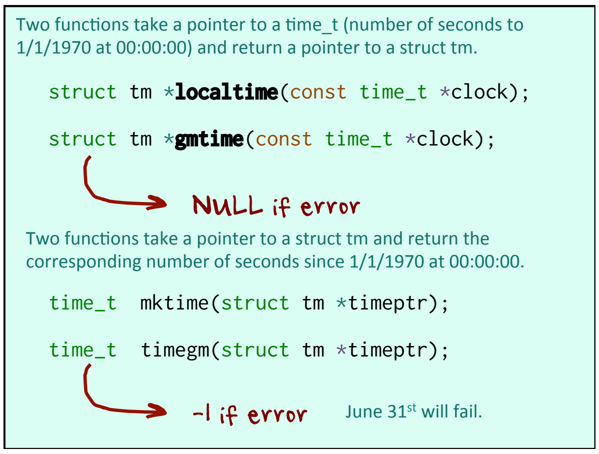
or a pointer to a time_t variable

I have mentioned time() that returns a number of seconds since 1/1/1970, and ctime() that turns it into a string.



Struct tm is defined in time.h and breaks up a date/time into its components. A few weird things to notice here and there. It's not impossible to have "60" for seconds (doesn't happen often). January is month 0, December is 11. Year is an offset to 1900.

```
struct tm {
  int tm_sec;     /* seconds (0 - 60) */
  int tm_min;     /* minutes (0 - 59) */
  int tm_hour;    /* hours (0 - 23) */
  int tm_mday;    /* day of month (1 - 31) */
  int tm_mon;     /* month of year (0 - 11) */
  int tm_year;    /* year - 1900 */
  int tm_wday;    /* day of week (Sunday = 0) */
  int tm_yday;    /* day of year (0 - 365) */
  int tm_isdst;   /* is summer time in effect? */
  char *tm_zone;  /* abbreviation of timezone name */
  long tm_gmtoff; /* offset from UTC in seconds */
};
```



Two functions take a pointer to a time_t (number of seconds to 1/1/1970 at 00:00:00) and return a pointer to a struct tm.

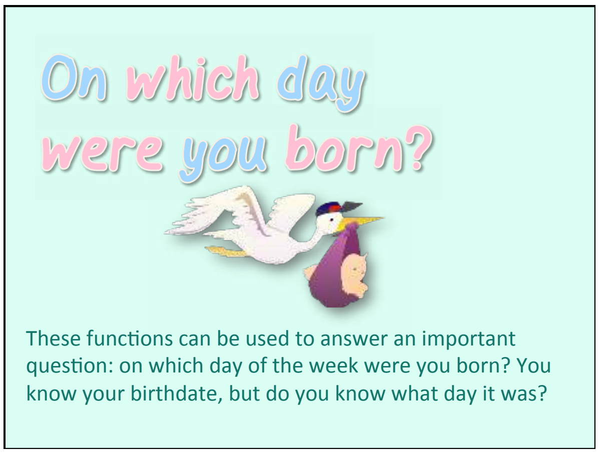
```
struct tm *localtime(const time_t *clock);

struct tm *gmtime(const time_t *clock);
```

NULL if error

Two functions take a pointer to a struct tm and return the corresponding number of seconds since 1/1/1970 at 00:00:00.

```
time_t  mktime(struct tm *timeptr);

time_t  timegm(struct tm *timeptr);
```

-1 if error   June 31st will fail.



On which day were you born?

These functions can be used to answer an important question: on which day of the week were you born? You know your birthdate, but do you know what day it was?

A bit tricky …

1. Populate a `struct tm` with known information
2. Have the library convert it to seconds since epoch
3. Convert back to `struct tm` – all fields populated

The program is a bit tricky because there are three steps. Month, day of the month and year are enough data to compute a number of seconds since 1970, if you assume midnight. By converting back this number of seconds to a struct tm, the system will fill in the gaps and provide the missing information - including the day of the week.

whichday.c

```
#include <stdio.h>
#include <time.h>
#include <errno.h>

int main(int argc, char *argv[]) {
  struct tm  birthdate;   This will hold the information you know
  struct tm *bdp;          This will point to information computed
  time_t clock = (time_t)0;      by the system
  int    mon;
  int    day;    Those three ints will store your birthdate
  int    year;
  char  *days[] = {"Sunday", "Monday", "Tuesday",
                   "Wednesday", "Thursday", "Friday",
                   "Saturday"};
```

As in struct tm the day of the week is a value between 0 and 6 (0 = Sunday), we need a (constant) array of strings with the corresponding day name.

whichday.c

Check the command line. We want the date passed as the program is called (it could also be prompted for and read from inside the program)

```
  if (argc != 2) {
     fprintf(stderr, "Usage: %s YYYY/MM/DD\n",
             argv[0]);
     return 1;
  }
```

Read the date into separate variables for month, day and year

```
  if (3 != sscanf(argv[1], "%d/%d/%d",
                  &year, &mon, &day)) {
     fprintf(stderr,
             "Invalid date %s (YYYY/MM/DD expected)\n",
             argv[1]);
     return 1;
  }
```

whichday.c

```
  birthdate.tm_mon = mon - 1;
  birthdate.tm_mday = day;
  birthdate.tm_year = year - 1900;
  birthdate.tm_hour = 0;
  birthdate.tm_min = 0;
  birthdate.tm_sec = 0;
  birthdate.tm_wday = 0;
  birthdate.tm_yday = 0;
  birthdate.tm_gmtoff = 0;
  if ((clock = mktime(&birthdate)) == (time_t)-1) {
     perror(argv[1]);
     return 1;
  }
  bdp = localtime(&clock);
  printf("%s is a %s\n", argv[1], days[bdp->tm_wday]);
  return 0;
}
```

Populate the structure with what you know, set everything else to zero.

Convert to seconds. It can fail, people may have wrongly entered another date format.

Make the system convert back to struct tm

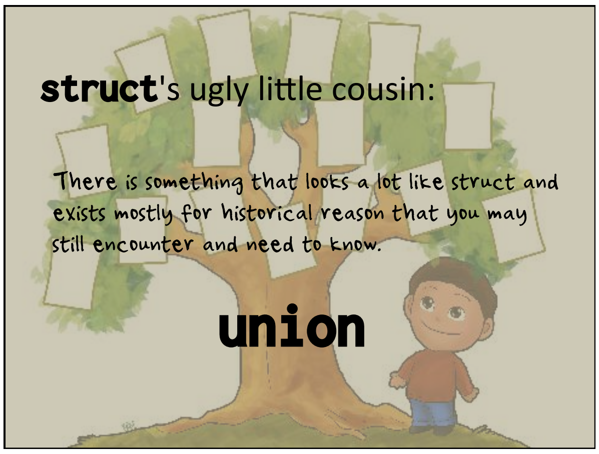
Et voilà.

Use the tm_wday value as index in our day name array

What it gives in practice in a Unix-like environment.

```
$ ./whichday 1995/09/29
1995/09/29 is a Friday
$
```

(Once again, you could also say "Enter your birthdate" in the program, read a string, and use sscanf with two s to read month, day and year from the string) .



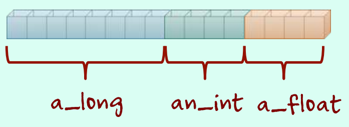
struct's ugly little cousin:

There is something that looks a lot like struct and exists mostly for historical reason that you may still encounter and need to know.

union

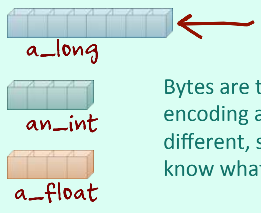
```
struct {
        long  a_long;
        int   an_int;
        float a_float;
    } stuff;
```

When you are declaring a struct variable, you are reserving memory for all the elements defined in the structure (plus, possibly, phantom bytes for alignment)



```
union  {
        long  a_long;
        int   an_int;
        float a_float;
    } stuff;
```

In a union, all components are using the same storage in memory



Reserved size is the size of the biggest

Same bytes in memory, wearing different hats.

Bytes are the same ones, but encoding and meaning are different, so you'd better know what you are doing!

You can store information about a rectangular pool as width and length in feet.

```
struct pool pool1;

pool1.shape = 'R';
pool1.depth = 130;
pool1.dim.rect.w = 300;
pool1.dim.rect.l = 600;
```

Notice the multiple dots because of nested structs/unions

Or information about a circular pool as a radius in feet. All using the same bytes in memory. The shape tells you how to understand the dimensions.

```
struct pool pool2;

pool2.depth = 105;
pool2.shape = 'C';
pool2.dim.radius = 130;
```

When memory isn't an issue, it's more understandable to have separate storage for rectangular dimensions and radius, and only use what you need, "wasting" a few bytes.

FILES

A few words about files. To process data, obviously you must input it. The keyboard is one way to do it, but kind of tedious for huge amounts of data. Historically, data files have been very important. They are still much used today, although not always in the same way as 40 or 50 years ago, when files were ruling the IT world. Today a lot of data comes through networks, or is read from, and written to, databases (you'll see that later).
Files are simpler in C than in Java.

Stream redirection

```
$ my_program < input_file
$ my_program > output_file
```

One easy way to read from a file or write to a file is to use "stream redirection". You can make the computer feed data from a file into your program as if it were coming from the keyboard, or make it write to a file what you thought would go to the screen. Your program only needs to know "stdin" and "stdout". Both redirections can be combined. It usually works with text files.

How do we handle a file in a program?

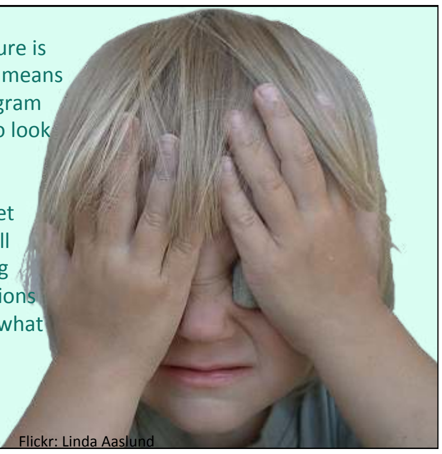
stdio.h

FILE

File management is one of the most standard areas of C and all functions, constants and required structures are defined in stdio.h

"Opaque" structure

The FILE structure is opaque, which means that YOUR program doesn't need to look at its content. Your program just needs to get a FILE * that will be passed along to library functions that will know what to do with it.


Flickr: Linda Aaslund

We only manipulate POINTERS to these structures.

FILE *fp = Handler

Stream

A file pointer is a stream exactly like stdin or stdout. In fact, wherever you can use stdin or stdout, you can replace them with a file pointer.

```
FILE *fp;
```

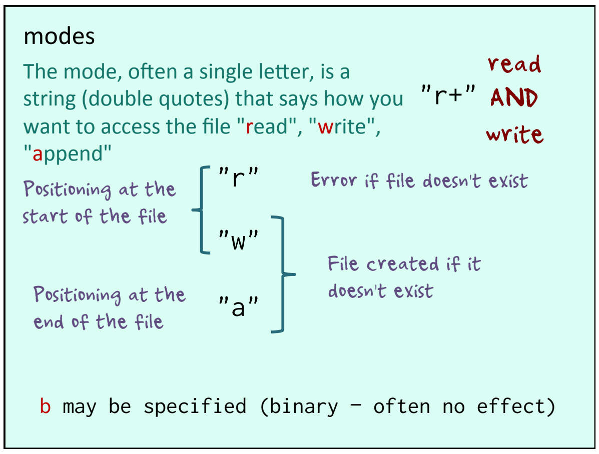
Function fopen() returns to you a file pointer to a file that you name, or NULL if it fails to open the file.

```
if ((fp = fopen(file_name, mode)) != NULL) {
```

It's common for fopen() to fail: you try to read from a non-existent file, or write to a file that you can only read from, etc.

```
   fclose(fp);
} else {
   perror(file_name);
   return 1;
}
```

Flushes everything and closes properly



modes

The mode, often a single letter, is a string (double quotes) that says how you want to access the file "read", "write", "append"

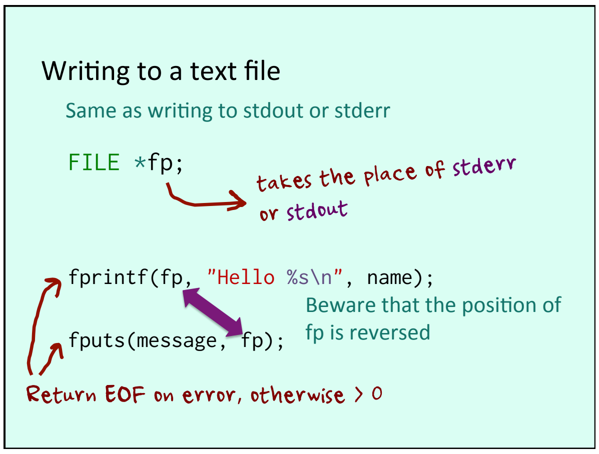
"r+" read AND write

Positioning at the start of the file
- "r" Error if file doesn't exist
- "w"

Positioning at the end of the file
- "a"

"w", "a": File created if it doesn't exist

`b may be specified (binary - often no effect)`



Writing to a text file

Same as writing to stdout or stderr

```
FILE *fp;
```

fp takes the place of stderr or stdout

```
fprintf(fp, "Hello %s\n", name);
fputs(message, fp);
```

Beware that the position of fp is reversed

Return EOF on error, otherwise > 0