Structs (1)

In data processing, a collection of data that can be treated as a single unit is a record. The components of this collection (fields or members or attributes) are uniquely named and have values. The values may be numbers (integers or floating point), texts, etc. and also other collections.

For example, data set presenting a student is a record. The attributes are his/her name (text), date of birth (it is itself a record and its attributes are day, month and year), number of collected points (integer), average mark (floating point number), etc. Records with the same set of attributes belong to the same record type.

In C the records are called as *structs*. To create and process a record we have at first declare the record type. Example:

```
struct Date
{ // declaring a new data type – record presenting the date
  int day, // attribute day, an integer
      month, // attribute month, an integer
      year; // attribute year, an integer
};  // semicolon needed
```

The declaration of a new *struct* type neither defines a variable nor allocates memory. It just explains to the compiler the meaning of words *Date*, *day*, *month*, *year*. The *struct* declarations are in most cases located at the beginning of source file or put into the project header file.

Structs (2)

Generally:

struct <struct_type_name> { <declarations_of_attributes> };

The attributes may be of different types. Examples:

```
struct Date
{
   int day; // attribute day, an integer
   char month[4]; // attribute month, array for strings like "Jan", "Feb", "Mar", etc.
   int year; // attribute year, an integer
};
```

```
struct Date
{
   int Day; // attribute Day, an integer
   char *pMonth; // attribute pMonth, pointer to string like "January", "February", etc.
   int Year; // attribute Year, an integer
};
```

```
struct Student
{
   const char *pName; // attribute pName, pointer to string presenting the name
   struct Date Birthdate; // attribute Birthdate, struct Date nested into struct Student
   double AverageMark; // attribute AverageMark, floating point value
};
```

Structs (3)

After declaring a new *struct* type we can declare variables of that type:

struct <struct_type_name> <list_of_variable_names>;

Example:

```
struct Date yesterday, today, tomorrow;
struct Student guy;
```

To access attributes of a *struct* variables use expression with point operator:

<struct_variable_name>.<attribute_name>

Examples:

```
today.day = 25;
tomorrow.day = today.day + 1;
today.year = 2018;
strcpy(today.month, "Oct"); // here we mean that the attribute is char month[4]
if (today.day == 31)
{
    today.day = 1;
    strcpy(today.month, "Nov");
}
guy.AverageMark = 4.14;
guy.Birthdate.day = 25; // nested struct, use point operator twice
strcpy(guy.Birthdate.month, "Oct");
printf("Student: %s: average mark %.2f\n", guy.pName, guy.AverageMark);
```

Structs (4)

The *struct* variables may be initialized:

struct <struct_type_name> <variable_name> = { <list_of_attribute_values> };

Examples:

```
struct Date today = { 26, "Oct", 2016 };
struct Student guy_1 = { "Al Capone", { 26, "Oct", 2008 }, 4.14 }; // nested structs
struct Student guy_2 = { "John Dillinger" }; // only the first attribute has value, the others
                                             // are automatically initialized to zero
struct Student guy_3; // no initalization, values of attributes are garbage (not zeroes)
```

From C standard 1999 (C99) it is possible to initialize any selection of attributes using the designated initializers like:

```
struct Student guy_2 = { .pName = "John Dillinger", .AverageMark = 4.14 };
```

Declaring of *stuct* type, variables of this type as well as their initialization can be concentrated into one statement, for example:

```
struct Point
{
   int x,
       y;
} p1 = { 0, 0 }, p2 = { 1,1 }; // you may later use the declaration of Point for other variables
```

Structs (5)

In C, the *struct* type name must always preceded by keyword *struct*. In C++ it is not obligatory, so

```
struct Date today = { 26, "Oct", 2016 }; // legal both in C and C++
Date today = { 26, "Oct", 2016 }; // legal in C++
```

However, in C there is a possibility to get rid of repeating the *struct* keyword:

```
typedef struct date DATE;
```

typedef does not create a new type, it simply creates an alias for existing types. Examples:

```
DATE today; // the same as struct Date today
typedef DWORD unsigned long int;
DWORD i; // the same as unsigned long int i
```

Declaration of a new type and assigning to it the *typedef* may be done in one statement, like

```
typedef struct Date
{
    int day;
    char month[4];
    int year;
} DATE;
```

Good programming practice: *typedef* names should be in uppercase letters.

Structs (6)

Arrays of *structs* are declared in well-known way:
struct <struct_type_name> <array_name>[dimension];

To access an attribute of an array element write expression:
<array_name>[<index>].<attribute_name>

To declare an array of *structs* with initalizations write:

```
struct <struct_type_name> <array_name>[dimension] =
{
  { <intial_values_for_element_0> },
  { <intial_values_for_element_1> },
    ……..……..……..……..……..……..
  { <intial_values_for_element_n> }
};
```

Assignment between *structs* of the same type is allowed. The complete contents of one struct is copied into the other.

Address-of operator (&), dereference operator (*) and *sizeof* operator for *structs* are allowed. Arithmetical and logical operations between *structs* are not defined. The comparison operations are also not possible.

A *struct* can be a formal as well as an actual parameter of a function. When the function is called, the contents of actual parameter is copied into the formal parameter.

Structs (7)

Examples:

```
struct Date October[31];
for (int i = 0;  i < 31;  i++)
{
   October[i].day = i + 1;
   strcpy(October[i].month, "Oct");
   October[i].year = 2019;
}
Student group[3] =
{
    { "Al Capone", { 26, "Oct", 2008 }, 4.14 },
    { "Bonnie Parker", { 25, "Nov", 2009 }, 3.14 },
    { "Clyde Barrow", { 20, "Dec", 2007 }, 2.14 }
};
struct Date today = { 26, "Oct", 2019};
struct Date tomorrow;
tomorrow = today; // assignment, the same as
                  // memcpy(&tomorrow, &today, sizeof (struct Date));
tomorrow.day++;
```

Structs (8)

Exercise:

```
typedef struct Date
{
  int day;
  char month[4];
  int year;
} DATE;

typedef struct Exam
{
  const char *pSubject;
  DATE date;
  int mark;
} EXAM;
```

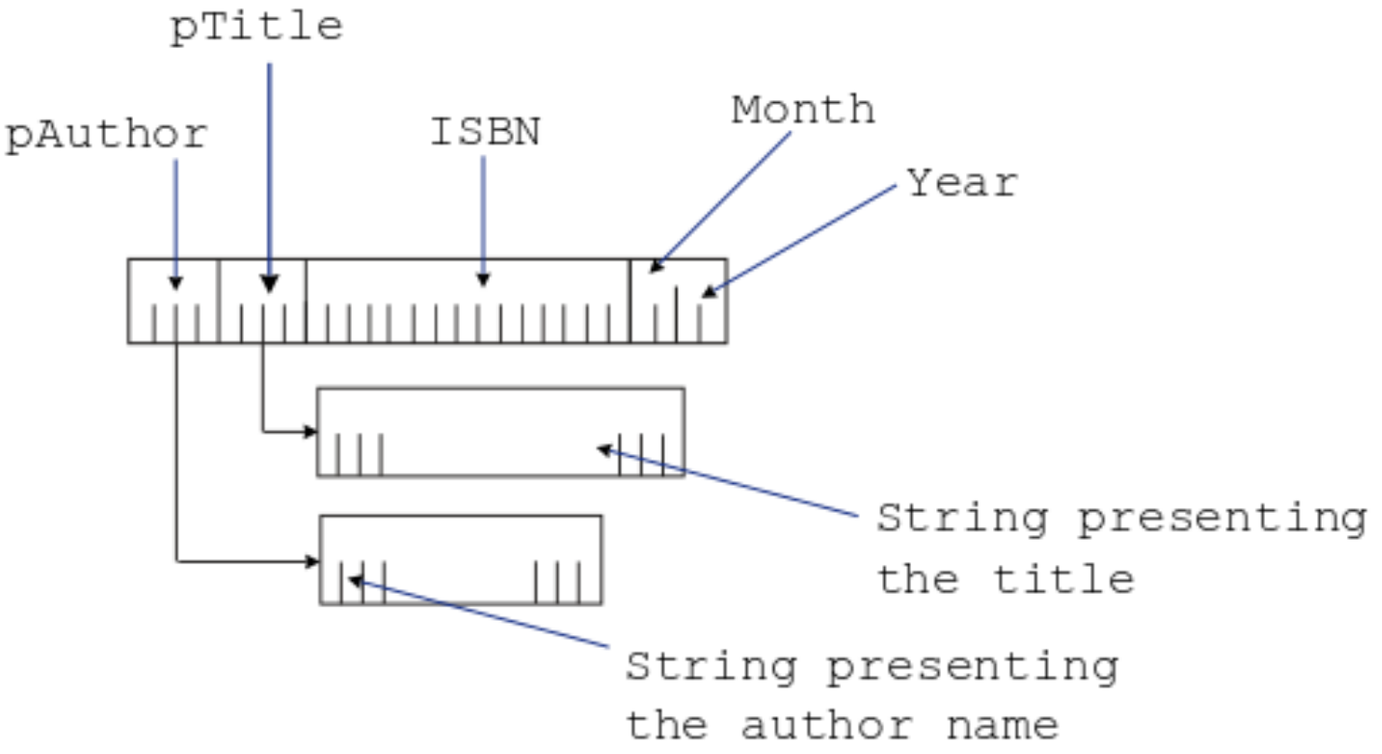
Write a program that:

1. Creates an array presenting all your examinations on this semester and initializes it. As the dates and marks are not known yet, fantasize.
2. Prints the array. The *printf* format string must be *"%s at %d-%s-%d, mark is %d\n"*
3. Calculates and prints the average mark.

Structs (9)

Let us have

```
struct Date
{
    short int Month,
              Year;
};
struct Book
{
    char *pAuthor,
         *pTitle;
    char ISBN[15];
    struct Date Edition;
};
```




```
struct Book textbook =
{ "Greg Perry & Dean Miller", "C programming. Absolute Beginners Guide",
  "978-0789751980", { 8, 2013} }; // errors, pointers to author and title are not constants
```

Structs (10)

```
struct Book
{
    const char *pAuthor,
               *pTitle;
    char ISBN[15];
    struct Date Edition;
};
```

or

```
struct Book
{
   char Author [100], // fixed length is bad solution
        Title [100];
    char ISBN[15];
    struct Date Edition;
};
```

```
struct Book textbook =
{ "Greg Perry & Dean Miller", "C programming. Absolute Beginners Guide",
   "978-0789751980", { 8, 2013} }; // now correct
```

To get flexible software we have to allocate memory for two strings located outside of the struct body.

Structs (11)

```
void ProcessBook(const char *pAuthor, const char *pTitle, const char *pISBN,
                 const struct Date edition)
{
  struct Book textbook; // local variable, exists only when the function is running
  textbook.pAuthor = (char *)malloc(strlen(pAuthor) +1); // allocate memory for author
  strcpy(textbook.pAuthor, pAuthor); // copy the author's name
  textbook.pTitle = (char *)malloc(strlen(pTitle) +1);
  strcpy(textbook.pTitle, pTitle);
  strcpy(textbook.ISBN, pISBN);
  textbook.Edition = edition;
  …….. // do something
  free(textbook.pAuthor); // do not forget to release memory, textbook as local variable is
                          // destroyed automatically, but dynamically allocated memory
                          // must be released by us
   free(textbook.pTitle);
}
```

Call example:

```
const struct Date when = { 8, 2013 }; // cannot change the values of attributes later
ProcessBook("Greg Perry & Dean Miller", "C programming. Absolute Beginners Guide",
"978-0789751980", when) ;
```

Structs (12)

Exercise:

```
typedef struct Date
{
    int day;
    char *pMonth; // full name, locates on its own memory field that must be allocated
    int year;
} DATE;

typedef struct Exam
{
    char *pSubject; // locates on its own memory field that must be allocated
    DATE date;
    int mark;
} EXAM;
```

Write a program that:

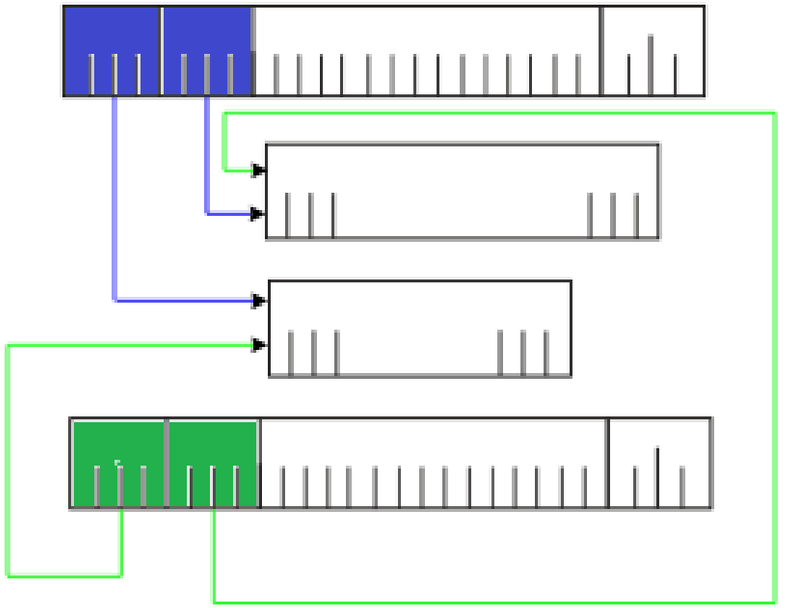
1. Creates an array presenting all your examinations on this semester and intializes it. As the dates and marks are not known yet, fantasize.
2. Prints the array. The *printf* format string must be *"%s at %d-%s-%d, mark is %d\n"*
3. Calculates and prints the average mark.
4. Before exit releases the allocated memory.

Structs (13)

Let us have:

```
struct Book textbook1, textbook2;
textbook2 = textbook1;
```

This is extremely dangerous, because now *textbook1* and *textbook2* are sharing the strings presenting the author and title.



Assignment is just a simple bytewise copy. So the pointer to *textbook1* author is copied into *textbook2.pAuthor* field.
Suppose we want to change the author of *textbook1*:

```
free(textbook1.pAuthor);
textbook1.pAuthor = (char *)malloc(strlen("Stephen Prata") + 1);
strcpy(textbook1.pAuthor, "Stephen Prata");
```

But now *textbook2.pAuthor* points to a memory field that is released: we have lost the author of *textbook2*.

Structs (14)

Solution:

```
struct Book textbook1, textbook2;
textbook2 = textbook1;
textbook2.pAuthor = (char *)malloc(strlen(textbook1.pAuthor) + 1);
                    // allocate new memory
strcpy(textbook2.pAuthor, textbook1.pAuthor);
                    // make a copy
textbook2.pTitle = (char *)malloc(strlen(textbook1.pTitle) + 1);
strcpy(textbook2.pTitle, textbook1.pTitle);
```

Now *textbook1* and *textbook2* can be handled separately..

Structs (15)

Let us have:

```
struct Date
{
  int day;
  char month[4];
  int year;
};
struct Date Today; // Today is a local or global variable, its visibility and lifetime are
                   // specified by C standard. After declaration we may work with its
                   // attributes
struct Date *pToday; // pToday is not the struct but only a pointer to it. The struct does
                     // not exist yet, to work with it we must at first allocate the memory
pToday = (struct Date *)malloc(sizeof(struct Date));
                     // Now the struct has memory.
                     // Never try to count bytes in a struct, use sizeof
Today.day = 2;                          pToday->day = 2;
Today.year = 2018;                      pToday->year = 2018;
strcpy(Today.month, "Nov");             strcpy(pToday->month, "Nov");
```

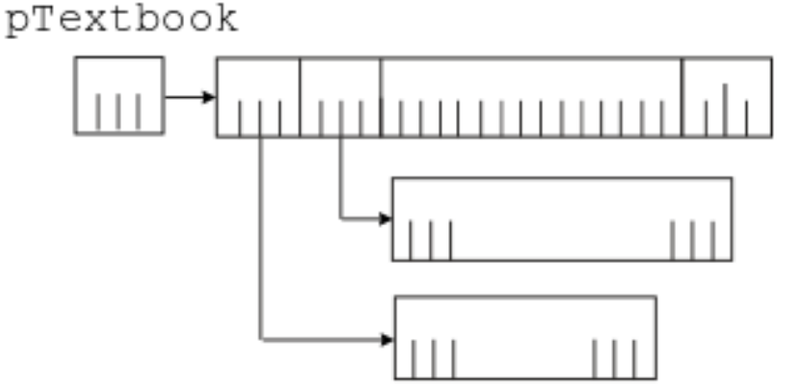
There are two different expressions to access the attributes of structs:

<name_of_struct_variable>.<attribute_name> // point operator

<pointer_to_struct_variable>-><atribute_name> // arrow operator

Structs (16)

```
void ProcessBook(const char *pAuthor, const char *pTitle, const char *pISBN,
                 const struct Date edition)
{ // compare with code on slide Structs (11)
  struct Book *pTextbook = (struct Book *)malloc(sizeof(struct Book));
                                                  // allocate memory for the main  body of struct
  pTextbook->pAuthor = (char *)malloc(strlen(pAuthor) +1); // allocate memory for author
  strcpy(pTextbook ->pAuthor, pAuthor); // copy the author's name
  pTextbook ->pTitle = (char *)malloc(strlen(pTitle) +1);
  strcpy(pTextbook ->pTitle, pTitle);
  strcpy(pTextbook ->ISBN, pISBN);
  pTextbook->Edition.Month = edition.Month; // Attention: both -> and .
  pTextbook->Edition.Year = edition.Year;
  …….. // do something
  free(pTextbook->pAuthor);
                  // do not forget to release memory
  free(pTextbook->pTitle);
  free(pTextbook); // must be the last release
}
```



Structs (17)

```
struct Book *pTextbooks = (struct Book *)malloc(n * sizeof(struct Book));
                                                    // dynamically allocated array of n books
```

pTextbooks is the pointer to array of *n* structs.
pTextbooks + i points to the i-th element in the array.
(pTextbooks + i)->pAuthor gives the pointer to the author of i-th book.
Parentheses are needed because the precedence of -> is higher than the precedence of addition but we need to execute the addition first.

Example: printing of data of the i-th book:

```
printf("%s\n", (pTextbooks + i)->pAuthor);
printf("%s\n", (pTextbooks + i)->pTitle);
printf("%s\n", (pTextbooks + i)->ISBN);
printf("%d\n", (pTextbooks + i)->Edition.Month);
printf("%d\n", (pTextbooks + i)->Edition.Year);
```

Syntactic shorthand like *pTextbooks[i].pAuthor* is also applicable.

Structs (18)

Exercise:

```
typedef struct Date {
  int day;
  char month[4];
  int year;
} DATE;
```

```
typedef struct Exam {
  char *pSubject;
  DATE date;
  int mark;
} EXAM;
```

Write a function with prototype

```
EXAM *MySession(int *pnExams);
```

that creates a dynamically allocated array presenting all your examinations on this semester, initializes it and returns the pointer to it.

Write also *main()* that prints the array, calculates and prints the average mark and before exit releases all the allocated memory. Example code snippet for *main()*:

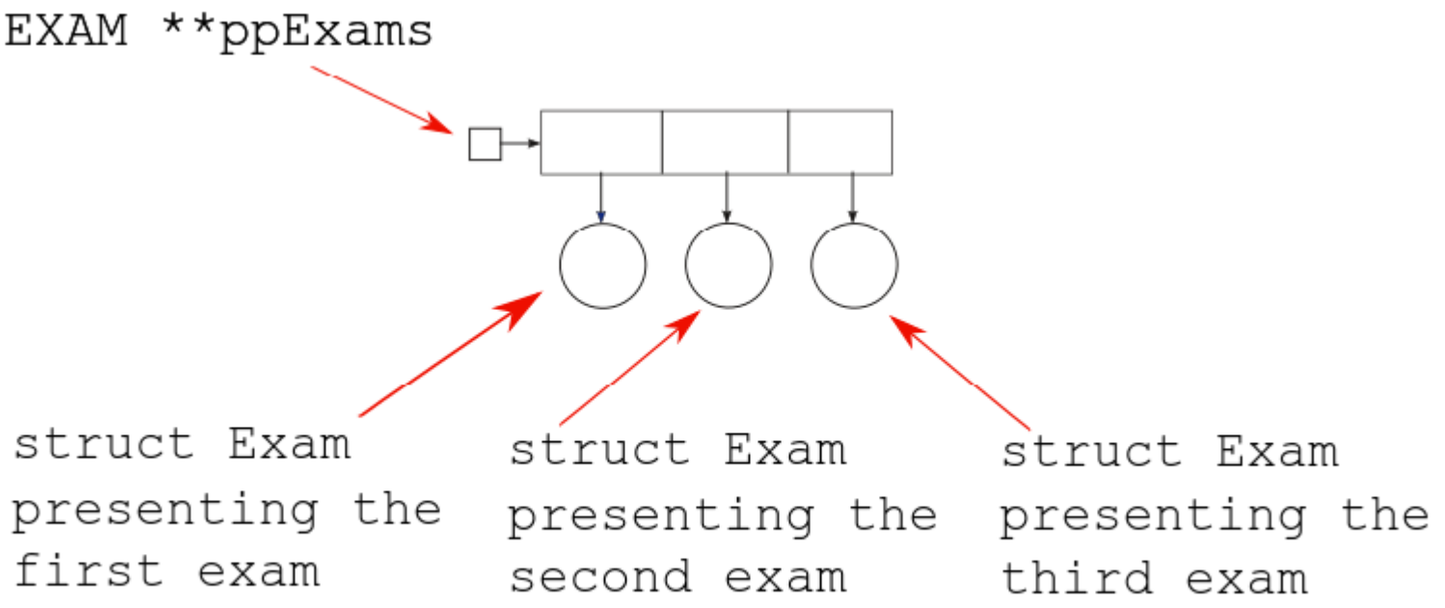
```
int nExams; // the value of this variable (i.e. number of exams) is set by function MySession
EXAM *pMyExams = MySession(&nExams);
```

Structs (19)

Exercise: write a function with prototype

EXAM **MySession(int *pnExams);

that creates a data structure similar to the following figure and returns the pointer to it:



Write also *main()* that prints the array, calculates and prints the average mark and before exit releases all the allocated memory. Example code snippet for *main()*:

```
int nExams; // the value of this variable (i.e. number of exams) is set by function MySession
EXAM **ppMyExams = MySession(&nExams);
for (int i = 0;  i< nExams; i++)
    printf("%s\n", (*(ppMyExams + i))->pSubject); // prints all the subjects
```

Structs: crib sheet (1)

```
struct Date {
    int Day; // attribute Day, an integer
    char Month[4]; // attribute Month, strings "Jan", "Feb", etc.
    int Year; // attribute Year, an integer
};
struct Student {
    char *pName; // attribute pName, pointer to separate memory field presenting the name
    struct Date Birthdate; // attribute Birthdate, struct Date nested into struct Student
    double AverageMark; // attribute AverageMark, floating point value
};
void fun() {
   struct Student Guy; // local variable, memory allocated automatically
   Guy.pName = (char*)malloc(strlen("John Smith") + 1));
                                // memory for name, to be allocated by us
   strcpy(Guy.pName, "John Smith");
   Guy.Birthdate.Day = 9;
   strcpy(Guy.Birthdate.Month, "Dec");
   Guy.Birthdate.Year = 2002;
   Guy.AverageMark = 4.5;
   free(Guy.pName); // variable Guy is removed automatically, but memory allocated by
                              // malloc must be removed by us
}
```

Structs: crib sheet (2)

```
struct Date {
    int Day; // attribute Day, an integer
    char Month[4]; // attribute Month, strings "Jan", "Feb", etc.
    int Year; // attribute Year, an integer
};
struct Student {
    char *pName; // // attribute pName, pointer to separate memory field presenting the name
    struct Date Birthdate; // attribute Birthdate, struct Date nested into struct Student
    double AverageMark; // attribute AverageMark, floating point value
};
void fun(){
   struct Student *pGuy = (struct Student *)malloc(sizeof(struct Student)); // memory for struct
   pGuy->pName = (char*)malloc(strlen("John Smith") + 1)); // memory for name
   strcpy(pGuy->pName, "John Smith");
   pGuy->Birthdate.Day = 9;
   strcpy(pGuy->Birthdate.Month, "Dec");
   pGuy->Birthdate.Year = 2002;
   pGuy->AverageMark = 4.5;
   free(pGuy->pName);
   free pGuy;
}
```

Structs: crib sheet (3)

```
struct Date {
    int Day; // attribute Day, an integer
    char Month[4]; // attribute Month, strings "Jan", "Feb", etc.
    int Year; // attribute Year, an integer
};
struct Student {
    char *pName; // attribute pName, pointer to separate memory field presenting the name
    struct Date *pBirthdate; // attribute pBirthdate, pointer to separate memory field
    double AverageMark; // attribute AverageMark, floating point value
};
void fun(){
   struct Student Guy;
   Guy.pName = (char*)malloc(strlen("John Smith") + 1)); // memory for name
   strcpy(Guy.pName, "John Smith");
   Guy.pBirthdate = (struct Date *)malloc(sizeof(struct Date)); // memory for birthdate
   Guy.pBirthdate->Day = 9;
   strcpy(Guy.pBirthdate->Month, "Dec");
   Guy.pBirthdate->Year = 2002;
   Guy.AverageMark = 4.5;
   free(Guy.pName);
   free(Guy.pBirthdate);
}
```

Structs: crib sheet (4)

```
struct Date {
    int Day, Year; // attributes Day and Year, integers
    char Month[4]; // attribute Month, strings "Jan", "Feb", etc.
};
struct Student {
    char *pName; // // attribute pName, pointer to separate memory field presenting the name
    struct Date *pBirthdate; // attribute pBirthdate, pointer to separate memory field
    double AverageMark; // attribute AverageMark, floating point value
 };
void fun(){
   struct Student *pGuy = (struct Student *)malloc(sizeof(struct Student)); // memory for struct
   pGuy->pName = (char*)malloc(strlen("John Smith") + 1)); // memory for name
   strcpy(pGuy->pName, "John Smith");
   pGuy-> pBirthdate = (struct Date *)malloc(sizeof(struct Date)); // memory for birthdate
   pGuy->pBirthdate->Day = 9;
   strcpy(pGuy->pBirthdate->Month, "Dec");
   pGuy->pBirthdate->Year = 2002;
   pGuy->AverageMark = 4.5;
   free(pGuy->pName);
   free(Guy.pBirthdate);
   free pGuy;
}
```

Operator precedence (1)

Precedence	Operator	Description	Associativity
1	++ and -- ( ) [ ] . ->	Increment and decrement, postfix Function call Reading element from array Structure member access Structure member access through pointer	Left -> Right
2	++ and -- - ! (type) * & sizeof	Increment and decrement, prefix Sign conversion Logical NOT Type cast Dereference Address-of Size-of	Right->Left
3	* / %	Multiplication Division Modulus	Left -> Right
4	+ -	Addition Subtraction	Left -> Right

Operator precedence (2)

Precedence	Operator	Description	Associativity
5	<= < >= >	Less or equal Less Greater or equal Greater	Left -> Right
6	== !=	Equal Not equal	Left -> Right
7	&&	Logical AND	Left -> Right
8	||	Logical OR	Left -> Right
9	?:	Conditional	Right->Left
10	= += -= *= /* %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right->Left
11	,	Comma	Left -> Right

Time (1)

Reading the current time from the system clock:

```
#include "time.h"
time_t now; // time_t is specified by typedef, in Visual Studio it is is a 64-bit integer
time(&now); // the number of seconds since January 1, 1970, 0:00 UTC
```

To get the current date and time understandable for humans use the standard *struct tm*:

```
struct tm // do not declare it in your code, it is already declared in time.h
{
   int tm_sec;   // seconds after the minute - [0, 60] including leap second
   int tm_min;   // minutes after the hour - [0, 59]
   int tm_hour;  // hours since midnight - [0, 23]
   int tm_mday;  // day of the month - [1, 31]
   int tm_mon;   // months since January - [0, 11], attention: January is with index 0
   int tm_year;  // years since 1900, attention, not from the birth of Christ
   int tm_wday;  // days since Sunday - [0, 6], attention: Sunday is with index 0, Monday 1
   int tm_yday;  // days since January 1 - [0, 365]
   int tm_isdst; // daylight savings time flag
};
```

To fill this struct:

```
struct tm date_time_now;
localtime_s(&date_time_now, &now);
```

Time (2)

Example:

```
printf("Today is %d.%d.%d\n",
   date_time_now.tm_mday, date_time_now.tm_mon + 1,  date_time_now.tm_year + 1900);
```

Function *asctime_s* converts the *struct tm* to string:

```
char buf[100];
asctime_s(buf, 100, &date_time_now);
printf("%s\n", buf); // prints like Fri Nov 2 17:21:51 2018
```

but here we cannot set the format. Better is to use function *strftime*, for example:

```
strftime(buf, 100, "%H:%M:%S %d-%m-%Y", &date_time_now);
      // prints according to Estonian format 17:21:51 02-11-2018
```

The complete reference of *strftime* is on http://www.cplusplus.com/reference/ctime/strftime/

The attributes of *struct tm* may be modified. For example, if we want to know what date is after 100 days, do as follows:

```
struct tm date_time_future = date_time_now;
date_time_future.tm_mday += 100; // add 100 days
time_t future = mktime(&time_date_future); // convert back to time_t
localtime_s(&date_time_future, &future); // convert once more to struct tm
asctime_s(buf, 100, &date_time_future);
printf("%s\n", buf); // prints like Sun Feb 10 17:21:51 2018
```

Time (3)

We may create our own *struct tm*. Example:

The ship departures on January 31 2020 at 13:20. It takes 2 days and 8.5 hours to reach Copenhagen. Find the arrival date and time.

```
struct tm departure = { 0, 20, 13, 31, 0, 120 };
// mktime ignores tm_wday and tm_yday, so here we can set them to zero.
// do not forget that tm_year must be the year from 1900
struct tm arrival = departure;
arrival.tm_hour += 8;
arrival.tm_min += 30;
arrival.tm_mday += 2;
time_t arrive_t = mktime(&arrival);
localtime_s(&arrival, &arrive_t);
char buf[100];
strftime(buf, 100, "%H:%M:%S %d-%m-%Y", &arrival);
printf("%s\n", buf); // prints 21:50:00 02-02-2020
```

Files (1)

To work with a disk file, our first task is to open it:

FILE <pointer_to_struct_typedefed_as_FILE> = fopen(<filename_as_string_constant>, <mode_as_string_constant>);

Example:

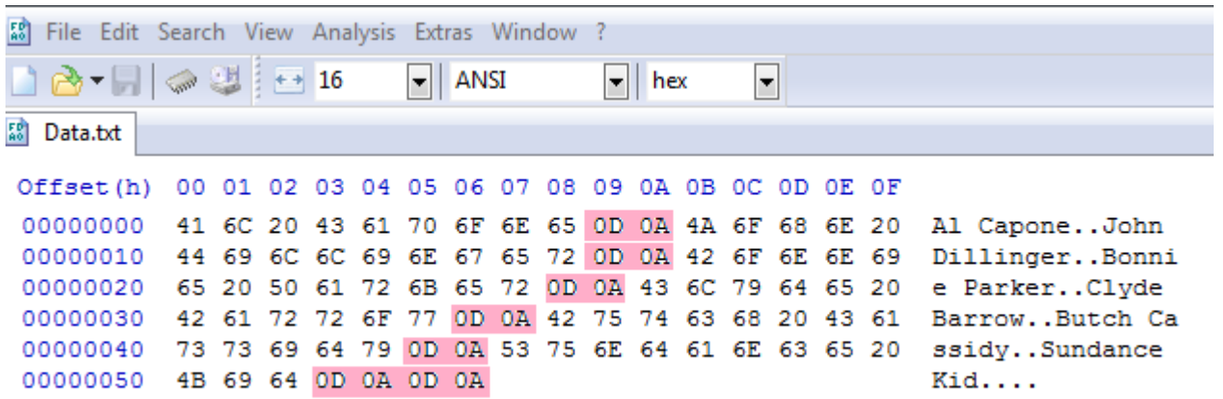
```
FILE *pFile = fopen("c:\\temp\\data.txt", "wt+"); // open text file data.txt for writing and
                                                  // reading
```

struct with typedef name *FILE* is defined in *stdio.h*. We do not need to know its attributes.

To avoid problems specify the complete path to the file. Do not forget that backslash as character constant is '\\'.

Binary files (character *'b'* in mode string) are handled as byte sequences. Text files (character *'t'* in mode string) consist of rows of text. Each row is terminated by two characters: carriage return or CR or *'\r'* (0x0D) and line feed or LF or *'\n'* (0x0A).

To see the contents of file use freeware utility HxD (https://mh-nexus.de/en/hxd/):



Files (2)

The access modes are:

Mode	Access
"r"	For reading only. If the file was not found, *fopen* returns null pointer.
"r+"	For reading and writing. If the file was not found, *fopen* returns null pointer.
"w"	For writing only. If the file was not found, creates it. If the file already exists, deletes its contents.
"w+"	For reading and writing. If the file was not found, creates it. If the file already exists, deletes its contents.
"a"	For writing only. If the file was not found, creates it. If the file already exists, its contents is kept and the new data is appended.
"a+"	For reading and writing. If the file was not found, creates it. If the file already exists, its contents is kept and the new data is appended.

The *fopen* mode string must specify the file type (binary or text) as well as the access mode. Examples: *"rb"*, *"at+"* .

If you have finished the operations with file, close it:

```
fclose(pFile);
```

Files (3)

To write into a file use function *fwrite*:

<number_of_written_items> = fwrite(<pointer_to_data_to_write>, <size_of_data_item>, <number_of_items_to_write>, <pointer_to_FILE_struct>);

Example:

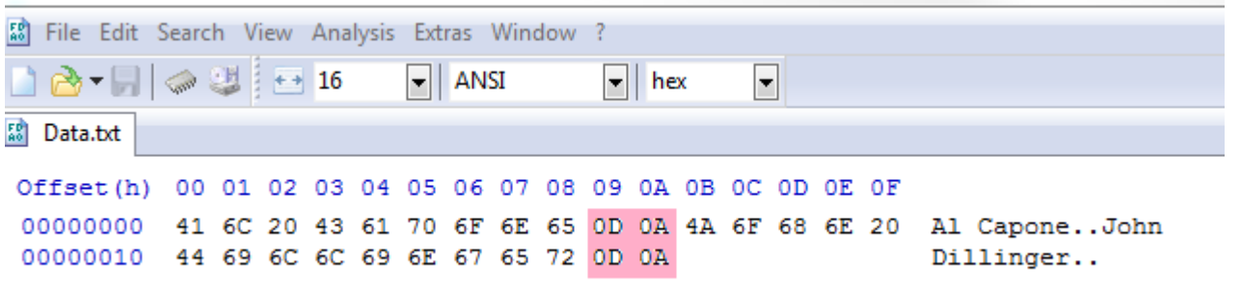
```
#pragma warning (disable: 4996) // for Visual Studio function fopen is unsafe. To
                                // suppress the compiler error message use this pragma
char *pData = (char *)malloc(100);
…… // fills the array with data
FILE *pFile = fopen("c:\\temp\\data.txt", "wt");
if (!pFile)
{  // Good programming practice: check always
    printf("Failure, the file was not open\n");
    return;
}
int n = fwrite(pData, 1, 100, pFile); // 100 characters, one byte each
if (n != 100)
{ // Good programming practice: check always
    printf("Failure, only %d bytes were written\n", n);
}
fclose(pFile);
```

Files (4)

If you want to store a string, remember that to mark the end of row use '\n' only. '\r' will be added automatically.

Example:

```
const char *pData[] = { "Al Capone\n", "John Dillinger\n" };
FILE *pFile = fopen("C:\\Temp\\Data.txt", "wt+");
if (pFile)
{
	fwrite(pData[0], 1, strlen(pData[0]), pFile);
			// Stores without terminating zero.
			// fwrite(pData[0], 1, strlen(pData[0]) + 1, pFile); // with terminating zero
	fwrite(pData[1], 1, strlen(pData[1]), pFile);
	fclose(pFile);
}
```



Files (5)

Function *fwrite* may not store the data immediately. There is an inaccessible for us system buffer and the data is collected into it. The writing is automatically performed when the buffer is full. In this way time is economized. Function *fflush* forces the system to perform the writing immediately:

fflush(<pointer_to_FILE_struct>);

The file has an associated with it inner pointer that specifies the location to where the first written byte will be placed. If the opening mode was *"w"*, then right after opening the pointer points to the beginning of file. If the opening mode was *"a"*, right after opening the pointer points to the first byte after the end of file. After each writing the system shifts the pointer to the byte following the last written byte.

If the opening mode was *"w"*, you may select the location to where the first written byte will be placed or in other words, you may shift the pointer before writing:

fseek(<pointer_to_FILE_struct>, <offset>, <origin>);

Origin is specified by constants defined in file *stdio.h*. They are *SEEK_CUR* (current position), *SEEK_END* (end of file) and *SEEK_SET* (beginning of file). Offset specifies the number of bytes from the origin. Examples:

```
fseek(pFile, 10, SEEK_SET); // put the pointer on the 10-th byte of file
fseek(pFile, -sizeof(struct Date), SEEK_END); // shift the pointer back to overwrite the last
                                              // struct Date
```

If the opening mode was *"a"*, the new data is always appended. Shifting with *fseek* is ignored.

Files (6)

To read from a file use function *fread*:

<number of_read_items> = fread(<pointer_to_buffer_for_read_data>, <size_of_data_item>, <number_of_items_to_write>, <pointer_to_FILE_struct>);

Example:

```
char *pData = (char *)malloc(100);
FILE *pFile = fopen("c:\\temp\\data.txt", "wt+");
if (pFile)
{
     int n = fread(pData, 1, 100, pFile); // 100 characters, one byte each
     if (n != 100)
    { // it may be not a failure, simply there was no data
        printf("Only %d bytes were read\n", n);
    }
    fclose(pFile);
}
```

In case of text files the carriage return – line feed pairs (*"\r\n"*) at the row ends are replaced by line feeds.

Use *fseek* to specify the location of the first byte to read. It is possible in each mode, even in case of *"a+"*. After reading the file pointer is shifted to the first not read byte.