Lecture 5

Structures
Dynamic data structures
Files, project compilation

Institute of Computing Science
Faculty of Computing and Telecommunication

Marcin Radom, Ph.D.

Structures

Theory and examples

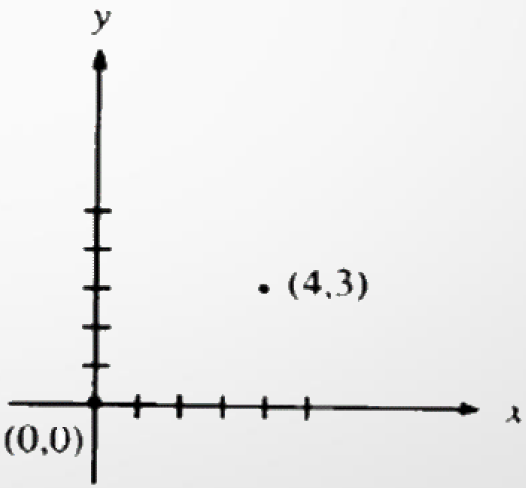
Structures

- **Structure** is a type which collects one or many variables (which are then called fields) under a common name (they are old „records” in Pascal language).
- Using structures allows easier organization of data which are then gathered as a single entity.
- <u>Example:</u> different variables describing a person in a company, like name, position, time of employment, salary, etc.
- Structure is therefore <u>a set of fields</u> where a field can be <u>single variable or an object of another structure</u>.
- What one can do with a structure object:
  - assign one to the other,
  - copy one into other,
  - return as a result of a function,
  - send them to a function.
- **Structure definition:**

```
struct type_of_structure { list_of_fields }
        list_of_identifiers ;
```

Structures

- Declaration of a new structure without specifying the identifiers (names for structure objects) will not reserve any memory.



- **Example:** point coordinates:

```
struct point {      // does not reserve memory
     int x;
     int y;
};
struct point pt; // pt variable definition
point pt; // only in C++ (i.e., we don’t have to use struct keyword here)
```

Structures

- Structure can be initiated using curly brackets syntax as in the example:

```c
struct point maxpt = {1920, 1200};
```

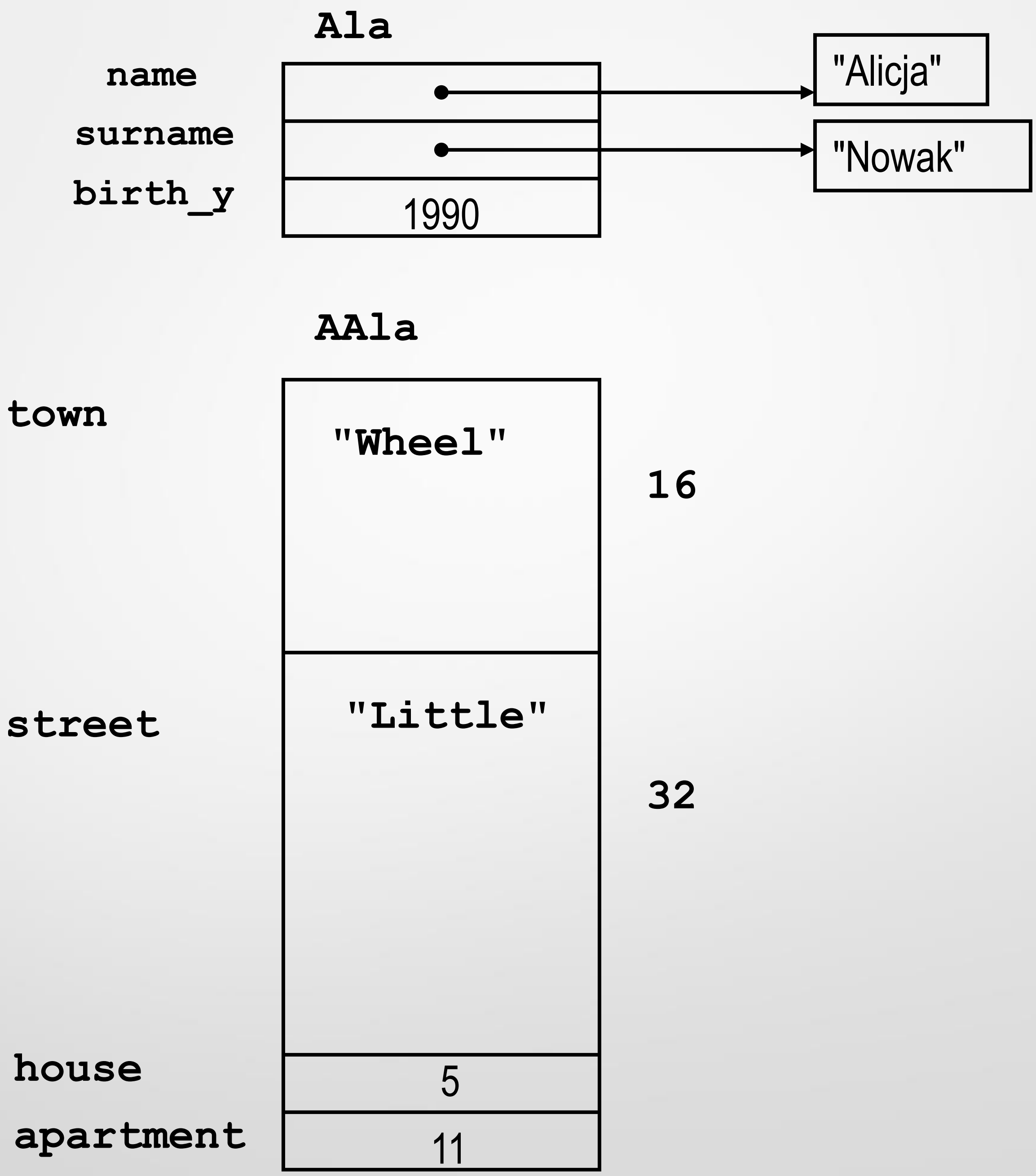
- In a direct initialization (as seen above) values are assigned to the specific fields of a structure in the order taken from brackets and the order of fields in a structure (i.e., if in **point** field **x** is defined before **y**, then x = 1920, y=1200).
- **Example:**

```c
struct{
   int width;
   int accuracy;
   char conversion;
}  pattern;

struct{
   float re;
   float im;
}  z0, z1, z2;
```

Structures

- **Examples:**

```c
struct person {
   char *name;
   char *surname;
   int birth_year;
};
struct person Ala, Ola, Ela;

struct address {
   char town[16];
   char street[32];
   int house;
   int apartment;
};
struct address AAla, AOla, AEla;
```


Ala
name
surname
birth_y
1990
"Alicja"
"Nowak"
AAla
town
"Wheel"
16
street
"Little"
32
house
5
apartment
11

Structures

- When we have a structure like this:

```
struct point {
 int x;
 int y;
}pt, pt1;
```

- then calling to a specific field of it can be done using the following syntax:

`structure_name.field name`

for example:

```
pt.x = 320;
pt.y = 200;
printf("%d \t %d", pt.x, pt.y);
```

- Operator **. (dot)** joins the structure object with its field.
- E.g.: calculating distance between two points
  - function: **`double sqrt(double)`** – returns square root from its argument:

```
double result = 0.0;
result = sqrt(pow((double)pt.x - pt1.x, 2) + pow((double)pt.y - pt1.y, 2));
```

Nested structures

- **Examples**:

```
struct engine {
   float power;
   char *fuel;
};

struct car {
  struct {
     char *producer;
     char *model;
  } mark;
  struct engine V8;
};
```

Nested structures

- How to define rectangle easy? For example by giving its two opposite coordinates (in a diagonal):

```
struct rect {
     struct point p1;
     struct point p2;
}
```

- <u>Calling:</u>

```
struct rect my_rectangle;
my_rectangle.p1.x = 10;    // refers to x coordinate
my_rectangle.p1.y = 20;    // of p1, where p1 is a field of
                           // rectangle
```

Structures – allowed operation

- **Allowed operations:**
  - Assigning whole structure to the other one.
  - Copying one structure into another.
  - Obtaining structure object address with & operator.
  - Calling a structure specific field.
  - Structure can have its fields initialized directly or using { }.

- **Structure cannot be directly compared**, i.e., a new function is required which compares all fields separately – meaning, we have to write it ourselves.

Assigning structures

```
struct signature {
   char *Name;
   char Initials;
   char *Surname;
};
struct signature AK = {"Ann", 'M', "Kowalska"}, NN, st;
st.Name  =  "Andrew" ; // string constant – cannot be modified!
NN  =  AK ;

// if types were equal:
//    NN . Name =  AK . Name ;
//    NN . Initial =  AK . Initial ;
//    NN . Surname=  AK . Surname;
//  AK.Name and NN.Name point to the same text
//  AK.Surname and NN.Surname point to the same text

char BeatifulName[20] = { "Teofil" };
AK.Name = BeatifulName;     // AK.Name points to beginning of array BeatifulName
NN = AK;                    // now NN.Name points the same as AK.Name
AK.Name[0] = 'M';           // modification will be visible in array
                            // BeatifulName both in AK.Name and NN.Name
printf("%s, %s", AK.Name, NN.Name);  // Meofil, Meofil
```

Assigning structures

- **Example**:

```
int T1[3], T2[3] = { 1, 2, 3 };
T1 = T2;  // error, T1 is not a pointer variable so it cannot be used
          // to modify its destination-address. Copying the
          // elements will also not work

struct TT{
     int T[3];
};

struct TT T1, T2 = { { 1, 2, 3} };
T1 = T2;       // or: copying arrays which are fields
               // of the structures T1 and T2
               // but:
int a = T1.T[1];
```

Arrays of structures

Theory and example

Arrays of structures

- Lets assume we have a structure with a sequence of characters and type **int** variable:

```
struct key {
    char *word;
    int count;
}
```

- An array of such a structure (i.e., an array which elements are objects of this structure type) can be declared as follows:

```
struct key keytab[50];
```

- so the syntax is:

```
struct structure_name array_name [ size ]
```

Arrays of structures

- **Example**:

```
struct library {
   char  name [ 16 ] ;
   int  number ;
}  computers [ 100 ] ;       // reserves memory for the array
struct library printers [ 100 ] ; // reserves memory for the array

int  how_k =  0,  how_d =  0;
for ( int  i  =  0 ; i  < 100 ;  ++i ) // sums up
{
     how_k += computers[ i ].number ;
     how_d += printers[ i ].number ;
}
```

Arrays of structures: Initialization

- An array can be initialized using bracket syntax:

```
int year[] = {31, 28, 31, …, 30, 31};
```

- An array of structures can be similarly initialized:

```
struct key {
    char *word;
    int count;
} keytab[] = {
    "auto",0,
    "break",0,
    /* … */
    "while",0
};
```

  - Initial values must be given in pairs, in order of their fields appearance within the structure type.

Arrays of structures: Initialization

- To be more precise, the pairs can also be separeted using brackets:

```
struct key {
    char *word;
    int count;
} keytab[] = {   // the number of elements will be
                 // calculated automatically
    {"auto",0},
    {"break",0},
    /* … */
    {"while",0}
};
```

- Internal brackets can be omitted when we give all values for all fields, and when the fields are simple types.

Arrays of structures: Example

```
const int MAX = 100
struct TV{
     char Mark[32];
     int Price;
     int Number;
};

int main(int argc, char* argv[]){
     struct TV TabTel[MAX];
     int how_many = 0, which;
     bool go_on= true;
     char option;

     while(go_on){
          printf("Choose option [N, W, U, S, Q] : ");
          fflush(stdin);
          scanf("%c", &option);
```

Arrays of structures: Example

```
switch(option & 0x5F){
// add new TV, variable how_many stores its number, must be
// less than MAX
case 'N’:
     if (how_many < MAX){
          printf("Enter a name : ");
          scanf("%31s", TabTel[how_many].Mark); // mark is a table
          printf("Enter price: ");
          scanf("%d", &TabTel[how_many].Price);
          printf("How many?: ");
          scanf("%d", &TabTel[how_many++].Number);
     } else
          printf("Table is full.\n");
     break;
     // show all TVs
case 'W’:
     for(int i = 0; i < how_many ; ++i)
          printf("%d. TV: %s, Price : %d, How many: %d\n", i, TabTel[i].Mark,
           TabTel[i].Price, TabTel[i].Number);
     break;
```

Arrays of structures: Example

```
        // remove TV with a given index:
        case 'U':  printf("Provide index of TV: ");
             scanf("%d", &which);
             if (which >= 0 && which < how_many){
             // in place of the removed one put the last one:
                  TabTel[which] = TabTel[--how_many];
                  printf("Removed.\n");
             } else printf("Wrong ID.\n");
             break;
        // compute sum of prices:
        case 'S’:
             which = 0;
             for (int i = 0; i < how_many; ++i)
                  which += TabTel[i].Price * TabTel[i].Number;
             printf("Total value: %d\n", which );
             break;
        // quit
        case 'Q’:
             go_on = false;
             break;
        default:  printf("Wrong option.\n");
      }

   }
   return 0;
}
```

Pointers to structures

- **Example:**

```
struct key {
      char *word;
      int count;
}
struct key *p; // pointer for key structure, initially points to nothing
struct key table [] = { <elements_of_table> };
p = & table [0];    // points on the first element
//
struct AZ {
   char z1;
   char z2;
   int lz;
} p1 = { 'a', 'b', 37 },
  p2 = { 'x', 'y', 35 };
struct AZ p3 = { 'k', 'l', 36 };
```

Pointers to structures

- **Example:**

```
struct S1 {
   int i;
   float f;
} es1;

struct S2 {
   int i;
   long l;
} es2;

struct S1 *wst1;
struct S2 *wst2;

wst1 = &es1;
wst2 = &es2;
wst1 = &es2;  // error, wrong (structure) type
wst2 = wst1;  // error, wrong (pointer) type
```

Structures – access to fields

- Access to structure fields:
  - **For the identifier .   (dot)**

    `name_of_structure.name_of_field`
  - **For the reference . (dot)**

    `reference.name_of_field`
  - **For the pointer   `->`   `(~arrow)`**

    `pointer->name_of_field`
- **Example:**

```
struct signature {
   char *Name;
   char Initial;
   char *Surname;
};
struct signature st, *wst = &st ;
st . Name =  "Andrew" ;       // st is identifier
wst -> Initial =  'K' ;       // wst is a pointer to the structure
```

Dynamic data structures

Pointers and lists

Pointers to structures

- Pointer to structures are still pointers, like for the normal variables:

```
struct point {
      int x;
      int y;
}pt;
struct point *pp = &pt;
```

  - ***pp*** is a pointer to the structure of a type: **`struct point`**
  - if ***pp*** points to structure *point*, then ****pp*** refers to this structure object, while ***(*pp).x*** and ***(*pp).y*** are its fields.
  - Parenthesis are necessary, because operator . (dot) has a higher priority than indirect addressing operator *
  - So: a statement ****pp.x*** means (for the compiler) exactly the same as ****(pp.x)***, and it is WRONG because *x* is not a pointer!!!

Pointers to structures

```c
struct rect{     // definition of a rectangle
      struct point pt1;
      struct point pt2;
}
struct rect r, *rp = &r;
```

- Operators **.** and **->** are left-side joined, so the following statement are equivalent:

```c
r.pt1.x
rp->pt1.x  // pt1.x, because pt1 is NOT a pointer
(r.pt1).x
(rp->pt1).x
```

- <u>Four operators:</u>
  - reference to the field of a structure: **.** (dot)
  - operator: **->**
  - operator: **()** - calling a function
  - operator: **[]** - array indexer

  have THE HIGHEST priority, so they are „computed" before any other.

Pointers to structures

- **Example:**

```
struct {
     int len;
     char *str;
} *p;
++p->len      // increases variable len, because ++ has higher
              // priority than ->
```

- <u>On the same principle, the following statements:</u>
  - **`(++p)->len`** increases *p* before calling field *len*
  - **`p++->len`** increases *p* after calling *len*
  - **`*p->str`** provides something on which *ptr* points
  - **`*p->str++`** z increases *str*, after providing that for *str* points to (the same as **`*s++`**)
  - **`(*p->str)++`** increases something what *str* points to
  - **`*p++->str`** increases *p*, after providing something, which *str* points to

- **Summary: it is far better to use parenthesis (). We minimize the probability of a mistake because of not remembering exactly what priorities differences are between the different operators.**

Structures – dynamic memory assignment

- **Example:**

```
struct dog{
   char *leash;
   char *collar;
   char *the_dog_itself;
};

struct dog *Morus;
Morus = new struct dog;     // allocates memory for a structure
                 // its fields (pointers) at the moment point at nothing

      // all field must have memory allocated separately:
Morus->leash = new char[20];
delete [] Morus->leash ;    // allocated memory must be released separately
                 // before removing the structure object itself
delete Morus;    // releases memory of the structure object
```

Structures – dynamic memory assignment example

```
// Program – registry of bicycles
const int MAX = 100;
struct Bicycle{
     char *Mark;
     int Price;
     int Number;
};

int main(int argc, char* argv[]){
     struct Rower* TabRow[MAX]; // array of bicycles
     int how_many = 0, which_one;
     bool go_on = true;
     char option ;
     char Bufor[64];
     while(go_on){
          printf("Choose an option [N, W, U, S, Q] : ");
          fflush(stdin);
          scanf("%c", &option);
```

Structures – dynamic memory assignment example

```
switch(option & 0x5F) {
     // add new bicycle
     case 'N':
          if (how_many < MAX) { // allocates memory for a new structure:
               TabRow[how_many] = new struct Bicycle;
               printf("Provide name: ");
               scanf("%63s", Bufor);
               // allocate memory for the bytes provided by user:
               TabRow[how_many]->Mark = new char[strlen(Bufor) + 1];
               // copy name from buffer to the field:
               strcpy(TabRow[how_many]->Mark, Bufor);
               printf("Enter price: ");
               scanf("%d", &TabRow[how_many]->Price);
               printf("Enter number of objects: ");
               scanf("%d", &TabRow[how_many++]->Number);
          } else
               printf("Table is full. n");
          break;
```

Structures – dynamic memory assignment example

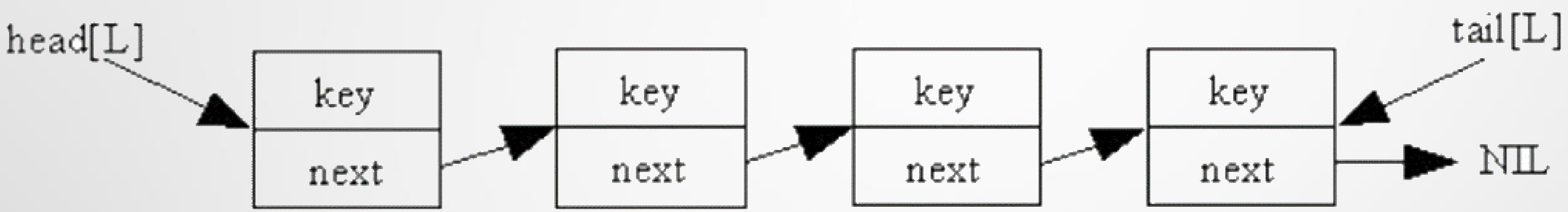
```
// show all bicycles:
case 'W':
     for(int i = 0; i < how_many; ++i)
          printf("%d. Bicycle: %s, Price: %d, How many: %d\n", i,
                TabRow[i]->Mark, TabRow[i]->Price, TabRow[i]->Number);
     break;
// remove bicycle object by an index:
case 'U':
     printf("Provide index: ");
     scanf("%d", &which_one);
     if (which_one >= 0 && which_one < how_many){
          // first remove the fields, then the structure:
          delete [ ] TabRow[which_one]->Mark;
          delete TabRow[which_one]; // removes the structure
          // move the last one in place of the removed one:
          TabRow[which_one] = TabRow[--how_many]; //copy address
          printf("Removed.\n");
     } else
          printf("Wrong ID.\n");
break;
```

Structures – dynamic memory assignment example

```
        // sum of values:
        case 'S’:
             which_one = 0;
             for (int i = 0; i < how_many ; ++i)
                  which_one += TabRow[i]->Price* TabRow[i]->Number;
             printf("Total value: %d\n", which_one);
             break;
        // quit
        case 'Q’:
             go_on = false;
             break;
        default:
             printf("Wrong option.\n");
        }
    }
}
```

List

- **One-directional list** – all the structures have one pointer to the next one in line



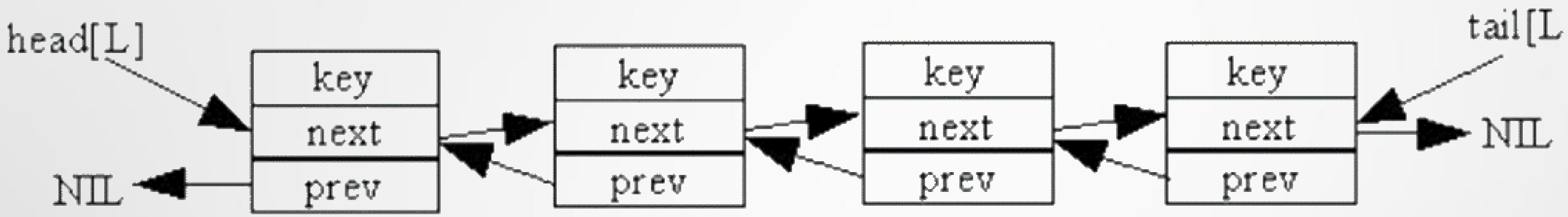
- <u>Single node:</u>

```
struct lnode{
    int number;
    struct lnode *next;
}
```

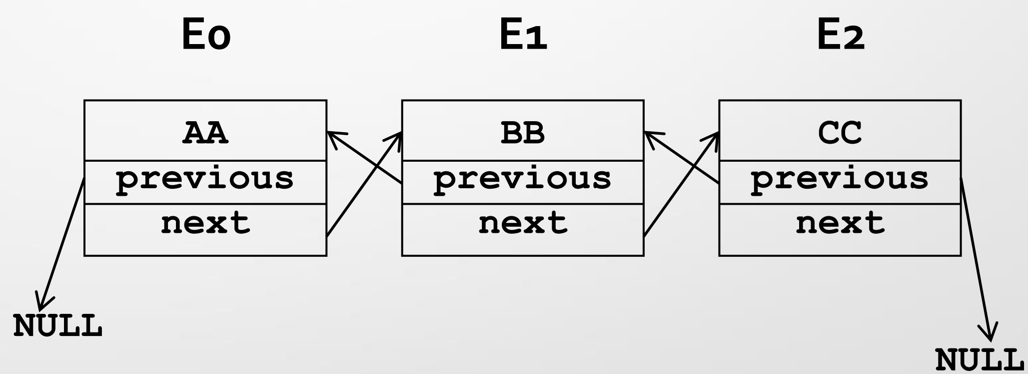
  - Pointer *next* points to the same types of object as the structure it is in.

List

- **Two-directional list** – each structure have two pointes: next and previous



- **Example:**




```
struct list{
      char etykieta[8];
      struct list *previous;
      struct list *next;
} E0 = { "AA", NULL, NULL };
struct list E1 = { "BB" },
      E2 = { "CC", NULL, NULL };
E0.next = &E1; // E0 has pointer pointing to the next element: E1
E1.next = &E2; // E1 has pointer pointing to the next element: E2
E2.previous = &E1; // E2 has pointer pointing to the previous element: E1
E1.previous = &E0; // E1 E2 has pointer pointing to the previous element: E0
```

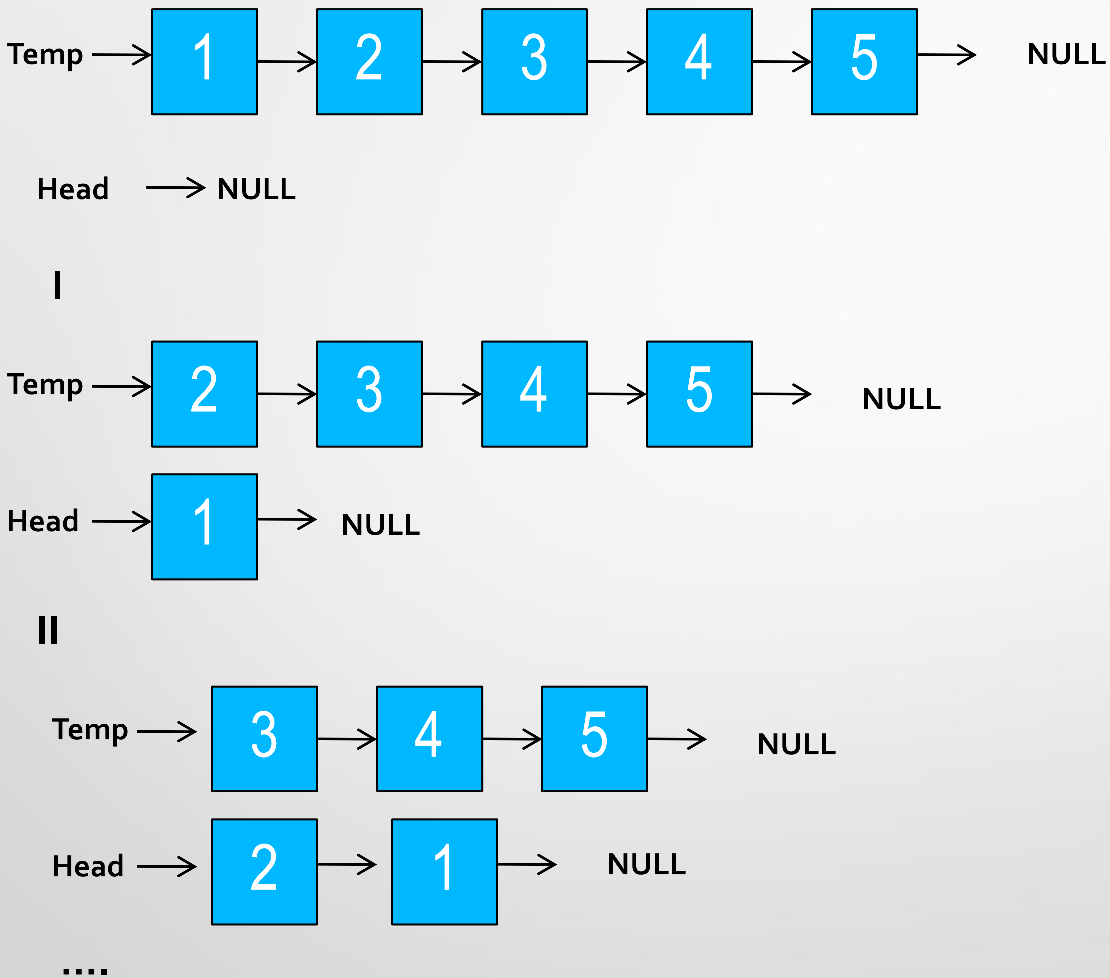
List with one-way pointers

```
struct Elem {             // single node declaration
     Elem *Next;          // pointer to the next node
     int Value;
};
struct Elem *Head;        // pointer to type Elem structures

int main(){
     //------------- Create list --------------
     Elem e1 = {NULL,  10};
     Elem e2 = {NULL,  20};
     Elem e3 = {NULL,  30};
     Elem e4 = {NULL,  40};
     Elem e5 = {NULL,  50};

     e1.Next = &e2;
     e2.Next = &e3;
     e3.Next = &e4;
     e4.Next = &e5;
     Head = &e1;          // first pointer of the list: e1 (i.e., Head of the list)
```

List with one-way pointers - Example

```c
//------------ Temporary variables ---------------
Elem *Temp, *Aux;

//------------ Writes down all the nodes: --------------
printf("\nAfter reverse:\n\n");
Temp = Head;
while (Temp != NULL){ // until the last elements
     printf("%3d", Temp->Value); // writes down value of a node
     Temp = Temp->Next; // go to the next node
}

//---------------- Reversing the list --------------------
Temp = Head; // current
Head = NULL; // head of reversed list
while (Temp != NULL){
     Aux = Temp->Next; // tail of the initial list (being reversed)
     Temp->Next = Head; //new head for partially reversed list
// first element of tail of the initial list becomes head of the reversed list
     Head = Temp;
     Temp = Aux; // shorted list (by head)
}
```

List with one-way pointers – Example



List with one-way pointers - Example

```c
//---------------- Writes down the list --------------------
 printf("\n\nAfter reversing:\n\n");

 Temp = Head;
 while (Temp != NULL){
      printf("%3d", Temp->Value);
      Temp = Temp->Next;
 }

 printf("\n\n");
 return 0;
}
```

Union

- Union is a variable while its syntax is similar to the syntax of a structure.
- However, an union can, at a given moment, **store only one value for one of its fields**.
- In different moments the stored field can change, but it can be only one stored at a time.
- **All union fields are stored in the same area of memory.**
- **Example:**

```
union u_tag {
     int ival;
     float fval;
     char *sval;
} u;
```

  - Variable *u* will have enough bytes to store the largest of its fields.
  - Variable *u* can be assigned a value of any of its fields..
  - At any given moment variable *u* can store either `int`, `float` or a pointer to the characters sequence. Only one of them at a time!

Union

- The programmer must know which field of a union is used at a given time (because if we try to access a field which is not stored at the moment, results are dependent on the program implementation (compiler, etc.)).
- **Accessing fields of a union:**

  `union_name.field`

  or

  `pointer_to_union->field`
- **Example:**

```
enum types {INT, FLOAT, STRING};
types utype;
/* … */
if (utype == INT) printf("%d\n", u.ival);
else if (utype == FLOAT) printf("%f\n", u.fval);
else if (utype == STRING) printf("%s\n", u.sval);
else printf("bad type %d in utype\n", utype);
```

Union

- **Examples:**

```
union
{
   char p1 ;        //  1 byte
   int  p2 ;        //  4 bytes
   long long p3 ;   //  8 bytes
} u1;               //  total size for union: 8 bytes

u1.p1 = 'A';         // fields p2, p3 are unidentified
u1.p2 = 1357;        // fields p1, p3 are unidentified
u1.p3 = 15432678LL; // fields p1, p2 are unidentified

union something
{
     char color_1 [ 64 ] ;
     char color_2 [ 64 ] ;
}
```

Union inside a structure

- Unions can be used in arrays or as structure fields.
- Example:

```
struct {
     char *name;
     int flags;
     int utype;
     union {
          int ival;
          float fval;
          char *sval;
     } u;
} symtab[NSYM]; // table of structures
```

  - for field *ival* we call:

```
symtab[i].u.ival
```

  - for the first sign of *sval* we can used two syntaxes:

```
// not: symtab[i].u.*sval !!!! – error, no variable: sval
*symtab[i].u.sval // dot has higher priority than *
symtab[i].u.sval[0]
```

Nesting unions and structures

- **Example:**

```
struct paper{
    char author[64];
    char title[64];
    int year;
    union {
        struct {
            char title[64];
            int volume;
            int page;
        } journal;
        struct {
            char name[64];
        } conference;
    } place;
} a1;
// string.h
strcpy ( a1.place.conference.name, "Polman" );
```

Multi files projects

Compiler joining rules, etc.

Compiler files joining theory

- **Separating code into different files allows:**
  - <u>Making the code which is more readable</u>.
  - <u>It is easier to use different modules in other programs.</u>
  - <u>Compiling different modules separately.</u>
- **Module** – a fragment of a program compiled as a separate file.
- **Each module consists of:**
  - <u>Interface</u> (information what a given module has) in a form of a header file (*.h or .hpp)
  - <u>Implementation</u> – file: *c. or *.cpp. At the beginning of it we should include its header file (e.g., for *module.c* it will be *module.h*).

Compiler files joining theory

- **Typical header file consists of:**
  - Explicit constants (e.g. **EOF**, **NULL** in *stdio.h*)
  - Macro-functions: e.g. *getchar()* is by default alias for *getc(stdin)* or functions from a library *ctype.h*
  - Functions declarations e.g. file *string.h* contains declarations (prototypes) of functions working on strings.
  - Definitions of patterns, structures, e.g., standard input/output functions (a pattern of I/O structure is in *stdio.h*).
  - Types definitions, e.g., type **FILE** is a pointer to the structure defined in *stdio.h* (with help of *typedef* or directive *#define*). Also types like *size_t* or *time_t* are in such header files.
- Each and every header file should be made in such a way that it can be included many times within a program into any module.

Compiler files joining theory

calc.h

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char []);
int getch(void);
void ungetch(int);
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    ...
}
```

getop.c

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop() {
    ...
}
```

stack.c

```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

getch.c

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
    ...
}
void ungetch(int) {
    ...
}
```

Compiler files joining theory

- Most programmers make their own header files and use them in their programs. Some of them are special-purpose files, while the others are for general common tasks.
- <u>Included files can have their own ***#include*** instructions.</u>
- Header files can have declarations of external variables to be shared in different files. It is however cumbersome, because one has to check if a given variable has been already declared or not.
- Header files usually contain variables with: ***const static***.
  - ***const*** qualifier protects from modifications, while ***static*** makes every included header file have its own copy of a variable.
  - In such a way one can define a variable in only one file and use it in other ones with keyword ***extern***.

Compiler files joining theory

- In a single file we can include another file with its own code.
- Program divided into different files will be merged into one „file" by compiler. For the compiler it will be one **module**, often called **translation unit**.
- It means that changing something in one file will require another compilation of the **whole program**, not one translation unit.
- In general a program can be divided into different translation units, each one compiled separately.
- Translation units can and should divide a program into some „logical" units from the point of view of task / problems that program solves.
- Each translation unit is compiled separately.

Compiler files joining theory

- <u>In C types of variables and calling of a function is always checked</u>. Each function called in a given translation unit must be declared in that unit.
- <u>Function definition</u> should be done once and placed into a single translation unit. All declarations and definitions must be compatible.
- After being joined by the **linker**, all functions „see each other" without problems.
- <u>Names of global functions</u> are in a „common" area of the global modules (they are exported).
- <u>Global variables are not shared</u>. Global variable **x** from one translation unit is accessible only in that unit; the other unit can defined global variable with the same name for its own use – there will be then two separate global variables, each visible in its own module.
- If we want to export global variable, it must be defined in one unit and in the other ones we have to use order **extern**, e.g.:

```
extern double x;
```

Example 1

- **Example:** Program separated into different files, while being a single translation unit for the compiler

```
// code in file Part1.cpp
double Recip(double x){
    if(x != 0)
        return 1.0/x;
    else
        return 0;
}
// code in file Part2.cpp
int Power(int n) {
    if (n < 0 || n > 12)
        return 0;
    if (n == 0)
        return 1;
    else
        return n * Power(n-1);
}
```

Example 1

```
// code in file main.cpp
#include <stdio.h>
#include "Part1.cpp"
#include "Part2.cpp"

int main(int argc, char* argv[])
{
      int k;
      printf("Integer number: ");
      scanf("%d",&k);
      printf("\n%lf\t%d\n\n", Recip(k), Power(k));

      return 0;
}
```

Example 2

- **Example:** Program in different files, and for the compiler it will be divided into different modules (translation units).

```cpp
//code in header file Part1.h
double Recip(double x);

// code in file Part1.cpp
#include "Part1.h"

double Recip(double x)
{
	if(x != 0)
		return 1.0/x;
	else
		return 0;
}
```

Example 2

```
// code in header file Part2.h
int Power(int n);

// code in file Part2.cpp
#include "Part2.h"
int Power(int n)
{
	if (n < 0 || n > 12)
		return 0;
	if (n == 0)
		return 1;
	else
		return n * Power(n-1);
}
```

Example 2

```
// code in file main.cpp
#include <stdio.h>
#include "Part1.h"
#include "Part2.h"

int main(int argc, char* argv[])
{
	int k;
	printf("Integer number: ");
	scanf("%d", &k);
	printf("\n%lf\t%d\n\n", Recip(k), Power(k));
	return 0;
}
```

Scope of variables

Local and global variables and range

Variable declaration scope

- **Variables scope** is an important issue, since we need to know if a given variable is valid (accessible) in a given code fragment or not.
- By **declaration scope** we understand such fragments of a code in which a given declaration is accessible.
- By **visibility scope of the declared variable** we understand an area of the code in which identifier can be used (i.e., the name of identifier is associated with the declaration). Narrow range can **shadow** declaration of another, broad-scope variables.
- **There are two main scopes for variables:**
  - global scope/range: for the whole program
  - local scope/range: definition of a single function

Variable declaration scope

- **Local** variables (privates, automatic variables) – they are declared inside a function.
  - Each local variable of a function starts to exist (in the memory) when a function is called, they end when the function ends.
  - Such variable cannot store its older values for the next calling of a function.
- <u>Variables declared inside a code block</u> (their scope is in range of { } brackets) – should be visible from the line where their declaration is to the end of the code block (e.g., control variable of a loop: ***for*** declared within () parenthesis of such a loop).
- **Global** variables (external) – they are visible (and accessible) for any function of a program in a single file.
  - **Global** variable must be defined once, outside of any function.
  - Global variable is visible starting from its declaration to the end of file.
  - It must be declared with ***extern*** if it should be visible in different files.

Variable declaration scope - Example

- **Example**

```
int i, j, k;
float X, Y;
int F1(int a, int b){
      char c1, c2;
      float B;
}

int F2(float Z1, float Z2, char cp) {
      int A1[15];
      long A2[15][15];
      float B1, B2, B3;
}

void main(void){
      int m, n, p, q;
      float V1, V2, V3;
      long T1[15][15], T2[15][15];
}
```

Variable declaration scope - Example

- **<u>Scope for variables in example:</u>**
  - **global :**
    `i, j, k, X, Y, F1, F2 (F1,F2 – functions)`
  - **local in function F1 :**
    `a, b, c1, c2, B`
  - **local in function F2 :**

    `Z1, Z2, cp, A1, A2, B1, B2, B3`
  - **local in function main :**

    `m, n, p, q, V1, V2, V3, T1, T2`

Variable declaration range

```
{   int i;
    ...
    {
        ...
        int j;
        ...
        {
            int n;
            ...
        }
        for(int k=0;... )
        {  ... }
    }
}
```

Variables shadowing

- Global variables can be **shadowed** inside a function (or a code block) by declaring a local variable with the same name (it can be of any type).
- In such a situation the name of the variable in such a block refers to the local variable only. Global one exists but it is not accessible by its name inside such a block.
- **Example:**

```
int i = 5;                          // i == 5
int F1 (int n){
      int i = 7;                    // i == 7, shadowing
      return i + n;
}

int F2 (int m)
{ return i + m; }                   // i == 5

int main(void){
      int k, z, i = 0;              // i == 0
      k = F1(0);                    // k == 7
      z = F2(0);               // z == 5
      return 0;
}
```

Same identifier in range - error

- **Example:**

```
float eps = 0.001;
...
double eps = 0.05;      // error – global variable range redefinition
...
void main(void)
{
   long k1;
   ...
   int k1;              // error – local variable redefinition in same scope
   ...
}
```

Accessing global variables

- In order to access a shadowed variables we can use scope operator **::**
- If, for example, variable **x** is shadowed by a local **x**, in order to access global x we need to write **::x**
- **Example:**

```
const int max = 15750;   // global scope
int MAX(int TAB[ ], int size )
{
   int max = TAB[0];     // local
   for (int i = 1 ; i < size ; ++i)
      if (::max > TAB[i] && max < TAB[i] ) //global ( :: ) & local
         max = TAB[i];   // local
   return max;           // local
}
```

Static variables – global scope

- **Static variable types:**
  - <u>Global scope</u>
  - <u>Local with **static**</u>
- Lets assume that our program has two modules and for example in *stack.h* there are defined two variables *sp* and *val*:

```
int sp;

double val[MAXVAL];
```

- For such variables we can access them from another module using keyword **extern** (so it gives us access to global variables from different files).
- If for some reason we do not want to allow such a scenario (i.e., to make some variables global only in one file) we have to use keyword **static**.
- Declaration with **static** used for a global variables and a function <u>limits their scope from the line of their declaration to the end of file they are in</u>.
- Declaring external object as **static** hides their name (in different files).

Static variables – global scope

- External declarations with **static** often are used for variables, but it can also be used for functions.
- Names of the functions are global, accessible in all code.
- If a function is declared as **static**, then it is accessible only within the file where it is declared.
- Example:

```c
static char buff[SIZE];
static int index;
static void count(double a):
int other(int a, int b);
```

  - Variables *index* and *buff*, also the function *count* are static– they are not accessible from other files.
  - Static function can only be called within the file where it is declared.
  - Such names (variables and functions) can be used for other variables in different files.

Static variables – local range

- **static** declaration can be used for local variables.
- Internal static variables are local for their function. However, the value they store are kept between different calling to their functions (they act like a memory of a function).
- **Example:**

```
int Counter(void){
   static int ct = 0;
   return ++ct;
}
int number;
number = Counter();              // number == 1
..................
number = Counter();              // number == 2
..................
number = Counter();              // number == 3
```

Register variables

- **register** variable is an information for the compiler that such variable will be extensible used.
- This suggest to store such a variable directly on a CPU register.
- **Example:**

```
register int i;
register char c;
```

- In practice the compiler / machine can ignore such a „suggestion" and make a variable follow normal rules of memory assignment.
- <u>Register variable cannot be used with & operator (to get their address).</u> It does not matter if such a variable will be stored in register of CPU or not (we don't know it anyway when we write the code, nor knows it the compiler).

Questions?