Maciej Sobieraj

Lecture 7

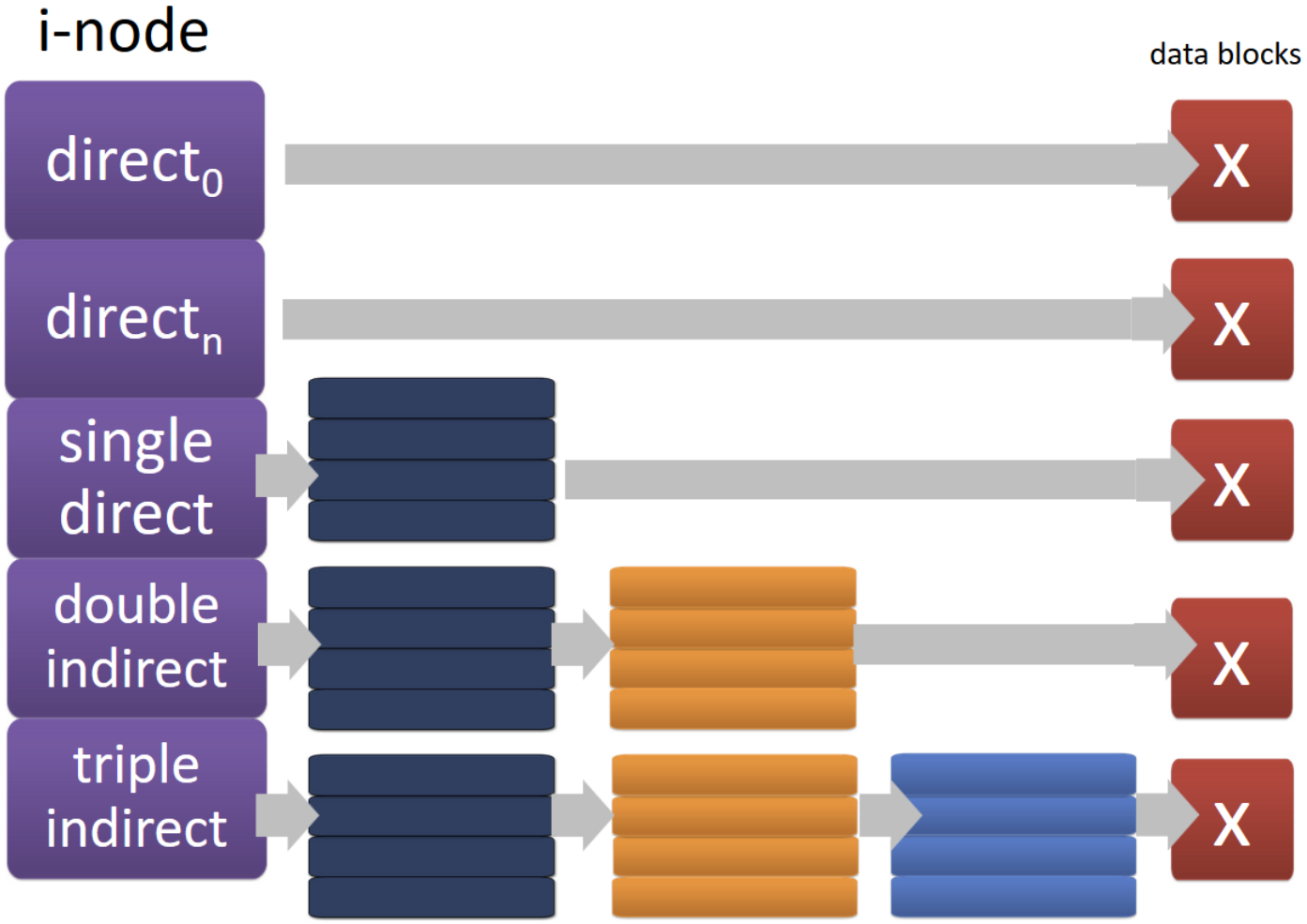
Outline

1. Arrays vs. structures: different aggregates for different purposes
    1. **Arrays of pointers as multidimensional arrays**
    2. Declaring arrays: traps and puzzles
    3. The structures: why?
    4. Declaring and initializing structures
    5. Pointers to structures and arrays of structures
    6. Basics of recursive data collections
2. Quiz

Arrays of pointers

- We only know that the number of columns is stored in the cols variable and the number of rows in the rows variable.
- We’re not going to attempt to allocate the array in the following way:
  - **int** *ptrtab = (**int** *) malloc(rows * cols * sizeof(**int**));
- If we want to access the element in column c and row r, we would have to calculate the pointer to the element as follows:
  - ptrtab + (cols * r) + c

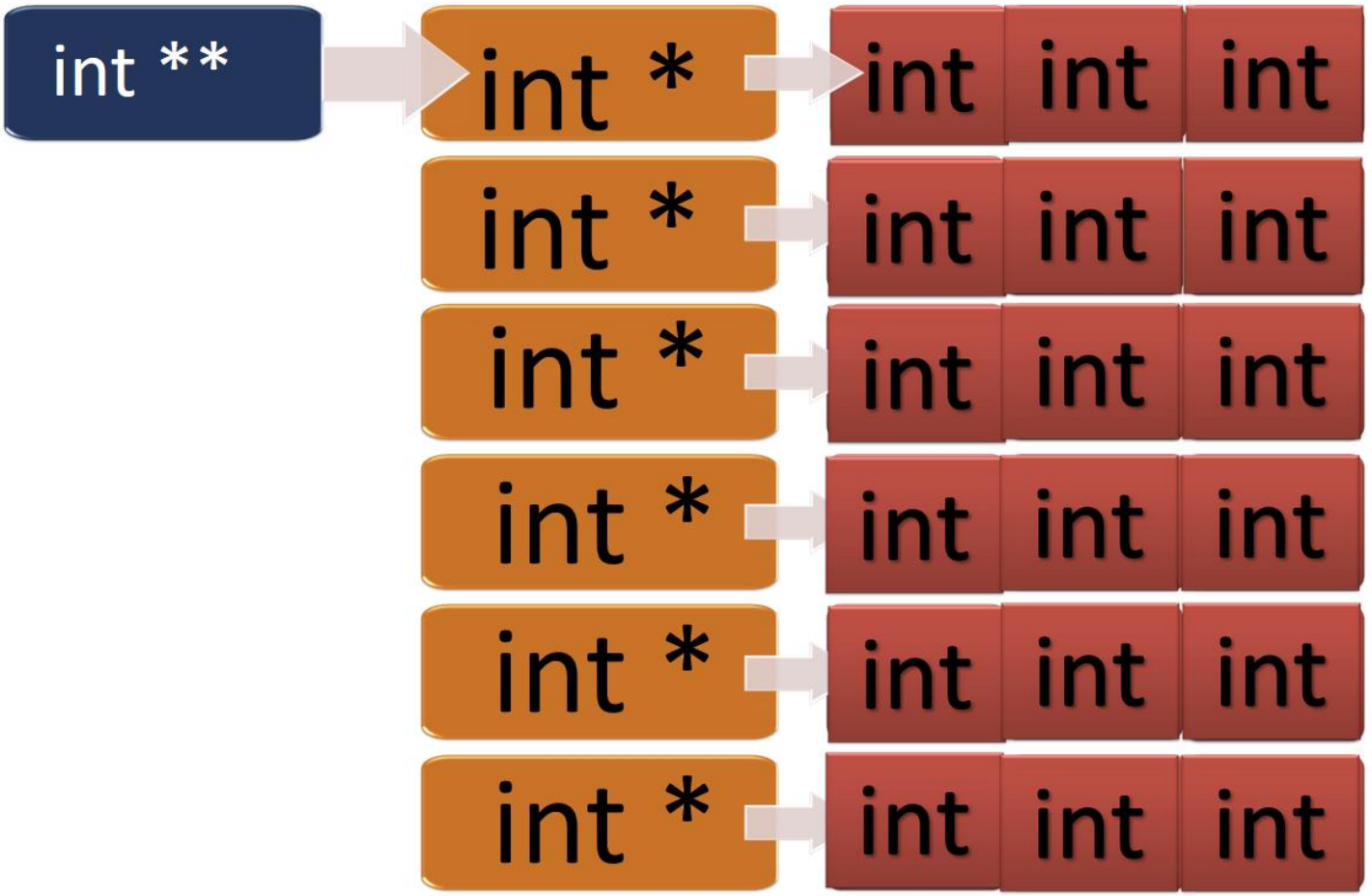
Arrays of pointers



Arrays of pointers

- This is how we do it:
  - to avoid the inconvenience of skipping over the subsequent rows, we'll store the pointer to the beginning of every row so we can reach each row without any acrobatics. How do we store these pointers? In the array, of course! We'll call it the **array of rows**; every row will have as many elements as columns of the desired array;
  - every element in the array of rows will be a **pointer to a separate row**;
  - we need one more pointer to point to the array of rows – we call it *ptrtab*.

Arrays of pointers



Arrays of pointers

- This means that the type of *ptrtab* is "**a pointer to a pointer to *int***", which is denoted as "***int*** **"

```
int **ptrtab;
```

Arrays of pointers

```
ptrtab = (int **) malloc (rows * sizeof (int *));
```

- Firstly, the pointer returned by *malloc surrender* **is converted** to type *int* ** and assigned to *ptrtab*.
- Secondly, the elements of the array of rows are pointers to the rows, so their type is *int* * and hence, the **size of the array is expressed as *sizeof (int *)*** multiplied by the number of rows.

Arrays of pointers

- Finally, we need to allocate memory for every row and store the resulting pointer inside the right element of the array of rows.

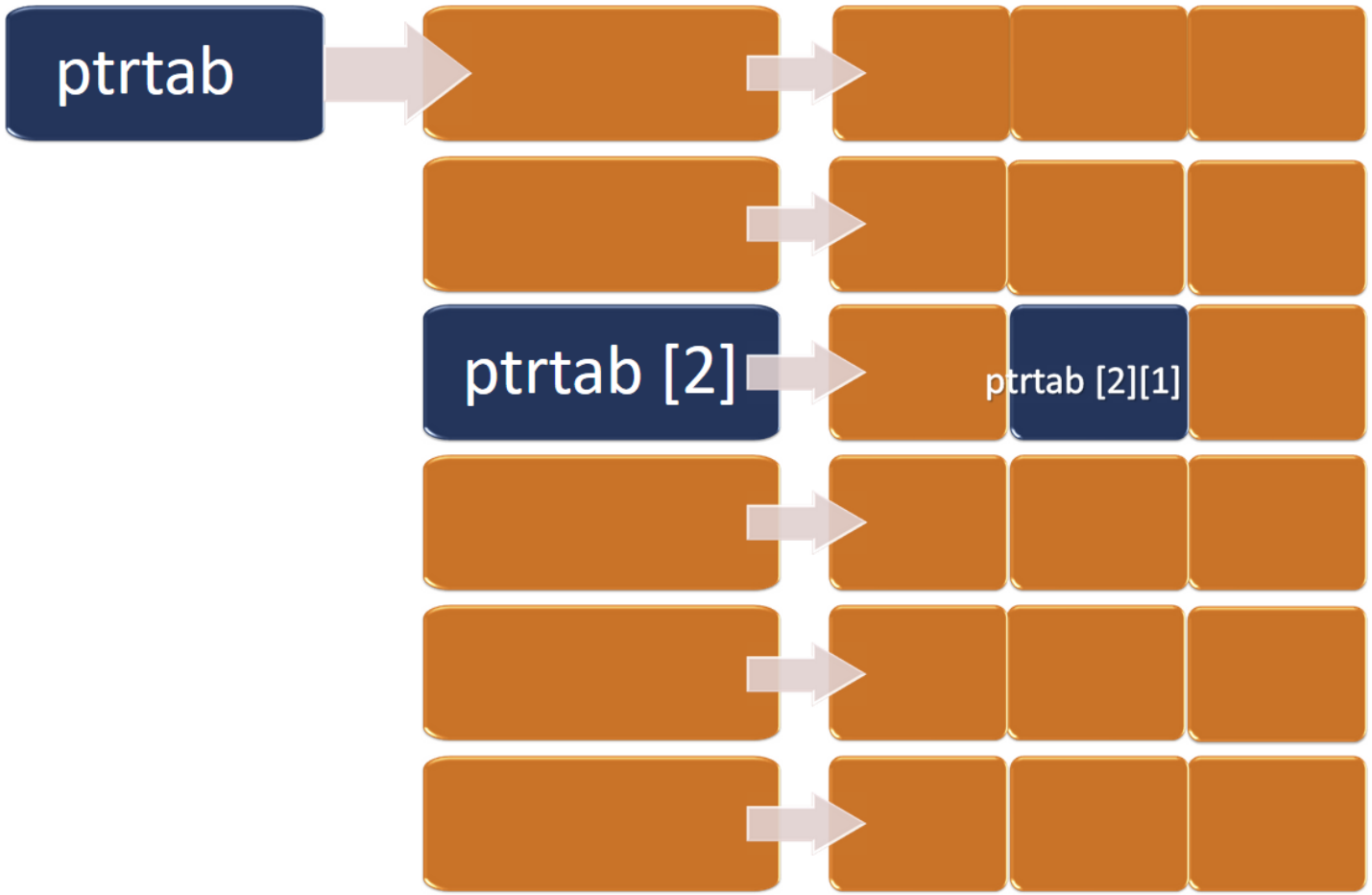
```
for (r = 0; r <rows; r++)
    ptrtab[r] = (int *) malloc (cols * sizeof (int));
```

Arrays of pointers

- For example, if we want to assign 0 to the element lying in row *r*, column *c*, we'll do it this way:

```
ptrtab[r][c] = 0;
```

Arrays of pointers

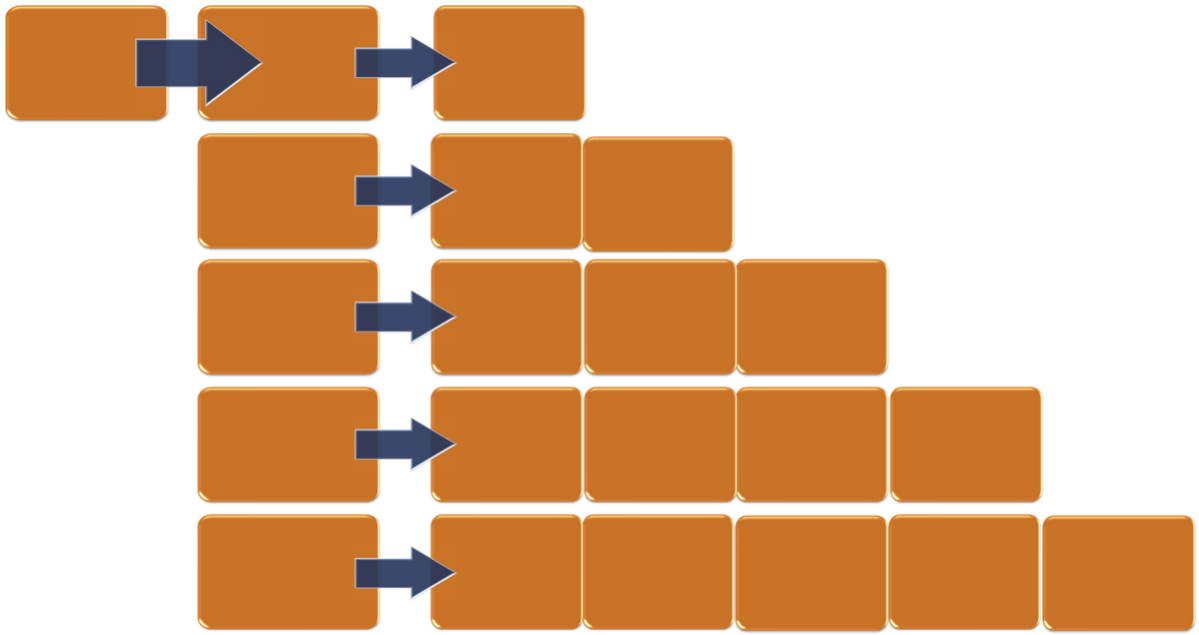


Arrays of pointers

- How does it work?
  - the *ptrtab[r]* expression is interpreted as *(*ptrtab* + *r*), which means the **dereferencing of the element** pointing to the selected row;
  - the pointer is **dereferenced once more** so the entire indexing expression looks as follows:
    - *(*(ptrtab + r) + c)
  - and this is simply the desired value of type *int*.

Arrays of pointers

- The advantage of such arrays is that, unlike ordinary arrays, every row may be of a **different length**.
- It refers specifically to ***triangular matrices***.



Arrays of pointers

- ```
  int rows = 5, r;
  int **ptrab;
  ptrtab = (int **) malloc (rows * sizeof (int *));
  for (r = 0; r <rows; r++)
      ptrtab[r] = (int *) malloc (sizeof (int) * (r + 1));
  ```


  ```

Outline

1. Arrays vs. structures: different aggregates for different purposes
    1. Arrays of pointers as multidimensional arrays
    2. **Declaring arrays: traps and puzzles**
    3. The structures: why?
    4. Declaring and initializing structures
    5. Pointers to structures and arrays of structures
    6. Basics of recursive data collections
2. Quiz

Some declarative traps

- In this way, we've declared a variable *array* which is a 10-element array of pointers to data of type *int*.

```
int *array[10];
```

Some declarative traps

- And now let's look at a seemingly very similar, but completely different, declaration.
- It declares *array* as a pointer to a 10-element array of type *int*.

```
int (*array)[10];
```

Some declarative traps

- The statement creates a variable *array*, which is a **pointer to a 10-element array whose elements are pointers to *ints*.**

```
int *(*array)[10];
```

Outline

1. Arrays vs. structures: different aggregates for different purposes
    1. Arrays of pointers as multidimensional arrays
    2. Declaring arrays: traps and puzzles
    3. **The structures: why?**
    4. Declaring and initializing structures
    5. Pointers to structures and arrays of structures
    6. Basics of recursive data collections
2. Quiz

Structures – why do we need them?

```
char student_name[100000][26];
```

- Let's try to manipulate that array. For example, suppose that the first registered student was Mr. Bond (James Bond).
  - strcpy(student_name[0],"Bond");

Structures – why do we need them?

```
float student_time[100000];
```

- We know that Mr. Bond spent three hours and thirty minutes studying our course. We'll denote it in the following way:
  - student_time[0] = 3.5;

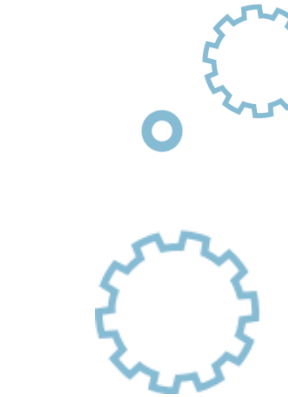
Structures – why do we need them?

- The primary objection is that the data regarding the same object (a student) is **dispersed between three variables**, although it should logically exist as a consolidated unit.
- A structure contains **any number of elements of any type**. Each of these elements is called a **field**.

Structures – why do we need them?

- This is the declaration of the structure:

```
struct STUDENT {
   char  name[26];
   float time;
   int   recent_chapter;
};
```



Structures – why do we need them?

- We should emphasize that the previous declaration **doesn't create a variable**, but only describes the structure we're going to use in our program.
- This declaration sets up a variable (a **structured variable**) named *stdnt*.

```
struct STUDENT stdnt;
```

Structures – why do we need them?

- As the “C” language offers a specialized indexing operator ***[]*** for arrays, we also have the **selection operator**, designed for structures and denoted as a single character . (dot).
- The priority of the selection operator is very high, equal to the priority of the [] operator.

Structures – why do we need them?

- This is a binary operator. Its left argument **must identify the structure** while the right argument must be the **name of the field** known in this structure.

stdnt.time

- Consequently, you can use both of these selectors:
  - stdnt.time = 1.5;
- and
  - **float** t;
  - t = stdnt.time;

Structures – why do we need them?

- Virtually any data could be used as a structure's field: scalars (including pointers), arrays and also almost all of the structures. We say “almost” because a **structure cannot be a field of itself.**
- Structures can be aggregated inside an array, so if we want to declare an array consisting of STUDENT structures:

```
struct STUDENT stdnts[100000];
```

Structures – why do we need them?

- This means that if we want to select the *time* field of the fourth *stdnts*' element, we write it as follows:
  - stdnts[3].time
- We've collected all these assignments which have been performed for the three separate arrays. Analyze them carefully:
  - strcpy(stdnts[0].name, "Bond");
  - stdnts[0].time = 3.5;
  - stdnts[0].recent_chapter = 4;

Declaring the structures

- For the purposes of simplicity, we'll use a simple structure designed to store the date.

```
struct DATE {
    int year;
    int month;
    int day;
};
```

- we can write this declaration much more compactly:
  - **struct** DATE {
    int year,month,day;
    };

Declaring the structures

- The new variable would be declared, for example, in this way:
  - **struct** DATE DateOfBirth;
- We can use it to store Harry Potter's date of birth:
  - DateOfBirth.year = 1980;
  - DateOfBirth.month = 7;
  - DateOfBirth.day = 31;
- We can also use the structure tag to declare array of structures:
  - **struct** DATE visits[100];

Declaring the structures

- We can defining the structure tag and declaring any number of variables simultaneously in the same statement, like this:
  - ```
    struct DATE {
        int year, month, day;
    } DateOfBirth, visits[100];

    struct DATE current_date;
    ```


    ```

Declaring the structures

- We can also omit the tag and declare the variables only:
  - **struct** {
    int year, month, day;
    } the_date_of_the_end_of_the_world;
- In this case, however, determining the type of the variable *the_date_of_the_end_of_the_world* (e.g. if we want to use it with the *sizeof* operator) becomes troublesome. Without a tag it has to be denoted as:
  - **sizeof**(**struct** {**int** year, month, day;})
- We find it too complex and unreadable, compared t*sizeof(struct DATE).*

Declaring the structures

- A structure could be a field inside another structure.

```
struct STUDENT {
   char  name[26];
   float time;
   int   recent_chapter;
   struct DATE last_visit;
} HarryPotter;
```

  - HarryPotter.last_visit.year = 2012;
  - HarryPotter.last_visit.month = 12;
  - HarryPotter.last_visit.day = 21;

Outline

1. Arrays vs. structures: different aggregates for different purposes
    1. Arrays of pointers as multidimensional arrays
    2. Declaring arrays: traps and puzzles
    3. The structures: why?
    4. **Declaring and initializing structures**
    5. Pointers to structures and arrays of structures
    6. Basics of recursive data collections
2. Quiz

Structures – a few important rules

- A structure's field names may overlap with the tag names, and this is generally not considered a problem, although it may cause some difficulty in reading and understanding the program.

```
struct STRUCT {
    int STRUCT;
} Structure;

Structure.STRUCT = 0; /* STRUCT is a field name here */
```

Structures – a few important rules

- It may happen that a particular compiler doesn't like it when a structure's tag name overlaps with a variable's name.

```
struct STR {
 int field;
} Structure;
int STR;

Structure.field = 0;
STR = 1;
```

Structures – a few important rules

- Two structures **can contain fields with the same names** – the following snippet is correct

```
struct {
 int f1;
} str1;

struct {
 char f1;
} str2;

str1.f1 = 32;
str2.f1 = str1.f1;
```

Initializing structures

- Structures can be initialized as early as at the time of declaration. The structure's initiator is enclosed in curly brackets and contains a **list of values assigned to the subsequent fields**, starting from the first.

```
struct DATE date = { 2012, 12, 21 };
```

Initializing structures

```
struct DATE date = { 2012, 12, 21 };
```

- This initiator is equivalent to the following sequence of assignments:
  - date.year = 2012;
  - date.month = 12;
  - date.day = 21;

Initializing structures

```
struct STUDENT he = { "Bond", 3.5, 4, { 2012, 12, 21 }};
```

- The initiator of this form is functionally equivalent to the following assignments:
  - strcpy(he.name, "Bond");
  - he.time = 3.5;
  - he.recent_chapter = 4;
  - he.last_visit.year = 2012;
  - he.last_visit.month = 12;
  - he.last_visit.day = 21;

Initializing structures

- Due to the completeness of the inner initializer, it can be written in the following, simplified form:
  - **struct** STUDENT he = { "Bond", 3.5, 4, 2012, 12, 21 };
- This type of simplification (omitting internal curly braces) can also be applied in the following case (with caution, though):
  - **struct** STUDENT she = { "Mata Hari", 12., 12, { 2012 } };
  - she.last_visit.month = 0;
  - she.last_visit.day = 0;

Initializing structures

- What happens when we apply an “empty” initializer?

```
struct STUDENT nobody = { };
```

- Here's the answer:
  - strcpy(nobody.name, "");
  - nobody.time = 0.0;
  - nobody.recent_chapter = 0;
  - nobody.last_visit.year = 0;
  - nobody.last_visit.month = 0;
  - nobody.last_visit.day = 0;

Outline

1. Arrays vs. structures: different aggregates for different purposes
    1. Arrays of pointers as multidimensional arrays
    2. Declaring arrays: traps and puzzles
    3. The structures: why?
    4. Declaring and initializing structures
    5. **Pointers to structures and arrays of structures**
    6. Basics of recursive data collections
2. Quiz

Pointers to structures

- Let's declare a pointer to the structures of type *struct STUDENT*.

```
struct STUDENT *sptr;
```

- If we want to allocate memory for one structure, we can do it in the following way:
  - sptr = (**struct** STUDENT *) malloc(**sizeof**(**struct** STUDENT));

Pointers to structures

- The *sptr* points to a piece of memory that can be used as if it were a “regular” structure of type *struct STUDENT*.
- The following dereference identifies the entire structure:
  - *sptr
- Unfortunately, an expression like the one below is **invalid**:
  - *sptr.time= 1.0;

Pointers to structures

- This error is caused by the very **high priority of the selection operator** – it's higher than the priority of the dereference operator, so the entire expression will be interpreted as follows:
  - *(sptr.time)
- It forces us to take into account the **mutual arrangement** of both these operators and to write the expression with **additional parentheses:**
  - (*sptr).time

Pointers to structures

- This means that given form:

```
pointer -> field
```

- is by definition equal to the form:
  - (*pointer).field

Pointers to structures

- We can use this information to assign some values to the fields of the newly allocated structure:
  - strcpy(sptr -> name, "Dobby");
  - sptr -> time = 0.1;
  - sptr -> recent_chapter = 0;
  - sptr -> last_visit.year = 2002;
  - sptr -> last_visit.month = 1;
  - sptr -> last_visit.day = 1;

Unions

- Syntactically, they’re both very similar. You only have **to replace the keyword *struct* with *union*** and the whole declaration remains valid. The real clue is hidden deeper – inside the computer‘s memory.

Unions

- They both have **two fields** named *a* and *b*. Accessing the fields is identical in both cases:
  - Data1.a = 0;
    Data2.b = 0;

```
struct {
        int a;
        int b;
} Data1;
union {
        int a;
        int b;
} Data2;
```

Unions

- In a structure, all its fields are placed **successively**. Every field has its own part of the computer memory. In other words: each field is a separate world and they don't overlap. Modifying the *a* field will leave the *b* field intact and vice versa.
- This also means that the size of the structure **is not less than the total size of all its fields**.

Unions

- In contrast, in a union, all its fields are placed **in the same memory location**. This means that the *a* and *b* fields share the same part of the memory.
- This fact has at least one interesting implication: the union's initializer **has to have one element.**
- It means also that modifying the *a* field will modify the *b* field **immediately.**

Unions

- The program will output two values: 0 and 1.

```c
#include <stdio.h>

int main(void) {
    struct {
        int a;
        int b;
    } Data1 = { 0, 0 };

    union {
        int a;
        int b;
    } Data2 = { 0 };

    Data1.a++;
    Data2.a++;
    printf("%d %d\n", Data1.b, Data2.b);
    return 0;
}
```

Unions

```c
#include <stdio.h>

int main(void) {

    union {
        int a;
        float b;
    } Data;

    scanf("%d", &Data.a);
    printf("\n%f\n", Data.b);
    return 0;
}
```

Unions

- The Data union is equipped with two fields of radically different types: *int* and *float*. As you already know, these types use a **completely different method of internal representation**. Assigning the *a* field with the value of 1 won't set the *b* field with anything even close to 1.

Outline

1. Arrays vs. structures: different aggregates for different purposes
    1. Arrays of pointers as multidimensional arrays
    2. Declaring arrays: traps and puzzles
    3. The structures: why?
    4. Declaring and initializing structures
    5. Pointers to structures and arrays of structures
    6. **Basics of recursive data collections**
2. Quiz

Structures + Pointers = Lists

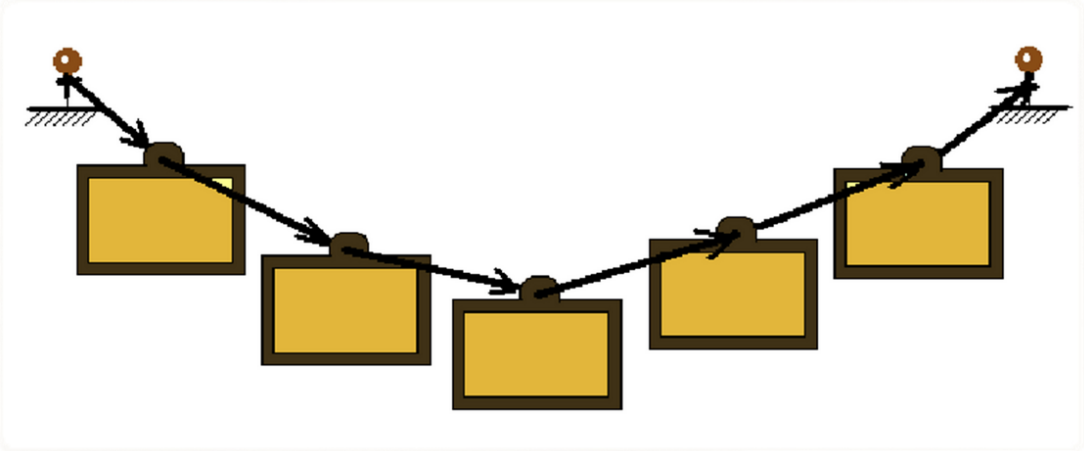
- A structure cannot be a field of itself, but any of the structure's fields can be a pointer to the structure currently declared.

```
struct ELEMENT {
   int                  value;
   struct ELEMENT   *next;
};
```

- This is a **recursive declaration**, as the entity being declared refers to itself before the declaration is completed.

Structures + Pointers = Lists

- The idea of a one-way linked list is based on the observation that if we have a multitude of data (e.g. integers), we can link them together (some could say "**aggregate them**"), like beads on a thread, and attach the thread to any steady point.



Structures + Pointers = Lists

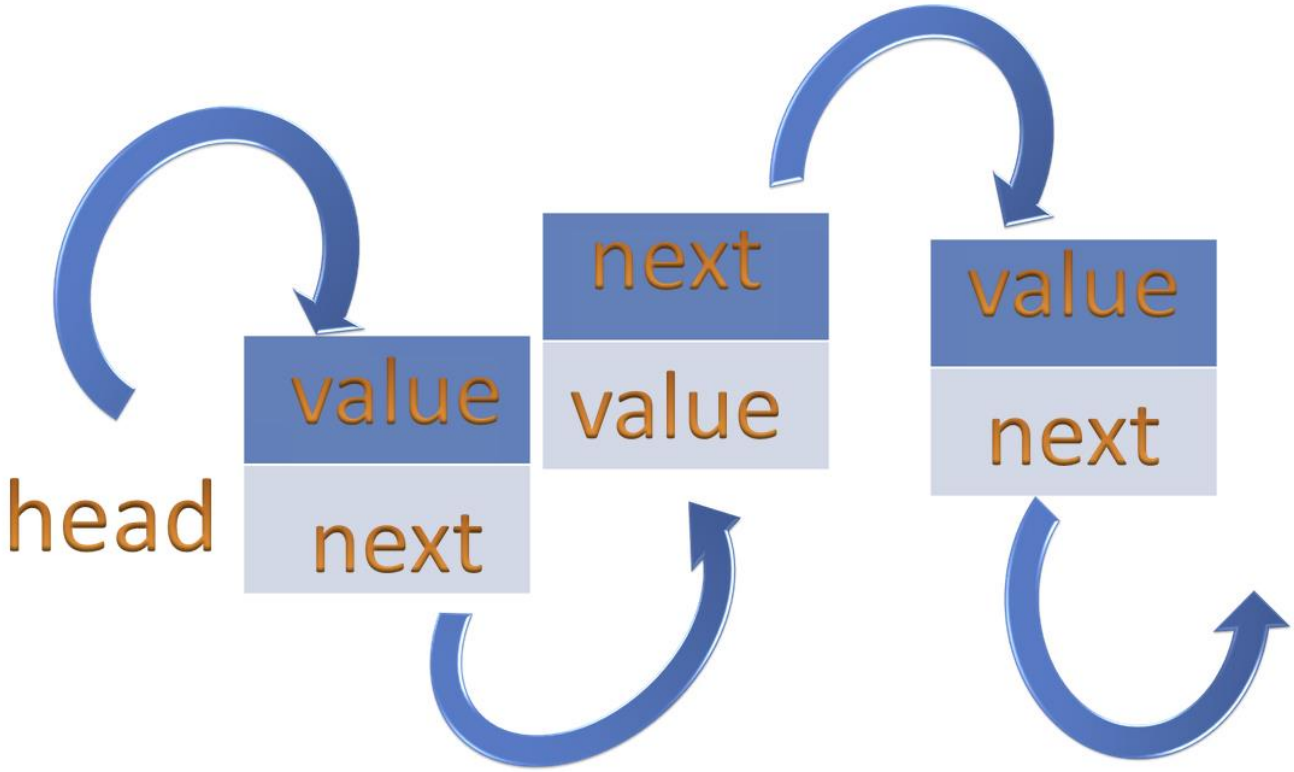
- We suggest you imagine something like this:
  - the threaded elements are structures of the same type;
  - each of these structures will store an integer value
  - we assume that every structure will have a kind of hook which can “attach” to the next bead in the chain;
  - the role of coupling will be played by one specific field that can store a pointer to the next element inside the chain;
  - the thread has to be permanently attached so we need a nail; we assign that role to the pointer variable commonly called "*head*" or "*list header*".

Structures + Pointers = Lists

- Look again at the ELEMENT structure declaration:
  - a field named *value* will serve as storage for any useful values;
  - a field named *next* will be used to “attach” to the next structure in the chain;
  - the first structure in the chain is “attached” to a head variable, declared as follows:
    - **struct** ELEMENT *head;
  - the last structure in the chain will be attached to nothing, so we’ll assign the NULL pointer value to its *next* field.

Structures + Pointers = Lists

- Let's imagine a thread with three beads on it



Structures + Pointers = Lists

```
struct ELEMENT {
 int          value;
 struct ELEMENT  *next;
};

struct ELEMENT *head, *ptr;

/* the one-way linked list is created here
   we don't know yet how it was done
   we only know that the head points to element #1 */
ptr = head;
/* ptr points to the first element now; we will move it
   through all elements until we reach the end */
while(ptr != NULL) {
    /* print the value stored in the element */
    printf("value = %d\n", ptr -> value);

    /* move ptr to the next element */
    ptr = ptr -> next;
}
printf("done!");
```

Outline

1. Arrays vs. structures: different aggregates for different purposes
    1. Arrays of pointers as multidimensional arrays
    2. Declaring arrays: traps and puzzles
    3. The structures: why?
    4. Declaring and initializing structures
    5. Pointers to structures and arrays of structures
    6. Basics of recursive data collections
2. **Quiz**

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
struct S1 {
   int p1;
};
struct S2 {
   int p1;
   struct S1 s1;
};
int main(void) {
   struct S2 s2 = { 4, 5 };
   printf("%d",s2.p1 + s2.s1.p1);
   return 0;
}
```

- ○ the program outputs 9
- ○ the program outputs 4
- ○ the program outputs 5

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
#include <stdlib.h>
struct S1 {
   int p1,p2;
};
struct S2 {
   int p1;
   struct S1 s1;
   int p2;
};
int main(void) {
   int s = 0;
   struct S2 s2 = { 1, 2, 3, 4 };
   struct S2 *p;
   p = (struct S2 *)malloc(sizeof(struct S2));
   *p = s2;
   s2. p1 = 0;
   s = p->p1 + s2.p1 + p->p2 + p->s1.p2;
   free(p);
   printf("%d",s);
   return 0;
}
```

- ○ the program outputs 32
- ○ the program outputs 16
- ○ the program outputs 8

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
   int **pt;
   pt = (int **)malloc (2 * sizeof (int *));
   pt[0] = (int *) malloc(sizeof(int) * 2);
   pt[1] = (int *) malloc(sizeof(int) * 2);
   pt[1][1] = pt[0][0] = 1;
   pt[0][1] = pt[1][0] = 2;
   printf("%d", pt[0][0] / pt[1][0]);
   free(pt[1]);
   free(pt[0]);
   free(pt);
   return 0;
}
```

- ○ the program outputs 0.5
- ○ the program outputs 1
- ○ the program outputs 0

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
   int t[2][3] = { { 3, 2, 1 }, { 1, 2, 3} };
   printf("%d", sizeof(t) / sizeof(t[1]));
   return 0;
}
```

- the program outputs 6
- the program outputs 2
- the program outputs 3

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
   int t[2][3] = { { 3, 2, 1 }, { 1, 2, 3} };
   printf("%d", sizeof(t) / sizeof(t[1][1]));
   return 0;
}
```

- the program outputs 2
- the program outputs 6
- the program outputs 3

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int main(void) {
   printf("%c", "ABCD"[2]);
   return 0;
}
```

- ◯ the program outputs A
- ◯ the program outputs C
- ◯ the program outputs B