CHAPTER 2 SUMMARY

Structs

2.10 Structures – why do we need them?

- **Problem**: Design a data structure that can store information about students attending our course. We need to store
  - the name of each student,
  - their time spent studying the chapters and
  - the number of the last completed chapter.
  - We know that the total number of students will not exceed 100,000 and this leads us to write the following declaration

```
string student_name[100000];
```

  - suppose that the first registered student is Mr. Bond (James Bond). Let's store the information in our array:

```
student_name[0] = "Bond";
```

2.10 Structures – why do we need them?

- The time spent on the site will be stored as a float.
- The number of hours will be represented as a decimal fraction. This leads us directly to the following statement declaration:

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

  - Mr. Bond has spent three hours and thirty minutes studying the course:

```
student_time[0] = 3.5;
```

2.10 Structures – why do we need them?

- The main issue here is that the data concerning the same object (a student) is **dispersed between three variables**, although it should logically exist as a **<u>consolidated unit</u>**.
- Handling multiple arrays is cumbersome and error-prone, and when life forces us to collect additional information (e.g. e-mail address) we’re going to need to declare another array and make a lot of changes.
- The array is an aggregate of elements.
  - The elements are numbered and are of the same type.
- Can we use **an aggregate whose elements could be of different types**? Yes, it's a great idea! This magical aggregate is called a **structure**.

2.10 Structures – why do we need them?

- **A structure can contain any number of any elements of any type**.
  - Each of these elements is called a field.
- Each field is identified by its name, not by its number.
  - The field names must be unique and cannot be doubled within a single structure.
- Structure declaration:
  - The declaration of the structure always starts with the keyword **struct**
  - There is a so-called struct **tag** after the keyword it's the name of the structure itself.
  - Opening **curly bracket** - a signal that the declaration of fields begins at this point.
  - **Structure fields**: the first is a *string* and is called *name*; the second is a *float* and is called *time*; the third is an *int* and it’s called recent_chapter.
  - The declaration ends with the **closing curly bracket followed by the semicolon**.

2.10 Structures – why do we need them?

- For our example:

```
struct STUDENT {
    string  name;
    float   time;
    int     recent_chapter;
};
```

- We want to emphasize that the previous declaration **doesn't create a variable**, but only describes the structure we're going to use in our program.
- If we want to declare a variable as a structure, we can do it in **one of two possible ways**

```
struct STUDENT stdnt;
STUDENT stdnt2;
```

2.10 Structures – why do we need them?

- This declaration sets up two variables (**structured variables**) named *stdnt* and *stdnt2* respectively.
- The variables are of type *struct STUDENT* or just *STUDENT* (notice, that the structure declaration creates a new type name).
- We know that this variable consists of three named fields.
- To access a strut variable fields the “C++” language offers a specialized **selection operator** designed for structures and is denoted as a single character **.** (dot).
- The priority of the selection operator is very high, equal to the priority of the **[ ]** operator used with arrays.

2.10 Structures – why do we need them?

- It's a binary operator. Its **left argument must identify the structure** while **the right one must be the name of the field known in this structure**.
- The result of this operator is the selected field of the structure and therefore the expression containing this operator is sometimes called a **selector**.
- In the following example the selector results in the selection of a field called *time*. The type of this expression is the type of the selected field and is an l-value.

STDNT.TIME

- Consequently, we can use both of these selectors:

  stdnt.time = 1.5; and

  float t;   t = stdnt.time;

2.10 Structures – why do we need them?

- Virtually any data can be used as a structure’s field: scalars, arrays and pretty much almost all of the structures. We say “almost” because **the structure can’t be a field itself**.
- We can aggregate structures inside an array, so if we want to declare an array of STUDENT structures, we can do it in this way :

```
STUDENT STDNTS[100000];
```

- Access to the selected fields requires two subsequent operations:
  - in the first step, the [] operator will index the array in order to access the structure we need.
  - in the second step, the selection operator selects the desired field.
- This means that if we want to select the *time* field of the fourth *stdnts*' element, we'll write it like this: stdnts[3].time

- We now collect all these assignments that are performed for the three separate arrays:
- stndts[0].name = "Bond";
  stndts[0].time = 3.5;
  stdnts[0].recent_chapter = 4;

2.10 Declaring the structures

Example:

- Declare a structure to store the date. It's equipped with three fields, each of type int, named year, month and day, which clearly denote their role and purpose.
- The first possible way of declaring the structure is :

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

- We can write this declaration much more compactly:

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

- Both variants are the same.

2.10 Declaring the structures

- This declaration doesn't create any new variables, but **only announces to the compiler our intention to use this structure tag to declare new variables**.
- The new variable would be declared, for example, in this way: 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 an array of structures: DATE Visits[100];

2.10 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, it becomes harder to determine 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). Without a tag, we have to denote it as: sizeof(struct {int year, month, day;})
- We think this is way too complex and unreadable, compared to *sizeof(struct DATE)*.

2.10 Declaring the structures

- Accessing a single structure stored in the array is easy. If we want to modify the data of the first visit, we do this:

```
Visits[0].year = 2012;
Visits[0].month = 1;
Visits[0].day =  1;
```

- We can also define 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];
DATE current_date;
```

2.10 Structures – why do we need them?

- **A structure can be a field inside another structure**. Imagine that we have to extend our STUDENT structure and add a field to save the date when a particular student visited the course last time. We can do it using the following declaration:

```
struct STUDENT {
   string  name;
   float time;
   int   recent_chapter;
   struct DATE last_visit;
} HarryPotter;
```

- Two subsequent selection operations will be used to go deeper into the structure i.e. first we select a structure within the structure, and then we select the desired field of the inner structure.
  - For exmaple:

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

  - Pop quiz: when did Harry last visit us?

2.11 DECLARING AND INITIALIZING STRUCTURES

2.11 Structures – a few important rules

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

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

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

- It may be the case that the particular compiler you're working with doesn't like it when a structure's tag name overlaps with the variable's name; therefore, it's better to avoid tricks like the

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

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

2.11 Structures – a few important rules

- **Two structures can contain fields with the same names**

```
struct {
 int f1;
} str1;

struct {
 char f1;
} str2;

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

2.11 Initializing structures

- Structures can be initialized early as **at the time of declaration using initiators**.
- The structure’s initiator is enclosed in curly brackets and contains **a list of values assigned to the subsequent fields**, starting from the first.
- The values listed in the initiator need to conform to the types of fields.
- If the initiator contains fewer elements than the number of the structure’s fields, it is presumed that the list is implicitly extended with zeros.
- If the particular field is an array or a structure, it should have its own initiator, which is also subject to be extended with zeros. If the “internal” initiator is complete, we can omit the surrounding curly brackets.

2.11 Initializing structures

- Example:
  - The initiator is equivalent to the following sequence of assignments:

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

```
date.year = 2012;
date.month = 12;
date.day = 21;
```

  - The initiator of this form is functionally equivalent to the following assignments:

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

```
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;
```

2.11 Initializing structures

- Due to the completeness of the inner initializer, we can write the following, simplified form:

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

- This simplification (omitting the internal curly brackets) cannot be applied in the following case:

```
STUDENT she = { "Mata Hari", 12., 12, { 2012 }  };
```

- The internal initiator, referring to the *last_visit*field, doesn't cover all the fields. This means that it'll be equivalent to the following sequence of assignments:

```
she.name= "Mata Hari";
she.time = 12.;
she.recent_chapter = 12;
she.last_visit.year = 2012
she.last_visit.month = 0;
she.last_visit.day = 0;
```

2.11 Initializing structures

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

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

- Answer:

```
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;
```

All fields will be initialized with default values according their types.

structs within a struct

```
struct  employeeType
{
    string firstname;
    string middlename
    string lastname;
    string empID;
    string address1;
    string address2;
    string city;
    string state;
    string zip;
    int hiremonth;
    int hireday;
    int hireyear;
    int quitmonth;
    int quitday;
    int quityear;
    string phone;
    string cellphone;
    string fax;
    string pager;
    string email;
    string deptID;
    double salary;
};
```

versus

```
struct addressType
{
    string address1;
    string address2;
    string city;
    string state;
    string zip;
};

struct nameType
{
    string first;
    string middle;
    string last;
};

struct dateType
{
    int month;
    int day;
    int year;
};

struct contactType
{
    string phone;
    string cellphone;
    string fax;
    string pager;
    string email;
};

struct employeeType
{
    nameType name;
    string empID;
    addressType address;
    dateType hireDate;
    dateType quitDate;
    contactType contact;
    string deptID;
    double salary;
};
```