User-defined data types

User-defined data types

In practice, there is often a need to work with a collection of different types as a single whole.

The programmer can define his own data types that have the necessary structure.

These data types are called user-defined types (i.e., programmer-defined types).

User-defined data types

User-defined data types include:

- Enum;
- Struct;
- Union;
- Class.

Enum

Enum is a data type that contains a set of user-defined named constants.

After definition, an enum is used almost the same as integer types.

Enum

When declaring an enumeration, the keyword enum is used.

```
enum Auto {Dodge, GMC, Chrysler, Chevrolet};
```

The enumeration may not have a name:

```
enum {SUV, Sedan, Truck} CarType;
```

Enum

When creating an enumeration type variable in C, you must precede the enumeration type with a keyword enum.

```
enum Auto {Dodge, GMC, Chrysler, Chevrolet};

enum Auto aut = GMC;
```

Enum

When introducing an alias for an enumeration (using the typedef operator), you do not need to specify the enum keyword in C:

```
typedef enum Auto {Dodge, GMC, Chrysler,
Chevrolet} AUTO;

AUTO aut = GMC;
```

Enum

При создании переменной типа перечисления в языке C++ перед типом перечисления ключевое слово enum можно не указывать.

```
enum Auto {Dodge, GMC, Chrysler, Chevrolet};

Auto aut = Chrysler;
```

Enum

```
typedef enum Auto {Dodge, GMC, Chrysler,
Chevrolet} AUTO;
```

C	C++
`enum Auto aut = Chevrolet;` `AUTO aut = Chevrolet;`	`Auto aut = Chevrolet;` `AUTO aut = Chevrolet;`

Enum

By default, integer values are assigned to enumeration elements in ascending order, starting from zero:

```
enum Auto {Dodge, GMC, Chrysler, Chevrolet};
```

```
Dodge = 0
GMC = 1
Chrysler = 2
Chevrolet = 3
```

Enum

```
void GetTheBestModel(Auto aut){
      switch(aut){
            case Dodge:
                  std::cout<<“Dodge RAM 3500”<<std::endl;
            break;

            case Chevrolet:
                  std::cout<<“Chevrolet Suburban”<<std::endl;
            break;
      }
}
```

Enum

Members can be initialized with values other than their default values:

```
enum Auto {Dodge = 3, GMC,
           Chrysler = 7, Chevrolet};
```

In this case, fields for which a value is not explicitly specified will be initialized with the next integer value relative to the field on the left.

Enum

```
enum Auto {Dodge = 3, GMC,
           Chrysler = 7, Chevrolet} Car;
```

```
Dodge = 3
GMC = 4
Chrysler = 7
Chevrolet = 8
```

```
Car = 0
```

Enum

Enumeration elements can have the same values, but their names must be different.

```
enum Auto { Dodge = 0, GMC,
            Chrysler = 0, Chevrolet };
```

```
Dodge = 0
GMC = 1
Chrysler = 0
Chevrolet = 1
```

Enum

An integer type value can be cast to an enumeration type. But, if a given value does not match any element of the enumeration, then there will be no corresponding symbolic name for it.

```
enum Auto {Dodge, GMC, Chrysler, Chevrolet};

...

Auto aut = (Auto)7;
```

Enum

An integer type can be assigned an enumeration type value:

```
enum Auto {Dodge, GMC, Chrysler, Chevrolet};
...
int val = GMC;                    val = 1
```

Enum

The size of an enumeration type does not exceed the size of a signed integer type(int)

sizeof(Auto) ≤ sizeof(int)

Struct

A structure is a user-defined data type that is built on the basis of previously defined data types, which can be either standard built-in data types or other structures.

The structure allows you to work with a collection of data of various types as a single whole.

Struct

A structure declaration (description) is a description of the internal structure of a composite data type, based on which the compiler will create instances of the user-defined type and manipulate them.

A structure declaration begins with the keyword `struct`. Then the types and names of the structure fields are indicated in curly braces.

When declaring a structure, memory is not reserved for it. Memory is reserved for a variable of this type. The variable can be created immediately after the declaration.

Struct

```
struct Student {
    int ID;
    int Age;
    char Name[20];
    float AverageMark;
} Stud;
```

Creating a variable Stud of type Student.

Struct

Creating structure instances:

When creating an instance of a structure in C, you must specify the struct keyword:

```
struct Student stud;
```

Struct

When introducing an alias for a structure (using the typedef operator), you do not need to specify the struct keyword in C:

```
typedef struct Student {
    int ID;
    int Age;
    char Name[20];
    float AverageMark;
} STUDENT;
...
STUDENT stud;
```

Struct

When creating an instance of a structure in C++, the struct keyword can be omitted:

```
Student stud;
```

Struct

```
typedef struct Student {
    int ID;
    int Age;
    char Name[20];
    float AverageMark;
} STUDENT;
```

C	C++
`struct Student stud = {0};` `STUDENT stud = {0};`	`Student stud = {0};` `STUDENT stud = {0};`

Struct

Elements of a structure can be pointers to the same structure:

```
struct ListNode {
    int val;
    ListNode* Next;
    ListNode* Prev;
};
```

Struct

The same initialization rules apply for structures as for local and global variables.

Fields of structures created in the global scope are initialized to zero values by default.

Fields of structures created in the local scope are not initialized by default (they can contain any values).

Struct

Accessing structure fields:

To access the fields of a structure, use the “.” operator, followed by the field name.

```
Student stud;
stud.ID = 1122;
strcpy(stud.Name, “John Wick”);
stud.AverageMark = 2.71;
int age = stud.Age;
```

Struct

If you need to access a structure field through a pointer, you can use the operator "->"

```
Student* p_stud = new Student;

p_stud->ID = 1122;          Эквивалентно (*p_stud).ID
strcpy(p_stud->Name, “John Wick”);
p_stud->AverageMark = 2.71;
int age = p_stud->Age;
```

Struct

When accessing a structure by pointer, you can also first dereference the pointer and then access the field using the “.” operator. :

```
Student* p_stud = new Student;


(*p_stud).ID = 1122;
```

Struct

Structures can be initialized using initialization lists in a similar way to arrays. In this case, the fields will be initialized in the order of their declaration in the structure.

If the number of elements in the initialization list is less than the number of structure fields, then the remaining fields will be assigned zero values.

Struct

```
Student stud = {1122, 18, “John Wick”};
```

```
stud.ID = 1122
stud.Age = 18
stud.Name = “John Wick”
stud.AverageMark = 0
```

Struct

The address of a structure field can be obtained using the address operator (&):

```
Student stud;
int* p_age = &(stud.Age);

*p_age = 19;                    stud.Age = 19
```

Struct

A structure field can be a built-in type, another structure, a union, an enumerated type, or a class (in C++).

```
struct Point {
     int X;
     int Y;
};

struct Circle {
     Point Center;
     int Radius;
};
```

Struct

```
Circle c = {0};

c.Center.X = 5;
c.Center.Y = 7;
c.Radius = 3;
```

Struct

If a structure is used only as a field of another structure, then it can be declared inside that structure:

```
struct Circle {
    struct Point {
        int X;
        int Y;
    } Center;
    int Radius;
};
```

Such structures are called nested.

Struct

```
Circle c = {0};

c.Center.X = 5;
c.Center.Y = 7;
c.Radius = 3;
```

Struct

Nested structures may not have a name. In this case, they are called anonymous.

```
struct Circle {
    struct {
        int X;
        int Y;
    } Center;
    int Radius;
};
```

Struct

The size of the structure is not less than the sum of the sizes of the types of fields included in it.

You can find out the size of a structure using the sizeof() operator

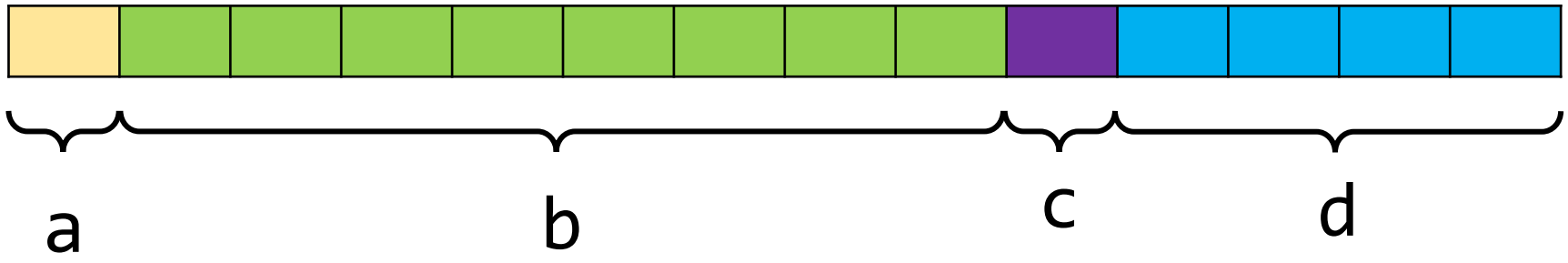
```
unsigned stud_size = sizeof(Student);
```

Struct

Location of structure fields in memory:

The fields of the structure are located in memory sequentially one after another

```
struct A {
    char a;
    double b;
    bool c;
    float d;
};
```



Struct

```
A a = {0};
std::cout << sizeof(A) << std::endl;
```

sizeof(A) = 24

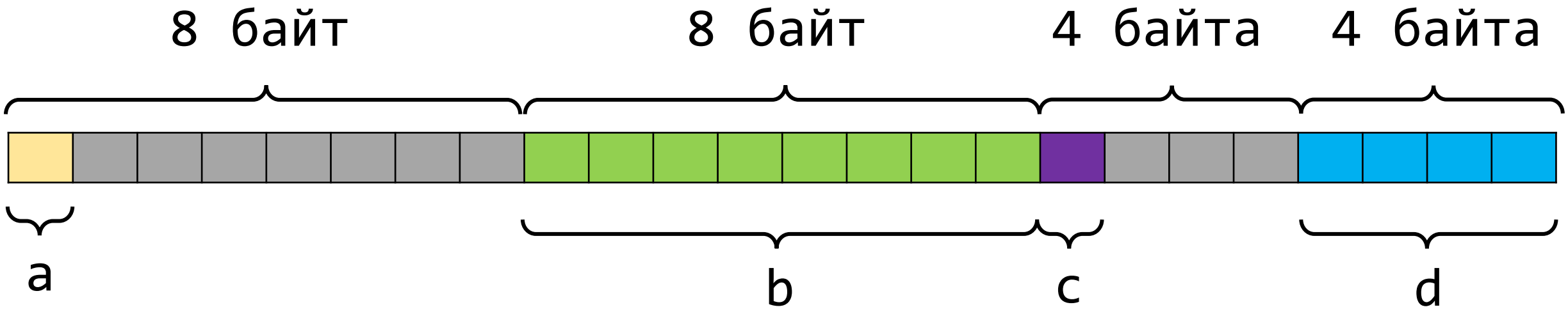
Struct

Some compilers, when performing optimization, place structure fields in such a way as to minimize the time it takes to access any field.

This means that any data element must be placed at an address that is a multiple of the size of the element's type.

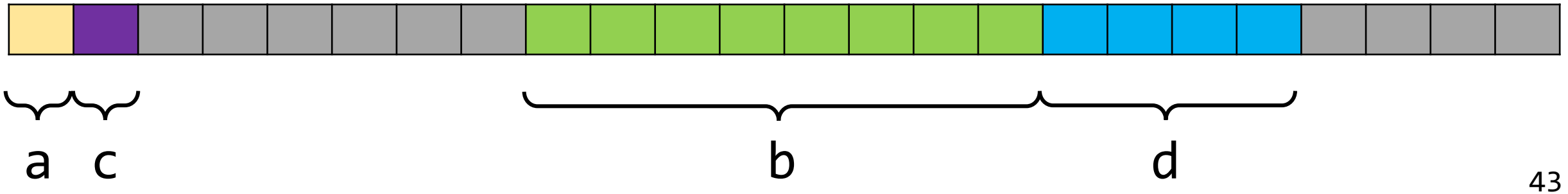
Struct

Placement of structure fields during optimization (VS):



Struct

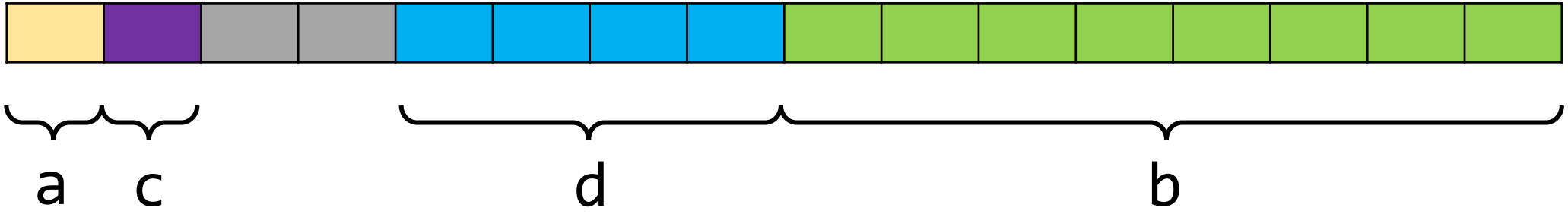
```
struct A {
      char a;
      bool c;
      double b;
      float d;
};
```



Struct

```
struct A {
    char a;
    bool c;
    float d;
    double b;
};
```

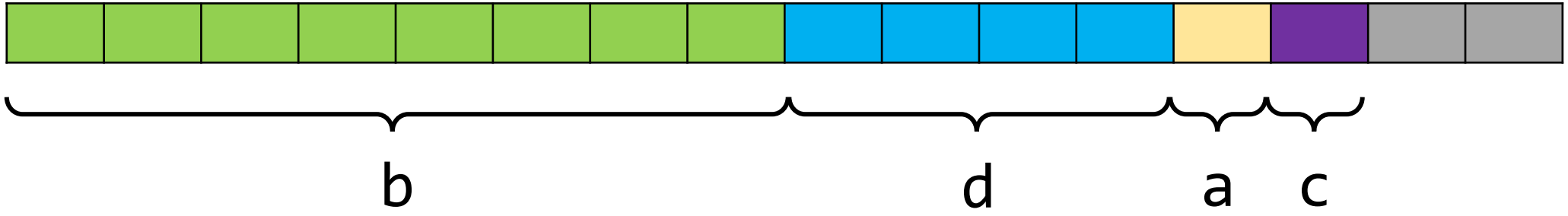
sizeof(A) = 16



Struct

```
struct A {
    double b;
    float d;
    char a;
    bool c;
};
```

sizeof(A) = 16



Struct

To prevent optimization of the placement of structure fields, its declaration can be placed inside #pragma pack directives:

```
#pragma pack(push, 1)


#pragma pack(pop)
```

Struct

```
#pragma pack(push, 1)
struct A {
    char a;
    double b;
    bool c;
    float d;
};
#pragma pack(pop)
```

sizeof(A) = 14

Struct

One structure can be assigned to another using the = operator. In this case, the fields of the structure are copied byte-by-byte.

```
Student stud = {1122, 43, “John Wick”, 2.71};
Student stud2 = {1123, 75, “Chuck Norris”, 5.45};


stud = stud2;
```

Struct

When casting a structure to the type of another structure, the internal fields will be treated according to the placement of the fields of the second structure in memory.

It is desirable that the structures have the same size.

Struct

```
struct Circle {
    struct {
        int X;
        int Y;
    } Center;
    int Radius;
};
```

```
struct Circle2 {
    int Radius;
    struct {
        int X;
        int Y;
    } Center;
};
```

Struct

```
Circle c1 = {7, 5, 3};
Circle2 c2 = *((Circle2 *)&c1);
```

```
c1.Center.X = 7
c1.Center.Y = 5
c1.Radius = 3
```

```
c2.Radius = 7
c2.Center.X = 5
c2.Center.Y = 3
```

Struct

Using type conversion, a structure can also be converted to built-in data types.

```
Circle c = {0x11223344, 0xAABBCCDD, 5};

long long val = *((long long *)&c);
```

val = 0xAABBCCDD11223344

Struct

You can also perform the reverse operation, i.e. Casting built-in types to a structure type. However, such a transformation does not make sense in most cases.

Struct

Bit fields:

A bit field is a special type of structure field that is used to pack data more tightly. That is, several logically different variables can be stored in one physical variable of suitable length.

The use of bit fields makes it possible to access individual bits.

Bit fields can be of any integer type.

Struct

The bit field is accessed in the usual way - by name.

The bit field address cannot be obtained.

The bitfield name may be missing. This is used to implement the required arrangement of bit fields in memory.

Struct

It should be remembered that operations with individual bits are implemented much less efficiently than operations with bytes and machine words (usually 4 or 8 bytes, the same size as the pointer).

Using bit fields saves memory for data, but increases the amount of machine code and the time required to process it.

Struct

When describing a bit field, the length of the field in bits is indicated after the name, separated by a colon:

```
struct BitMap {
    unsigned char Bit0 : 1;
    unsigned char Bit1 : 1;
    unsigned char : 5;
    unsigned char Bit7 : 1;
};
```

Struct

As many bytes are allocated for bit fields as are minimally necessary to store all bit fields.

```
struct BitMap {
    unsigned char Bit0 : 1;
    unsigned char Bit1 : 1;
    unsigned char Bits2_6 : 5;
    unsigned char Bit7 : 1;
};
```

sizeof(BitMap) = 1

```
struct BitMap2 {
    unsigned char Bit0 : 1;
    unsigned char char Bit1 : 1;
    unsigned char Bits2_6 : 5;
    unsigned char Bit7 : 1;
    unsigned char Bit8 : 1;
};
```

sizeof(BitMap2) = 2

Struct

```
BitMap bm = {0};
bm.Bit0 = 1;                    bm = 0b00000001
bm.Bit7 = 1;                    bm = 0b10000001
...
int res = bm.Bit7;              res = 1
```

Struct

Operations with a structure containing bit fields can be replaced with shifts and bitwise logical operations. But this is less convenient:

```
unsigned char bitmap = 0;
bitmap |= (1 << 0);
bitmap |= (1 << 7);

int res = (bitmap >> 7);          res = 1
```

Struct

Passing structures to a function:

Structures can be passed to a function, like variables of built-in types, by value, by pointer, and by reference.

Struct

```
void fun(Student St){
     St.ID += 10;
}

void fun2(Student* St){
     St->ID += 10;
}

void fun3(Student& St){
     St.ID += 10;
}
```

```
struct Student {
     int ID;
     int Age;
     char Name[20];
     float AverageMark;
};
```

Struct

```
Student stud = {1122, 43, “John Wick”, 2.71};
Student stud2 = {1123, 75, “Chuck Norris”, 5.45};
```

```
fun(stud);          stud.ID = 1122
fun2(&stud);        stud.ID = 1132
fun3(stud2);        stud2.ID = 1133
```

Struct

Writing structures to a file:

The structure can be, like any other data type, written to a file byte by byte.

```
std::fstream file(“circle.bin”,
std::fstream::out);
Circle c = {7, 5, 3};

file.write((char *)&c, sizeof(Circle));
```

Struct

Reading structures from a file:

A structure can, like any other data type, be read from a file one byte at a time.

```
std::fstream file(“circle.bin”,
std::fstream::in);
Circle c = {0};

file.read((char *)&c, sizeof(Circle));
```

Struct

If the structure contains a field with a pointer type, a class (for example, a file stream, etc.) or if the field stores information that changes from run to run (for example, a process identifier), then a byte-by-byte write of such a structure to a file may not make sense.

To write such structures to a file and read them from it, you need to implement your own function.

Struct

```
struct DataHandler {
    int* Data;
    int Size;
};

int num = 500;
DataHandler dh = {new int[num], num};
```

Writing such a structure to a file byte-by-byte does not make sense, since the address of the array that contains the Data field will be different each time the program is launched.

Struct

You can write such a structure to a file as follows:

```
void WriteData(std::string& filename, DataHandler& Dh){
     std::fstream file(filename.c_str(), std::fstream::out);

     file.write((char *)&(Dh.Size), sizeof(int));
     for(int i = 0; i < Dh.Size; i++)
          file.write((char *)&(Dh.Data[i]), sizeof(int));

     file.close();
}
```

Struct

When using file streams for structures to work with files, you can overload the write to stream << and read from stream >> operators.

Struct

```
std::fstream& operator << (std::fstream& out, Circle& C){
     out << “Center.X = ” << C.Center.X << std::endl;
     out << “Center.Y = ” << C.Center.Y << std::endl;
     out << “Radius = ” << C.Radius << std::endl;

     return out;
}
...
std::fstream file(“circle.bin”, std::fstream::out);
Circle c = {7, 5, 3};
file << c;
```

Struct

In C++, structs are classes, all of whose fields are externally accessible by default.

As a consequence, in C++, structures can have their own functions. Such functions are called methods.

They are called on a specific struct instance, have access to fields, and cannot be called independently (i.e. not on a struct instance).

Struct

```
struct DataHandler {
    int* Data;
    int Size;
    int* Allocate() { Data = new int[Size];
                      return Data;}
    void SetSize(int size) { Size = size;}
};
```

Struct

```
DataHandler dh;
dh.SetSize(500);
dh.Allocate();
```

Struct

Since structures in C++ are classes, you can also create special functions for them, in which you can specify actions when initializing an instance of a structure (constructors) and actions when destroying an instance of a structure (destructor).

Struct

Methods are common to all instances of a structure, so the presence of methods does not affect the size of the structure.

When calculating the size of a structure, only the field types (including alignment) matter.