Object-oriented programming. Classes.

Object-oriented programming

The paradigm of object-oriented programming:

Representing a program as a collection of objects that execute tasks by invoking methods designed for them. The execution of a task may vary for objects of the same type depending on the parameters they are given. Classes can form a hierarchy, known as an inheritance hierarchy.

Object-oriented programming

When using an object-oriented approach to software development, both runtime and memory usage tend to increase. While this is not critical for user applications, it should be considered when developing software for specialized platforms.

However, when creating large software products, the object-oriented approach allows reducing the complexity of development and maintenance by unifying interaction processes.

The main principles of OOP

The main principles of object-oriented programming are:

Abstraction;

Encapsulation;

Inheritance;

Polymorphism.

The main principles of OOP

- Abstraction - a mechanism by which each class represents a unique set of data fields and a set of operations (methods) on this data.

- Encapsulation - a mechanism that allows using an object as a "black box". That is, the implementation details are hidden from the user, and an interface (a set of methods) is provided to the external code for interacting with the object.

The main principles of OOP

Inheritance - a language mechanism that allows creating a hierarchy of classes. In this case, subclasses include all the fields and methods of the superclass.

A subclass can override superclass methods and add its own. That is, it modifies and/or extends the functionality of the parent.

The main principles of OOP

Polymorphism - the ability of objects to behave differently during execution.

Polymorphism can be represented as a property of classes within an inheritance hierarchy to have identical methods that will work differently depending on the specific type of object.

Classes

Class - an abstract data type defined by the user, which represents a model of some object intended to perform a specific task.

A class can be seen as a collection of fields and methods for processing them.

Classes

Concrete variables of type "class" are called class instances or objects.

A class is ONLY used through a collection of public methods (sometimes called an interface). Implementation details of the methods for the class user are considered insignificant.

A non-static method of a class is a function called for a specific instance of the class, with access to all fields and methods. This function cannot be called independently (i.e., not for an instance of the class). Implicitly, these functions are passed a pointer to the current instance of the class.

Classes

Class declaration is the description of data fields included in the class and functions for their processing.

The class declaration starts with the keyword "class". Then, within curly braces, the types and names of class fields and method prototypes are specified.

Classes

```
class Complex
{
      double Re;
      double Im;
public:
      Complex();
      Complex(double, double);
      double GetRe();
      double GetIm();
      Complex& operator+(const Complex&);
      ~Complex();
};
```

Classes

Class fields can have any type, including classes, structures, unions, and enumerations declared up to the current moment. A class field can be a pointer to the same class.

Bit fields can also be included in a class, similar to structures.

Access specifiers

In addition to the public interface (i.e., methods accessible in user code), a class typically has a private internal part. Fields and methods in the private section of the class are not accessible from user code.

The private section usually contains fields, as well as auxiliary methods and methods that perform actions that may change in future versions, depend on the operating system or hardware, etc.

Access specifiers

Making class fields public is not recommended. Modifying class fields should be done through method calls to the interface.

This simplifies making changes to the code later. For example, reading or writing class fields may require additional operations.

Access specifiers

In C++, the following access specifiers for class fields and methods are supported:

- `private`;
- `protected`;
- `public`.

By default, all fields and methods of a class have private access mode, while for structures, it is public.

Access specifiers

Fields and methods declared with the private specifier are accessible only to methods within the same class.

Attempting to access fields or methods declared with the private keyword from external code will result in a compilation error.

Access specifiers

Fields and methods declared with the protected specifier are accessible to methods within the same class and to methods of derived classes (i.e., subclasses).

Attempting to access fields or methods declared with the protected keyword from external code will result in a compilation error.

Access specifiers

Fields and methods declared with the public specifier are accessible both within the methods of the class and its subclasses, as well as from external code.

Access specifiers

The effects of the keywords public, private, and protected apply to fields and methods described after the keyword until the end of the class or until a new access specifier is declared.

Access specifiers

```
class Some {
      int a;                          } private
      int GenerateNum();              }
public:
      Some();                         } public
      int GetA();                     }
private:
      void SetupValue(int Val);       } private
protected:
      int b;                          } protected
private:
      float c;                        } private
};
```

Pointer this

Class methods, unlike fields, exist in a single instance and are shared among all instances of the class.

For each non-static method of a class, a pointer to the current instance of the class (the address of the object to which the method will be applied) is implicitly passed. This address is accessible through the keyword 'this'.

The 'this' pointer cannot be changed.

Pointer this

```
class A {
    int a;
public:
    void SetA(int A) {
        this->a = A;
    }
};
```

Pointer this

```
A aObj;
aObj.SetA(7);
```

The invocation of the SetA method can be represented as a function call within the class namespace, where the first parameter is the address of the object to which the method will be applied (this):

```
A::SetA(&aObj, 7);
```

The mechanism for passing this parameter can vary.

The size of a class

The size of a class is at least the sum of the sizes of the types of its fields. The access specifier type of the fields does not affect the size of the class. Methods also do not affect the size of the class.

You can determine the size of a class using the sizeof() operator.

The size of a class

Fields of classes are stored in memory in the same way as fields of structures.

Therefore, the same considerations about alignment of field locations in memory apply to classes as they do to structures.

Accessing Class Fields and Methods

Accessing class fields and methods is done similarly to structures, using the . (dot) and -> (arrow) operators.

```
class A {
	int a;
public:
	void SetA(int A);
};

A a1;
a1.SetA(5);

A* a2 = new A;
a2->SetA(7); // ⇔(*a2).SetA(7);
```

Declaration and Implementation

The declaration of a class and global functions for working with it are placed in a header file (*.h).

The implementation of non-template classes and global non-template functions is done in source code files (*.cpp).

For class methods, it is necessary to additionally specify the name of the class to which they belong, as these methods are located in the namespace of the class.

Declaration and Implementation

MyString.h

```
class MyString {
      char* buffer;
      int length;
public:
      int Length() const;
      const char* CStr() const;
      MyString();
      MyString& operator = (const MyString& Str);
...
};
MyString operator+(const MyString& Str1, const MyString& Str2);
```

Declaration and Implementation

MyString.cpp

```
#include “MyString.h”
int MyString::Length() const {
        return this->length;
}

const char* MyString::CStr() const {
        return this->buffer;
}

MyString::MyString() { ... }

MyString& MyString::operator = (const MyString& Str) { ...}

MyString operator+(const MyString& Str1, const MyString& Str2) { ... }
```

Constructors

When creating an instance of a class, the following actions need to be performed:

- Allocate memory for the object.
- Initialize fields.

In some cases, additional initializing actions may also be necessary.

Constructors

Для выполнения инициализации экземпляров класса в C++ имеются специальные методы, называемые конструкторами.

Данные методы вызываются компилятором автоматически при создании экземпляров класса.

Constructors

The constructor's name must match the class name.

A constructor does not have a return value. Additionally, when declaring and defining a constructor, no return type is specified, not even void.

A constructor cannot be called using the . or -> operators similarly to methods.

Constructors

Types of constructors:

1. Default constructor
2. Constructor with parameters
3. Copy constructor
4. Move constructor

Constructors

```
class MyString {
    char* buffer;
    int length;
public:
    MyString();                    //Конструктор по умолчанию
    MyString(const char* Str);   //Конструктор с параметрами
    MyString(const MyString& Str); //Конструктор копирования
    MyString(MyString&& TmpStr); //Move-конструктор
    ~MyString();                   //Деструктор
};
```

Constructors

Default constructor:

The default constructor has no parameters. This constructor is called when an instance of the class is created without explicitly assigning it a value.

The compiler automatically creates this constructor if no other types of constructors are declared in the class. Such a constructor can call the constructors of built-in classes and the constructors of the base class.

Constructors

```
MyString::MyString() {
    this->length = 0;
    this->buffer = new char[1];
    this->buffer[0] = '\0';
}
...
MyString str;                    str = ""
MyString str2 = MyString();      str2 = ""
```

Constructors

Constructor with parameters:

A constructor that has one or more parameters is called a constructor with parameters.

The number of arguments in the constructor with parameters is determined by the programmer based on the task.

Constructors with parameters are not automatically generated by the compiler.

A class may not have constructors with parameters.

The number of constructors with parameters can be any.

Constructors

```
MyString::MyString(const char* Str){
     this->length = strlen(Str);
     this->buffer = new char[this->length + 1];
     memset(this->buffer, 0, this->length + 1);
     strcpy(this->buffer, Str);
}
...
MyString str("Str 1");                    str  = Str 1
MyString str2 = MyString("Str 2");        str2 = Str 2
MyString str3 = "Str 3";                  str3 = Str 3
```

Constructors

If a constructor with parameters has only one argument, it can be called implicitly.

```
MyString str3 = “Str 3”;
```

str3 = Str 3

To prevent such implicit conversions, the keyword 'explicit' must be used when declaring the constructor:

```
explicit MyString(const char* Str);
...
MyString str3 = “Str 3”; //CE
```

Constructors

Copy Constructor:

The copy constructor is called when initializing a new object with an already created one.

If the copy constructor is not implemented by the programmer, the compiler automatically generates it. In this case, a bitwise copy of the class fields is performed.

Such behavior may be incorrect for a class. For example, if the class contains pointers to dynamically allocated memory, file descriptors, etc. In such cases, it is necessary to implement a custom copy constructor.

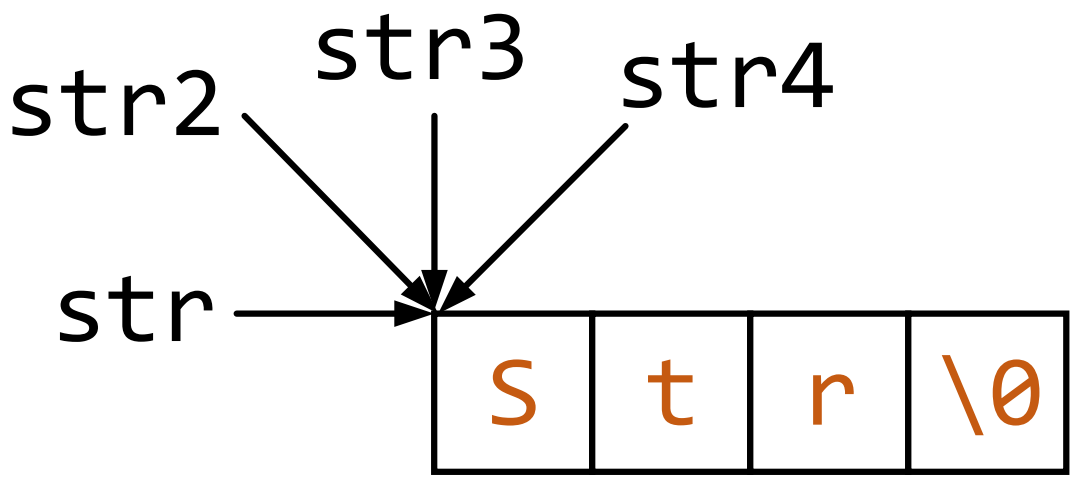
Since the copy constructor is part of the class, it also has direct access to the fields of the object being copied.

Constructors

```
MyString::MyString(const MyString& Str){
      this->length = Str.length;
      this->buffer = new char[this->length + 1];
      memset(this->buffer, 0, this->length + 1);
      strcpy(this->buffer, Str.buffer);
}
...
MyString str = MyString(“Str”);
MyString str2 = str;                  //Вызов конструктора копирования
MyString str3(str);                   //Вызов конструктора копирования
MyString str4 = MyString(str);  //Вызов конструктора копирования
```

Constructors

If the copy constructor is not implemented, then str2, str3, and str4 will point to the memory region allocated in the constructor of str.



Constructors

If memory is deallocated in the destructor, then when one of these strings is deleted, the remaining ones will contain the address of the deallocated memory region. Actions with this memory region can lead to a runtime error.

Constructors

Move Constructor:

When passing objects, especially anonymous ones, by value in functions and returning objects from functions by value, the copy constructor will be called. This can be a costly operation, especially considering that the created anonymous object will soon be destroyed.

These costs can be avoided with the move constructor.

Constructors

```
MyString::MyString(MyString&& TmpStr){
    // Data is transferred to the new object (complete
data copying is not performed)
    this->length = TmpStr.length;
    this->buffer = TmpStr.buffer;

    // We nullify the data so that when the destructor is
called for the temporary object and memory is released, the
data from the new object is not deleted
    TmpStr.buffer = nullptr;
    TmpStr.length = 0;
}
```

Constructors

```
MyString str = MyString(“Hello, ”);
MyString str2 = MyString(“World!”);

//MyString operator+(const MyString& Str);
MyString str3 = str + str2;
```

Anonimous object

For the initialization of str3, the move constructor will be called.

Constructors

Order of Constructor Calls:

If the fields of class A are other classes (for example, B and C), then before the body of the constructor A is executed, the default constructors for the member classes B and C will be called in the order of declaration of the fields in class A.

Constructors

```
class Test {
    MyString str;
    Complex c;
public:
    Test(){ };
};
...
Test t;
```

Order:

1) MyString()
2) Complex()
3) Test()

Constructors

If it is necessary to use constructors with parameters for member classes, then after the name of the constructor and before implementing its body, you need to specify an initialization list.

For this, the colon operator is used, after which the necessary constructors for member fields are listed separated by commas.

The order of specifying constructors for member fields in the initialization list does not matter, as the fields will be initialized in the order of declaration in the class.

Constructors

```
Test::Test() : str(“Default”), c(1,1)
{
}
...
Test t;
```

Order:

1) MyString(const char*)
2) Complex(double, double)
3) Test()

Constructors

If you don't use a field initialization list and perform assignments in the body of the constructor, then the default constructors for member fields will be called first, and then these fields will be assigned new values using the assignment operator.

Constructors

```
Test::Test() {
     this->str = MyString(“Default”);
     this->c = Complex(1, 1);
}
...
Test t;
```

Order:
1) MyString()
2) Complex()
3) MyString(const char*)
4) MyString::operator =
5) ~MyString
6) Complex(double, double)
7) Complex::operator =
8) ~Complex

Constructors

Since assigning values to fields of class types in the body of the constructor entails additional overhead in the form of additional function calls, it is preferable to use initialization lists rather than assignments in the constructor body.

Constructors

Initialization lists with the enumeration of field values in curly braces {}, similar to arrays and structures, can only be used with classes that do not have:

- private and protected fields;
- constructors;
- parent classes;
- virtual functions.

Destructor

When an instance of a class is destroyed, for example, when it goes out of scope or when the delete operator is called, a special method is automatically invoked to perform the cleanup work for the resources used by the class instance.

This method is called the destructor.

Destructor

Деструктор, как и конструкторы, не имеет возвращаемого значения, даже void.

Деструктор не принимает аргументов.

Имя деструктора должно совпадать с именем класса и предваряться символом ~.

Класс может иметь только один деструктор.

Destructor

```
MyString::~MyString()
{
    delete[] this->buffer;
    this->buffer = nullptr;
    this->length = 0;
}
```

Destructor

Unlike constructors, the destructor can be called similarly to class methods using the . or -> operators. However, it is not recommended to call the destructor directly.

```
MyString str = MyString("Hello, World");
str.~MyString();
```

After the destructor is called, attempting to access the object will result in a runtime error.

Destructor

If the destructor is not implemented by the programmer, it will be created by the compiler.

Such a destructor does not release any resources (e.g., it does not deallocate dynamically allocated memory), but only calls destructors for fields of class type, if they exist, in the reverse order of declaration of the fields in the class.

When a class enters an inheritance hierarchy, destructors of base classes are also called.

Destructor

```
class Test {
    MyString str;
    Complex c;
public:
    Test() : str("Default"), c(1,1) { };
};
...
Test t;
```

Order:
1) ~Complex()
2) ~MyString()

Creating objects in dynamic memory

To create an object in dynamic memory, the new operator is used. In this case, the constructor of the object specified by the programmer will be called.

```
MyString* dStr = new MyString;    //конструктор
                                  по умолчанию
MyString* dStr2 = new MyString(“Dynamic”);
```

Creating objects in dynamic memory

When using the malloc function, only a region in dynamic memory is allocated. The constructor is not called in this case.

Creating objects in dynamic memory

To destroy an object created in dynamic memory, the delete operator is used. In this case, the destructor of the class will be automatically called.

```
delete dStr;
delete dStr2;
```

Creating objects in dynamic memory

When using the free function, a region of dynamic memory is deallocated. The destructor is not called in this case.

Creating objects in dynamic memory

When deallocating an array of objects created in dynamic memory, it is important not to forget to specify [] after the delete operator.

Because in the absence of [], only the destructor for the first object in the array will be called. Additionally, this may lead to a runtime error.

Creating objects in dynamic memory

```
MyString* strArr = new MyString[3]{MyString("Abc"),
                                   MyString("Def")};
...
delete[] strArr;
```

```
strArr[0] = "Abc"
strArr[1] = "Def"
strArr[2] = ""
```

Constant Methods

If a method or operator logically should not modify the internal state of an object, it is recommended to make such a method const. To do this, after the method's arguments, the const keyword should be specified.

In a const method, only const methods of the class can be called. Calling global functions and working with local variables is also allowed.

Constant Methods

```
class A {
    int val1;

public:
    ...
    int get() const {
        int val = foo(3);
        return val1;
    }
};
```

A.h

```
class A {
    int val1;

public:
    ...
    int get() const;
```

A.cpp

```
int A::get() const
{
    …
}
```

Constant Methods

An attempt to modify a field in a constant method will result in a compilation error.

```
int A::get() const {
    val1++;              //CE
    return val1;
}
```

This allows preventing field modifications in methods that should only read class fields, rather than modify them, at compile time.

Constant Methods

In a constant method, you can modify fields declared with the static keyword, which are shared among all instances of the class.

Operator Overloading

For simplification and improvement of code readability, operators in C++ can be overloaded for a class.

When overloading an operator, you need to specify the keyword "operator" followed by the operator symbol to be overloaded. Similar to other class methods and global functions, you also need to specify the return type and the list of arguments.

Operator Overloading

Some built-in operators are equivalent to a combination of other built-in operators.

For example, a++ is equivalent to a = a + 1, which is also equivalent to a += 1.

This relationship between operators is not preserved when overloading them, unless the programmer specifically addresses it.

Operator Overloading

Operators can be overloaded both as class methods and global functions.

In the case of a global function, it does not have access to the private and protected fields and methods of the class. Therefore, it interacts with the class through the public interface.

When overloading as a class method, the this pointer is the left parameter of the expression.

Operator Overloading

Operators can be unary or binary.

A unary operator performs an action on one object, while a binary operator operates on two.

In C and C++, there also exists the ternary operator, but it is not allowed to be overloaded.

Operator Overloading

Overloading a unary operator can be done:

- As a class method with no parameters.
- As a global function with one parameter.

Operator Overloading

Overloading a binary operator can be done:

- As a class method with one parameter.
- As a global function with two parameters.

Operator Overloading

Overloading a binary operator as a class method:

A @ B

The left operand is the object A, for which the operator @ is called. The parameter of the operator @ will be the object B.

Operator Overloading

An operator can only be declared with syntax existing for it in the language's grammar. For example, it's not possible to declare a unary operator / or %.

Additionally, it's forbidden to define a new operator token, for instance **.

Operator Overloading

Operators forbidden for overloading:

```
.       (member access)
.*      (pointer-to-member access)
::      (scope resolution)
?:      (ternary conditional)
#       (preprocessor symbol)
##  (preprocessor symbol)
sizeof
typeid (runtime type identification)
```

Operator Overloading

Not all operators can be overloaded using a global function.

Operators that can only be overloaded by a class method:

=

[]

->

() (type conversion)

() (function call)

Operator Overloading

Ways to invoke an operator::

```
MyString str1(“Hello, ”), str2(“World!”);
MyString str3 = str1 + str2;
```

If the operator is overloaded by a class method, it can be invoked as:

```
MyString str3 = str1.operator+(str2);
```

If overloaded by a global function:

```
MyString str3 = operator+(str1, str2);
```

Operator Overloading

Operator overloading:

In some cases, if the assignment operator is not defined, the compiler will generate its implementation. Such an implementation performs a shallow copy of the object's fields. Therefore, in most cases, especially when the class contains pointers, it's necessary to implement a custom assignment operator.

If you need a translation into another language or more information, feel free to ask!

Operator Overloading

Assignment operator is invoked when assigning a new value to an existing object.

Copy constructor is invoked when initializing a new object with the value of an existing one.

Operator Overloading

```
MyString str = MyString(“Abc”);
MyString str2 = str;                    //конструктор копирования

MyString str3 = MyString(“Def”);
str3 = str;                             //оператор присваивания
str3 = MyString(“Qwerty”);              //оператор присваивания
```

Operator Overloading

```
MyString& MyString::operator = (const MyString& Str) {
     if (this != &Str) {
          delete[] this->buffer;
          this->length = Str.length;
          this->buffer = new char[this->length + 1];
          memset(this->buffer, 0, this->length + 1);
          strcpy(this->buffer, Str.buffer);
     }
     return *this;
}
```

Operator Overloading

The assignment operator, similar to the copy constructor, can have a move version.

```
MyString& MyString::operator = (MyString&& Str){
	if (this != &Str) {
		delete[] this->buffer;
		this->length = Str.length;
		this->buffer = Str.buffer;
		Str.buffer = nullptr;
		Str.length = 0;
	}
	return *this;
}
```

Operator Overloading

```
MyString str = MyString(“Abc”);
MyString str2 = MyString(“Def”);

MyString str3 = MyString(“Qwerty”);
str3 = str + str2;     //move-присваивание
```

Operator Overloading

String concatenation:

The + and += operators can be overloaded for string concatenation.

The A += B operator appends the string B to the current string A.

The + operator creates a new string, which is the concatenation of A and B.

Operator Overloading

Overloading the += method as a class method:

```
MyString& MyString::operator+=(const MyString& Str) {
    this->length += Str.length;
    char* tmpBuffer = new char[this->length + 1];
    memset(tmpBuffer, 0, this->length + 1);
    strcpy(tmpBuffer, this->buffer);
    strcat(tmpBuffer, Str.buffer);
    delete[] this->buffer;
    this->buffer = tmpBuffer;
    return *this;
}
```

Operator Overloading

Overloading the + operator as a class method:

```
MyString MyString::operator+(const MyString& Str){
    MyString res(*this);
    res += Str;
    return res;
}
```

Operator Overloading

Overloading the + operator as a global function:

```
MyString operator+(const MyString& Str1,
                   const MyString& Str2) {
    MyString res(Str1);
    res += Str2;
    return res;
}
```

Operator Overloading

Since when overloading as a class method, the this pointer is the left argument in the expression defined by the operator, such an operator can only be used when the left argument of the expression is an instance of the class.

Operator Overloading

For example, concatenating a string with an integer:

```
MyString operator+(int N);
```

This operator will handle the expression:

```
MyString str = MyString("Qwerty");
MyString str2 = str + 1;  //str2 = Qwerty1
```

Operator Overloading

However, it won't be able to handle the expression:

```
str2 = 1 + str;
```

Because the left operand has the type int, not MyString.

To handle this situation, you need to overload the + operator as a global function:

```
MyString operator+(int N, const MyString& Str);
```

Operator Overloading

Index access operator:

This operator can only be overloaded as a class method.

```
char& MyString::operator[](int N)
{
    return this->buffer[N];
}
```

Operator Overloading

```
MyString str = MyString("Abc");

char ch = str[0];          ch = 'A'
str[1] = 'Z';              str = "AZc"
```

Operator Overloading

Overloading the unary operator -:

Let this operator return a string whose characters are arranged in reverse order.

Перегрузка операторов

Overloading the unary operator - as a class method:

```
MyString MyString::operator-(){
      char* resBuffer = new char[this->length + 1];
      memset(resBuffer, 0, this->length + 1);
      for (int i = 0; i < this->length; i++)
            resBuffer[i] =
                  this->buffer[this->length - 1 - i];
      MyString res = MyString(resBuffer);
      delete[] resBuffer;
      return res;
}
```

Перегрузка операторов

Overloading the unary operator - as a global function:

```
MyString operator-(const MyString& Str){
      char* resBuffer = new char[Str.Length() + 1];
      memset(resBuffer, 0, Str.Length() + 1);
      for (int i = 0; i < Str.Length(); i++)
            resBuffer[i] =
                  Str.CStr()[Str.Length() - 1 - i];
      MyString res = MyString(resBuffer);
      delete[] resBuffer;
      return res;
}
```

Operator Overloading

Increment (++) and decrement (--) operators have both prefix and postfix forms.

Therefore, in order for the compiler to distinguish between them, their prototypes must differ.

Operator Overloading

Prefix form (when overloaded as a class method) has no parameters

```
MyString& operator--();
```

In the postfix form, there is one dummy argument of type int. The compiler usually passes 0 there:

```
MyString& operator--(int);
```

Operator Overloading

Let the prefix form of the operator remove the first letter in the string, and the postfix form remove the last one.

Operator Overloading

```
MyString& MyString::operator--() {
    char* resBuffer = new char[this->length];
    memset(resBuffer, 0, this->length);
    memcpy(resBuffer, this->buffer + 1,
                          this->length - 1);
    delete[] this->buffer;
    this->buffer = resBuffer;
    this->length--;
    return *this;
}
```

Operator Overloading

```
MyString& MyString::operator--(int) {
    char* resBuffer = new char[this->length];
    memset(resBuffer, 0, this->length);
    memcpy(resBuffer, this->buffer,
                      this->length - 1);
    delete[] this->buffer;
    this->buffer = resBuffer;
    this->length--;
    return *this;
}
```

Operator Overloading

```
MyString str = MyString("Qwerty");
--str;                          str = "werty"
str--;                          str = "wert"
```

Operator Overloading

Type Conversion Operator Overloading:

The type conversion operator can be overloaded. In this case, the return type of the method is not specified.

The type conversion operator must be a method of the class.

Operator Overloading

```
MyString::operator const char*() const
{
    return this->buffer;
}

MyString::operator int()
{
    return this->length;
}
```

Operator Overloading

```
MyString str = MyString("Qwerty");

const char* s = (const char*)str; s = "Qwerty"
int n = (int)str;                 n = 6
```

Operator Overloading

Function Call Operator Overloading:

Objects in which the function call operator () is overloaded are called function objects. That is, such an object can behave like a function.

The function call operator can only be overloaded as a method of the class.

Operator Overloading

- Let's overload the function call operator.
- Let it have the following prototype:

```
MyString operator() (const char* Str, int Pos);
```

Here, Str is a C-style string that will be inserted into the current string at the position indicated by Pos. The method returns a new object of type MyString and does not modify the object to which it is applied.

Operator Overloading

```
MyString MyString::operator() (const char* Str, int Pos) {
     char* resBuffer=new char[this->length+strlen(Str)+1];
     memset(resBuffer, 0, this->length + strlen(Str) + 1);
     for (int i = 0; i < Pos; i++)
          resBuffer[i] = this->buffer[i];
     strcat(resBuffer, Str);
     strcat(resBuffer, this->buffer + Pos);
     MyString res = MyString(resBuffer);
     delete[] resBuffer;
     return res;
}
```

Operator Overloading

```
MyString str = MyString(“Qwerty”);
MyString str2 = str(“ABC”, 2);
```

Result:

str = “Qwerty”

str2 = “QwABCerty”

Operator Overloading

Operator Overloading->:

```
class A {
private:
      unsigned int count;
      int a;
public:
      A(): count(0), a(0) { }
      void set(int val) { this->a = val; }
      int get() { return this->a; }

      A* operator->() { this->count++; return this;}

      void counter() { std::cout << “count = ” << this->count <<
                       std::endl; }
};
```

Operator Overloading

```
A a;
a.counter();     //count = 0
a->set(3);
std::cout << “a = ” << a->get() << std::endl;
a.counter();     //count = 2
a.set(4);
std::cout << a.get() << std::endl;
a.counter();     //count = 2, т.к. оператор -> не
                        вызывался
a->counter();    //count = 3
```

Operator Overloading

The dereference operator.

```
class A {
    int val;

public:
    A(int Val){ this->val = Val; }
    int operator *() { return val; }
};
```

Operator Overloading

```
A a(777);
int res = *a;        //res = 777


A* aaa = new A(5);
res = **aaa;         //res = 5
```

Operator Overloading

Usually, the -> and * (dereference) operators are used in the implementation of smart pointers, as well as iterators.

Operator Overloading

The new and delete operators for dynamic memory allocation can also be overloaded.

For a single object:

```
void* operator new(size_t);
void operator delete(void*, size_t);
```

For an array of objects:

```
void* operator new[](size_t);
void operator delete[](void*, size_t);
```

Operator Overloading

The new and delete operators are static, so they do not receive a this pointer.

Overloading the memory allocation and deallocation operators is necessary, for example, when you need to allocate objects in specific memory areas or perform additional actions during memory allocation.

Prohibitions of certain operations for a class

Sometimes, depending on the problem being solved, it is necessary to prohibit certain operations for instances of a particular class. For example, prohibiting copying, prohibiting placement in dynamic or static memory, prohibiting assignment, and so on.

To do this, you need to place the corresponding method or constructor in the private or protected section.

Prohibitions of certain operations for a class

Prohibition of copying and assignment :

```
class NoCopy {
    NoCopy(const NoCopy& NC);
    NoCopy& operator=(const NoCopy& NC);
public:
    NoCopy(int Val);
};
```

Prohibitions of certain operations for a class

Prohibition of dynamic memory allocation

```
class NoHeap {
    void* operator new(size_t);
    void operator delete(void*);
    void* operator new[](size_t);
    void operator delete[](void*);
public:
    NoHeap(int Val);
};
```

Prohibitions of certain operations for a class

Prohibition of creating a static object :

```
class NoStatic {
    ~NoStatic();
public:
    void destroy() { delete this; }
};
```

Prohibitions of certain operations for a class

When the object leaves the scope, the destructor will be implicitly called. For an instance of the class `NoStatic`, this will lead to a compilation error because the destructor is declared as private.

```
void Fun() {
    NoStatic ns;    //CE
    ...
}
```

Friend functions and classes

A friend function is a global function that can access the private and protected fields and methods of a class in which it is declared.

When declaring a friend function, the keyword "friend" must be specified.

Friend functions and classes

It doesn't matter in which section (private, protected, or public) a friend function is declared because it is not a member of the class.

Friend functions and classes

```
class MyString {
      char* buffer;
      int length;
public:
      MyString();
      MyString(const char* Str);
      MyString(const MyString& Str);
      MyString(MyString&& TmpStr);
      ~MyString();
      friend std::istream& operator>>(std::istream& In,
                                      MyString& Str);
};
```

Friend functions and classes

```
std::istream& operator>>(std::istream& In, MyString& Str){
     std::string res;
     In >> res;
     Str.length = res.length();
     delete[] Str.buffer;
     Str.buffer = new char[Str.length + 1];
     memset(Str.buffer, 0, Str.length + 1);
     strcpy(Str.buffer, res.c_str());
     return In;
}
```

Friend functions and classes

```
MyString str(“Qwerty”);
std::cin >> str;
```

Friend functions and classes

In the current architecture of the `MyString` class, the output stream insertion operator `<<` can be overloaded as a regular global function (not a friend).

```
std::ostream& operator<<(std::ostream& Out,
                         MyString& Str) {
    Out << Str.CStr();
    return Out;
}
```

Friend functions and classes

```
MyString str;
std::cin >> str;
std::cout << “str = ” << str << std::endl;
```

Friend functions and classes

You can make all methods of one class accessible to another by declaring it with the `friend` keyword.

```
class A {
    ...
    friend class B;
};
```

All methods of class B have access to all fields and methods of class A, including protected ones.

Friend functions and classes

The use of friend classes and functions often indicates a poorly designed class architecture.

Static fields and methods

Static fields are fields that are common to all instances of a class.

A static field is declared with the keyword `static`.

```
class A {
    int a;
public:
    static int count;
    ...
};
```

Static fields and methods

Properties of static fields:

- Static fields are placed in the static memory area during compilation (even before any instances of the class are created), regardless of where the object itself is created.
- They exist in a single instance regardless of how many objects of the class are created, including before the creation of the first object.
- Static fields need to be defined as global variables, specifying the class name to which they belong. This definition should not be inside functions or class methods.

```
int A::count;
int A::count = 5;
```

- If a static field has a public access specifier, it can be accessed from outside the class using the class name:

```
int cnt = A::count;
```

- If a static field has a public access specifier, it can also be accessed from outside the class using the object name:

```
int cnt2 = aObj.count;
```

Static fields and methods

- Accessing a static field inside non-static methods of the class can also be done in the same way as accessing a non-static field:

```
int cnt = this->count;
```

- Accessing a static field inside both static and non-static methods can be done using the class name:

```
int cnt = A::count;
```

- A static field can be modified in a const method.

Static fields and methods

Since static fields exist in a single instance for all instances of the class and are located in static memory, they do not affect the size of the class.

The size of the class is only influenced by non-static fields (and their alignment).

Static fields and methods

A class method can also be declared with the static keyword.

Static methods are essentially global functions declared within the namespace of the class.

Static fields and methods

A.h

```
class A {
      int a;
      static int count;
public:
      static int GetCount();
      ...
};
```

A.cpp

```
int A::GetCount() {
      return A::count;
}
```

Static fields and methods

Properties of static methods:

- Static methods do not receive a this pointer. Consequently, non-static fields are not accessible within such methods.
- Static methods can be called independently of any object:

```
int res = A::GetCount();
```

- Since a static method is a method of the class, it can also be called for a specific object:

```
A a;
a.GetCount();
```

- Static methods are accessible within the code of non-static methods..

Static fields and methods

Since static methods do not receive a this pointer, constructors and destructors cannot be static.

Pointers to class methods

Just like regular functions, class methods can be accessed through pointers.

Since non-static class methods implicitly receive a pointer to the current object (this), declaring a pointer to such a method differs from declaring a pointer to a global function.

Therefore, in addition to specifying the return type, type, and number of arguments, you need to specify the class name.

Pointers to class methods

Since static methods are essentially global functions and they do not have a this pointer, pointers to static methods are defined in the same way as pointers to regular global functions.

Pointers to class methods

```
class Calc {
    int A;
    static int StA;
public:
    Calc(int A) { this->A = A; }
    int Sum(int B) { return this->A + B; }
    int Mul(int B) { return this->A * B; }
    static int StMul(int B){return Calc::StA*B;}
};
```

Pointers to class methods

```
int Calc::StA = 3;
int main() {
      Calc c(5);
      int (Calc::*Fun)(int) = &Calc::Mul;
      int res = (c.*Fun)(7);                          res = 35

      Fun = &Calc::Sum;
      int res2 = (c.*Fun)(7);                         res2 = 12

      int(*FunStatic)(int) = Calc::StMul;
      int res3 = FunStatic(8);                        res3 = 24
}
```

Template classes

Classes can contain fields whose types are defined by the programmer for each instance of the class.

The declaration of a template class is preceded by the keywords template<typename T1, ... typename TN>, with the required number of template parameters.

Template classes

Since the methods of a template class are created only after the specification of the class instance for specific template parameter values, the implementation of template classes cannot be performed in source code files (*.cpp).

Template classes

Usually, the class declaration, like for regular non-template classes, is done in the header file.

The implementation of a template class is performed in a *.hpp file. This file should be located in the "Header Files" section.

When using template classes, the corresponding *.hpp file is included in the project.

Template classes

Complex.h

```
template<typename T>
class Complex {
      T Re;
      T Im;
public:
      Complex();
      Complex(T R, T I);
      ~Complex();
      T GetRe();
      T GetIm();
      template<typename T>
      friend std::ostream& operator<<(std::ostream& Out, Complex<T>& C);
};
```

Template classes

Complex.hpp

```
#include “Complex.h”

template<typename T> Complex<T>::Complex(){
     this->Re = 0;
     this->Im = 0;
}

template<typename T> Complex<T>::Complex(T Re, T Im){
     this->Re = Re;
     this->Im = Im;
}
```

Template classes

Complex.hpp

```
template<typename T> Complex<T>::~Complex(){ }

template<typename T> T Complex<T>::GetRe() {
	return this->Re;
}

template<typename T> T Complex<T>::GetIm() {
	return this->Im;
}

template<typename T> std::ostream& operator<<(std::ostream& Out, Complex<T>& C) {
	Out << C.Re << “+i” << C.Im;
	return Out;
}
```

Template classes

main.cpp

```
#include <iostream>
#include “Complex.hpp”

int main()
{
    Complex<int> c(5, 7);
    int re = c.GetRe();                    re = 5
    int im = c.GetIm();                    im = 7
    std::cout << c << std::endl;           5 + i7
    return 0;
}
```