

Q1.

The history of OOP:

The earliest program designs were written in machine language, directly using binary code to represent instructions and data that machines could recognize and execute. Simply put, it means directly writing sequences of 0 and 1 to represent the programming language.

Due to the difficulty of writing machine language, assembly language was developed. Assembly language, also known as symbolic language, uses mnemonics to replace the opcode of machine instructions, and uses address symbols or labels to replace the address of instructions or operands. Assembly language, due to its use of mnemonic symbols to write programs, is more convenient than binary code programming in machine language, which to some extent simplifies the programming process.

The development of C++:

The development history of C++ can be traced back to 1979, when Bjarne Stroustrup, who was working at Bell Labs, began designing C++, initially known as "C with Classes". This name reflects the fact that C++ initially added object-oriented programming features on top of the C language. In 1983, C++ was officially renamed C++, marking the independent development of this language. In the following years, C++ gradually became popular.

In 1985, the first official C++ compiler was released. At this point, C++ began to be widely used in the field of software development because it combines the efficiency of C language with the advantages of object-oriented programming. In 1989, the first C++ standard (C++98) was released, which defined the basic features and standard libraries of C++ and laid the foundation for its development.

The C++03 standard was released in 2003, which corrected some issues and added new features such as new conversion rules and template specialization. The C++03 standard makes the C++ language more comprehensive and standardized.

In 2011 was an important milestone in the development history of C++, and the release of the C++11 standard introduced many important new features, completely changing the programming paradigm of C++. C++11 introduces features such as automatic type inference, smart pointers, lambda expressions, and right value references, greatly improving the programming efficiency and expressive power of C++. In addition, C++11 also introduces features such as multithreading support and concurrent programming, making C++ more convenient and efficient in handling concurrent tasks.

In 2014, the C++14 standard was released, continuing to improve and expand language features. The C++14 standard has made some improvements on the basis of C++11, adding new features such as generic lambda expressions, return type derivation, binary literals, etc., further enhancing the expressive power and programming efficiency of C++.

In 2017, the C++17 standard was released, further enriching the functionality and features of C++. C++17 has introduced some new features, such as structured binding, collapsed expressions, inline variables, etc., further improving the C++ language. The C++17 standard has also fixed some issues in the C++14 standard, improving the stability

and performance of C++.

The latest C++ standard is C++20, released in 2020. C++20 has introduced many new features, such as concepts, coroutines, and scope based for loops, further enhancing the modernization and practicality of C++. The release of the C++20 standard marks that C++ is still constantly evolving and maintaining its position as an important programming language.

Q2.

Abstract:

The definition of abstraction on Wikipedia is as follows:

Conceptual abstractions may be formed by filtering the information content of a concept or an observable phenomenon, selecting only the aspects that are relevant for a specific objective valued purpose. In object-oriented programming, the abstract process is manifested as a pruning process, where different and not essential features are all pruned out, that is, extracting the common features of things is extracting the essential features of things, discarding different features

For example:

In the context of shopping mall transactions, the common feature of items such as water, meat, clothing, hats, etc. is that they have a price. We ignore their different usage methods and come up with the concept of merchandise, which is an abstract process

Encapsulation:

Encapsulation is the collection of data or functions into individual units (which we refer to as classes). Encapsulated objects are often referred to as abstract data types.

The purpose of encapsulation is to protect or prevent code (data) from being unintentionally destroyed by us. In object-oriented programming, data is viewed as a central element and closely integrated with the functions that use it to protect it from accidental modifications by other functions.

Encapsulation provides an effective way to protect data from accidental damage. Compared to defining data (implemented using domains) as public in a program, defining them as private would be better in many ways. Private data can be indirectly controlled in two ways. The first method is to use traditional storage and retrieval methods. The second method we use is properties, which not only control the legality of accessing data, but also provide flexible operation methods such as read write, read-only, and write only.

For example:

```
#include<iostream>
#include<string>
using namespace std;
class person
{
private:
    int age;
    int phone;
    string name;
    string IDcard;
public:
    void change_phone(int new_phone)
    {
        phone = new_phone;
    }
    person(int a, int b, string s, string ID)
    {
        age = a;
        phone = b;
        name = s;
        IDcard = ID;
    }
};

int main()
{
    person A(18, 166786799, "mike", "344560200401778756");
    A.change_phone(134786559);
    //A.phone = 134786559;
```

The diagram shows a C++ code snippet for a `person` class. Red circles and arrows highlight key features of encapsulation:

- A red circle around the `private:` section (containing `age`, `phone`, `name`, and `IDcard`) is pointed to by an arrow from the box: "Data and functions are encapsulated into the class **person**".
- A red circle around the `public:` section (containing the `change_phone` method) is pointed to by an arrow from the box: "External users can only access intra class elements through the interfaces provided in the class".
- A red circle around the `A.change_phone` call in the `main` function is pointed to by an arrow from the box: "Direct access outside of class is not allowed".
- A red circle around the commented-out line `//A.phone = 134786559;` is also pointed to by an arrow from the box: "Direct access outside of class is not allowed".

Q3.

information hiding

Many times, developers of different modules do not need to pay attention to the working principles and structures within other modules. They only need to focus on the input and output of these modules. Therefore, modules like this communicate with each other through their APIs, and one module does not need to know the internal situation of another module, which is called information hiding.

The main purpose of information hiding is to reduce the coupling between modules and improve the maintainability and scalability of software systems. Through information hiding, software modules can encapsulate internal data and implementation details, providing limited interfaces for external modules to access, making the system easier to understand, modify, and maintain. And information hiding can also improve the security of software. By restricting direct access to module internal data and methods, malicious operations and illegal access to the system can be prevented, thereby protecting the data

security and integrity of the system

implementation hiding refers to hiding the internal and implementation details of a class during class design, exposing only necessary interfaces and methods to the outside world, thereby achieving class encapsulation.

By implementing hiding, the internal implementation details of a class are invisible to external users, who can only interact with the class through public interfaces. So as to protect the internal data and implementation details of the class, prevent external users from directly accessing and modifying the internal state of the class, and improve the convenience of use.

Q4.

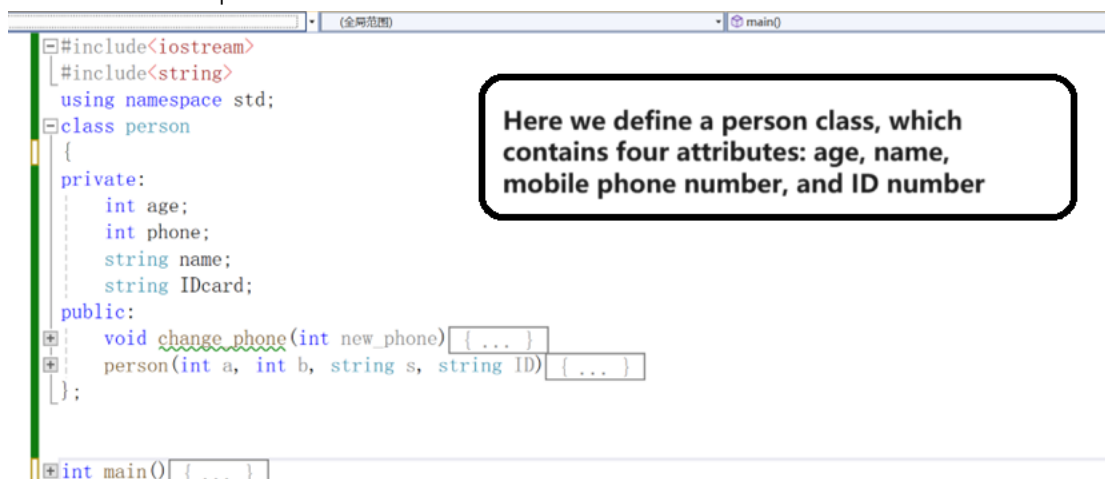
A **class** is a template or blueprint used to describe objects with similar properties and behaviors. The properties (member variables) and behaviors (methods) of objects are defined in the class, and concrete objects can be created by instantiating the class.

An **object** is the instantiation result of a class and is the fundamental unit in a program. An object is a concrete instance of a class, with properties and methods defined within the class.

Inheritance implements the reuse of similar code, allowing one class (called a subclass or derived class) to inherit the properties and methods of another class (called a parent or base class). Through inheritance, subclasses can obtain the characteristics of the parent class, including properties and methods, thereby achieving code reuse and extension.

Polymorphism is a concept that allows different objects to respond differently to the same message. Under the concept of polymorphism, a reference to a parent class can point to an object of a subclass, thereby achieving interchangeability between different classes.

for example:



```
#include<iostream>
#include<string>
using namespace std;
class person
{
private:
    int age;
    int phone;
    string name;
    string IDcard;
public:
    void change_phone(int new_phone) { ... }
    person(int a, int b, string s, string ID) { ... }
};

int main() { ... }
```

Here we define a person class, which contains four attributes: age, name, mobile phone number, and ID number

```

#include ...
using namespace std;
class person
{
private:
    int age;
    int phone;
    string name;
    string IDcard;
public:
    void change_phone(int new_phone) { ... }
    person(int a, int b, string s, string ID) { ... }
};

int main()
{
    person stu1(18, 166786799, "mike", "344560200401258756");
    person stu2(28, 18874799, "bob", "377854200503116730");
    person stu3(18, 16600979, "ff", "355067200501115781");
    stu1.change_phone(134786559);
    //A.phone = 134786559;
}

```

Here we instantiated three objects (three different students) using the defined person class

The screenshot shows a C++ project in Visual Studio. The code defines a base class `father` and a derived class `son1` using public inheritance. The `father` class has private attributes `age`, `phone`, and `bank_account`, and a protected attribute `name`. The `son1` class inherits from `father` and overrides the `display` method. The `main` function creates a `son1` object and calls its `display` method.

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4 class father
5 {
6 private:
7     string bank_account;
8 protected:
9     int phone;
10 public:
11     int age;
12     string name;
13     father(int a, int b, string s, string bank_account2) { ... }
14 };
15
16 class son1 : public father
17 {
18 public:
19     son1(int a, int b, string s, string bank_account2) : father(a, b, s, bank_account2) {}
20     void display()
21     {
22         cout << "name is : " << name << endl;
23         cout << "age is : " << age << endl;
24         cout << "phone is " << phone << endl;
25         cout << "bank_account can not be visited" << endl;
26     }
27 };
28
29 int main()
30 {
31     //father s(22, 999999, "mike", "345366546");
32     son1 s1(18, 12234134, "john", "");
33     s1.display();
34 }

```

The console output shows the following text:

```

name is :john
age is :18
phone is12234134
bank_account can not be visited

```

E:\Project10\x64\Debug\Project10.exe (进程 25516)
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“在调试停止时自动关闭控制台”。
按任意键关闭此窗口。 . . .

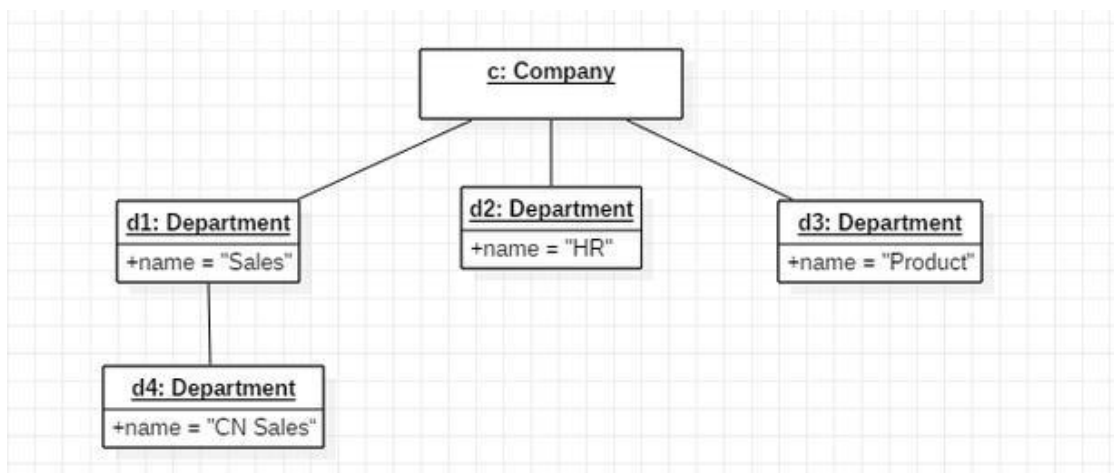
Son1 inherits the properties of the parent class through public inheritance

Q5:

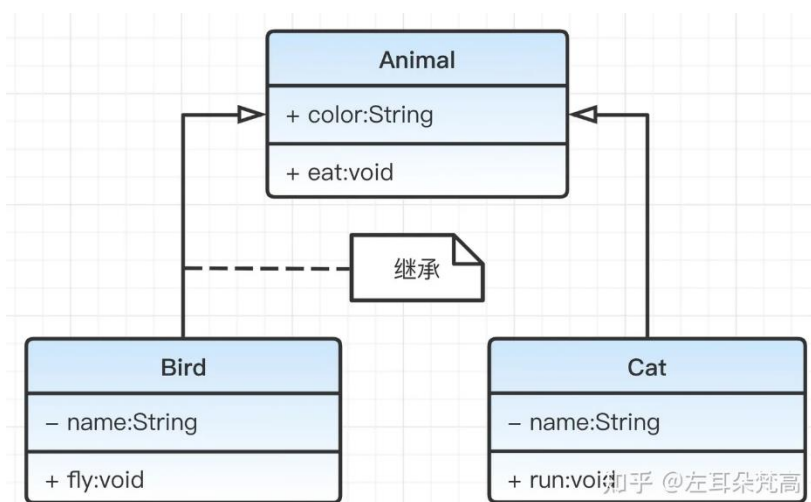
ClassDiagram:



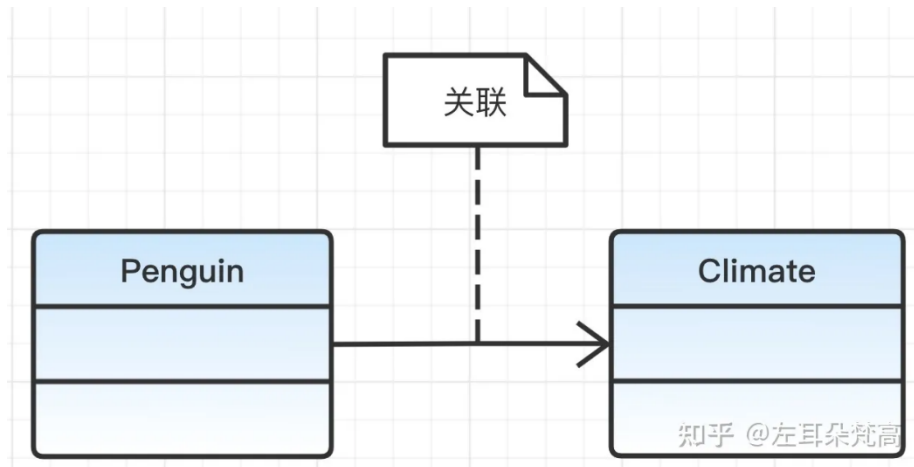
Object Diagram:



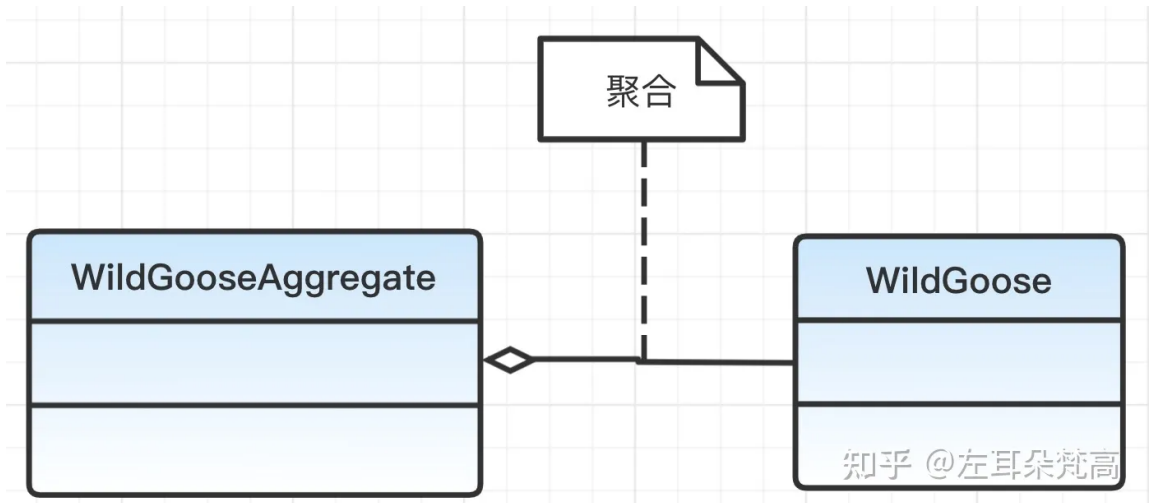
Inheritance:



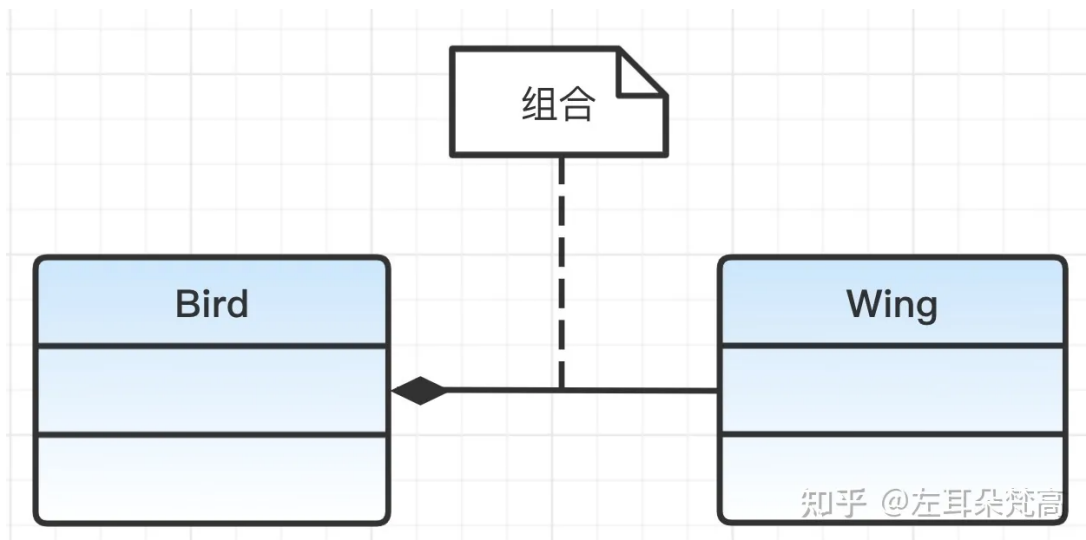
Association:



Aggregation:



Combination:



Q6:

There are many popular C++ development tools, such as **Visual Studio**, **CLion**, **Xcode**, **Eclipse CDT**, and so on. Here are their evaluations.

Visual Studio: Developed by Microsoft, Visual Studio is a comprehensive integrated development environment (IDE) that provides a wide range of features for C++ development. It offers advanced debugging capabilities, IntelliSense, code refactoring, and seamless integration with other Microsoft tools. Visual Studio is known for its user-friendly interface and strong community support.

CLion: CLion is a powerful cross-platform IDE developed by JetBrains specifically for C and C++ development. It offers intelligent code assistance, refactoring, and integration with the CMake build system. CLion is known for its strong support for modern C++ features and its seamless integration with other JetBrains tools.

Eclipse CDT: Eclipse CDT is an open-source IDE that provides a rich set of features for C and C++ development. It offers a customizable interface, support for various compilers, and integration with popular version control systems. Eclipse CDT is known for its extensibility through plugins and strong community support.

Code::Blocks: Code::Blocks is a free and open-source IDE that is highly customizable and easy to use. It supports multiple compilers, including GCC and Clang, and offers features like code completion, syntax highlighting, and project management. Code::Blocks is known for its simplicity and lightweight nature.

Xcode: Xcode is an IDE developed by Apple for macOS and iOS development, which also provides support for C++ development. It offers a range of tools for debugging, testing, and profiling C++ code. Xcode is known for its integration with Apple's ecosystem and its user-friendly interface.

As for the C++ development tool I adopt, I personally prefer using Visual Studio for C++ development due to its comprehensive features, strong debugging capabilities, and seamless integration with other Microsoft tools.

Visual Studio provides excellent support for C++ development, including modern C++ features and standards like C++11/14.

Q7:

Boolean Type: C++ introduces a boolean data type (`bool`) that can hold either true or false values. In C, the boolean type is typically represented using integers (0 for false, non-zero for true).

String Type: C++ introduces a string data type (`std::string`) that allows for easier manipulation of strings compared to C-style strings. In C, strings are typically represented as arrays of characters.

References: C++ introduces references, which are aliases to existing variables. References provide a convenient way to work with variables without the need for pointers. C does not have references.

Classes and Objects: C++ introduces the concept of classes and objects, allowing for object-oriented programming. C does not have built-in support for classes and objects.

Templates: C++ supports templates, which allow for generic programming. Templates enable the creation of generic functions and classes that can work with different data types. C does not have built-in support for templates.

Namespace: C++ introduces namespaces, which help organize code and prevent naming conflicts. C does not have namespaces.

Type Inference: C++11 introduced the `auto` keyword for type inference, allowing the compiler to deduce the data type based on the initialization expression. C does not have type inference.

Size of Data Types: The size of some data types may differ between C and C++. For example, the long double data type may have different sizes in C and C++ depending on the compiler and platform.

Q8:

a=12

b=3

c=45

d=634