**命名空间**

namespace A

{

}

using namespace std;//使用std命名空间

cerr//标准错误流

clog//标准日志流

**内联函数**

inline int add（）

减少函数开销，直接将函数插入

**foreach遍历：**

for （Student stu：stus）{

}

**getline(cin, str1)**

读取整行输入

**strcat（newstr，str）**

字符数组串接

**封装**

class note

{

    private:

        int age;//成员变量

        string name;

    public:

        int age;//局部变量

    // 初始化方法一

        note():age(),name(){};//带有初始化列表的构造方法

        note(int *age*,string *name*):age(age),name(name){}; //构造函数的重载

Teacher():ps(new Student("王五",22)){} //带有内存操作

    // 初始化方法二

        // 无参构造函数

        not()

        {

            name="";

            age=0;

        }

        // 有参构造函数

        note(int *age*,string *name*)

        {

            this->age=age;

            this->name=name;

        }

    // 公开函数

        void setAge(int *a*){

            this->age=age;

        }

        int getAge(){

            return age;

        }

    // 析构函数，清理对象

        ~Teacher(){

delete ps;

        }

};

Student\*stu = new Student("赵六",30);

**继承**

权限修饰（限制访问范围）

|  |  |  |  |
| --- | --- | --- | --- |
| 修饰符 | 本类里 | 子孙类 | 外部类 |
| **public** | √ | √ | √ |
| **protected** | √ | √ |  |
| **private** | √ |  |  |

class Animal

{

public:

    int m\_Age;

};

// 菱形继承

class Sheep:public Animal{};

class Tuo : public Animal{};

class SheepTuo : public Sheep, public Tuo{};

// 虚基类:继承只包含一个虚基类的子对象

// 消除二义性\节省空间

// 注意:虚基类构造函数只调用一次

class Sheep: virtual public Animal{};

class Tuo : virtual public Animal{};

class SheepTuo : public Sheep, public Tuo{};

int main(){

    SheepTuo st;

    st.Sheep::m\_Age=18;

    st.Tuo::m\_Age=28;

}

子类型在任何时候都可以当做父类型来用

**面向对象**

**内存分区模型**

图示

描述已自动生成

字符串常量，静态变量也在全局区

局部常量和变量都在栈区

int\* func(){

    int\* p=new int(10);

    return p;

}

// 不能返回局部变量引用

void myswep(int &*a*,int &*b*){

    int temp=*a*;

*a*=*b*;

*b*=temp;

}

int main(){

    int a=10;

    int b=20;

    myswep(a,b);

}

// 静态区可返回引用

int& test(){

    static int a = 10;

    return a;

}

// 常量引用

int main(){

    // int temp=10;const int &ref=temp;

    const int & ref=10;

}

// 申请空间

int main(){

    int\* p=new int(10);

    delete p;

    p=nullptr;

}

**静态**

类内声明，类外初始化

// 静态变量

// 静态函数只能通过类名访问

// 静态成员函数，只能访问静态成员变量

//1.类级别：静态函数属于类而不是类的实例，可以通过类名直接调用，而不需要创建对象。

//2.访问限制：静态函数只能访问静态成员变量和静态成员函数，不能访问非静态成员变量和非静态成员函数，因为它没有this指针。

//3.共享：所有类的实例共享同一个静态函数和静态成员变量。

//4.生命周期：静态函数在程序运行期间存在，直到程序结束。

//5. 定义位置：静态函数可以在类内部声明，但必须在类外部定义。

class Student{

public:

    static int age;

    static void study(){

        cout<<"hello"<<endl;

    }

};

int Student::age=10;

int main(){

    Student::study();

}

// 统计对象数量

class MyClass{

    public :

        static int count;

        MyClass(){

            count++;//在构造函数中自增

        }

};

int MyClass::count=0;

// 一个类之中公开的静态变量，相当于全局变量

class MyClass1{

    public :

        static const int count;

};

// 建立工具类

class MathUtils {

public:

    static int add(int *a*, int *b*) {

        return *a* + *b*;

    }

};

单例设计模式

保证一个类只有一个实例，并提供一个全局访问点来获取该实例

实现方式：类的构造函数是私有的

目的：限制类的实例化次数，确保在程序运行过程中只有一个实例存在，避免资源浪费和不一致的状态

饿汉式

//可以确保在程序运行期间只会创建一个实例，在类加载时就创建实例。但是它的缺点是在程序启动时就会创建实例，无论是否需要，可能会造成资源浪费。天然线程安全。

懒汉式

//在第一次调用时才创建实例。节省内存，只有在需要时才占用内存。需要额外处理以确保线程安全（如加锁）。

文本

描述已自动生成文本, 信件

描述已自动生成

**友元**

// 友元

// 声明友元，在building类中声明

friend void goodfriend();//全局函数做友元，访问building类所有属性

friend class goodfriend;//整个friend类做友元，友类可访问building

friend void goodfriend::visit1();//friend的成员函数做友元，访问building

// +运算符重载

// 成员函数

class Student{

    private:

    public:

        int chinese\_score;

        int math\_score;

    Student operator+(Student &*s*){

        Student temp;

        temp.chinese\_score=this->chinese\_score+*s*.chinese\_score;

        temp.math\_score=this->math\_score+*s*.math\_score;

        return temp;

    }

};

// 全局函数，实现对两个对象的操作后返回一个新的对象

Student operator+(Student &*s1*,Student &*s2*){

    // 略

}

// 左移运算符重载

// 只能利用全局函数重载

class Person{

friend ostream & oprator<<(ostream &cout,Person &p);

}

ostream & operator<<(ostream &*cout*,Person &*p*){

    cout<<//略

    return cout;

}

// 赋值运算符重载

class Student{

    private:

        string name;

    public:

        Student():name(){}

        Student(string *name*):name(name){}

    //默认拷贝构造函数（系统内自带，不用构造）

        Student(Student *stu*){

            this->name = stu.name;

        }

    // 拷贝赋值运算符

        Student& operator=(const Student& *other*){

            if(this!=other)

            {

                name=other.name;\

                age=other.age;

            }

            return \*this;

        }

    //赋值运算符重载

        Student& operator=(Student& *stu*){

            this->name = stu.name;

        }

        ~Student(){}

};

// 递增运算符重载

    Person& operator++(){

        m\_num++;

        return \*this;

    }

    Person operator++(int){

        Person temp=\*this;

        m\_num++;

        return temp;

    }

// 深拷贝和浅拷贝

// 浅拷贝：简单的赋值拷贝

    Person p2(p1);//调用系统默认构造函数

// 深拷贝：在堆区重新申请空间，进行拷贝

    // 重载拷贝构造函数

    Person (const Person& p){

        m\_height= new int (\*p.m\_height);

    }

// 函数调用运算符（仿函数）

    void operator()(string *str*){

        cout<<str<<endl;

    }

文本, 信件

描述已自动生成

**多态**

// 动态多态满足条件：

// 有继承关系

// 子类重写了父类的虚函数

class Animal{

    public:

        // 在基类定义虚函数

        virtual void speak(){

            cout<<"动物在叫"<<endl;

        }

};

class Dog : public Animal{

    public :

        void speak()override{//标志改写，确保改写成功

            cout<<"狗在叫"<<endl;

        }

};

void dospeak(Animal &*animal*){

*animal*.speak();

}

// 是否是多态的判定

    // 父类中有虚函数，子类重写父类的虚函数

    // 父类指针指向子类对象，或用父类引用绑定子类对象

    // 通过父类指针引用，调用的是子类虚函数

Dog dog;

Animal \*a=new Dog;//指针新子类对象，同时申请新空间

Animal \*b=&dog;//指针指向一个已经建立的子类对象

Animal &c=dog;//引用一个已经建立的子类对象

// 纯虚函数

// virtual 返回类型 函数名(参数列表)=0;

// 有纯虚函数的类也叫抽象类；抽象类不可创建对象

Animal \*a=new Dog;//可使用Dog的重写的函数

// 函数调用的同时新建对象

void Dowork(Animal\* *animal*){

*animal*->speak();

    delete *animal*;

}

Dowork(new Dog);

// 抽象父类的析构函数必须也是虚函数

// 虚析构法：virtual ~类名(){}

// 纯虚析构：virtual ~类名() = 0; 类名::~类名(){}

**UML统一建模语言**

// 类图：描述系统中类的内部结构以及类与类之间的关系，包括继承、组合、关联等。有助于理解系统的静态结构。

    // 泛化（继承）

    // 接口（实现）实现关系

    // 关联（类和类之间有关系）

    // 聚合关系（小类型可以单独存在）

    // 组合关系（小类型单独存在没有意义）

// 时序图

    // 描述了系统中对象之间的动态交互关系，展现了消息在时间轴上的传递顺序。时序图有助于理解系统的动态行为。

// 协作图

    // 强调的是发送和接收消息的对象之间的组织结构。

// 如果需要强调时间和序列，最好选择序列图；如果需要强调上下文相关，最好选择协作图。

图示

描述已自动生成图示

描述已自动生成

图示

描述已自动生成

**设计原则**

// 类图：描述系统中类的内部结构以及类与类之间的关系，包括继承、组合、关联等。有助于理解系统的静态结构。

    // 泛化（继承）

    // 接口（实现）实现关系

    // 关联（类和类之间有关系）

    // 聚合关系（小类型可以单独存在）

    // 组合关系（小类型单独存在没有意义）

// 时序图

    // 描述了系统中对象之间的动态交互关系，展现了消息在时间轴上的传递顺序。时序图有助于理解系统的动态行为。

// 协作图

    // 强调的是发送和接收消息的对象之间的组织结构。

// 如果需要强调时间和序列，最好选择序列图；如果需要强调上下文相关，最好选择协作图。

// OO（面向对象）六大原则

// 开闭原则：一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。

    // 软件实体应当对扩展开放，对修改关闭。

    // 通过抽象类及接口，规定了具体类的特征作为抽象层，相对稳定，不需修改，从而满足“对修改关闭”；

    // 从抽象类导出的具体类可以改变系统的行为，从而满足“对扩展开放”

// 里氏代换原则：子类可以扩展父类的功能，但不能改变父类原有的功能。

    // 所有引用基类的地方必须能透明地使用其子类的对象。

    // 如果子类不能完整实现父类的方法，或者是父类的某些方法在子类中已经发生“畸变”，那么建议断开父子继承关系，采用依赖、聚集、组合等关系代替继承。

// 单一职责原则：一个类只负责一项职责。

// 依赖倒置原则：多用抽象的接口来描述相同的动作。

// 接口隔离原则：使用多个专门的接口，而不使用单一的总接口，即客户端不应该依赖那些它不需要的接口。

// 迪米特法则：一个对象应该对其他对象保持最少的了解。类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。

**设计模式**

// 设计模式

//公有+，保护#和私有操作-

// 简单工厂模式：（黑盒）

    // 定义一个简单工厂类，它可以根据参数的不同返回不同类的实例，被创建的实例通常都具有共同的父类

    // 工厂：根据客户提供的具体产品参数，创建具体产品实例

    // 抽象产品：具体产品类的基类，包含创建产品的公共方法

    // 具体产品：抽象产品的派生类，包含具体产品特有的实现方法，是简单工厂模式的创建目标

// 策略模式：

    // 定义一系列算法，将每一个算法封装起来，并让它们可以相互替换。

    // 允许用户从该算法族中任意选择一个算法解决问题，同时还可以方便地更换算法或者增加新的算法。

    // 将算法族中的每一个算法都封装成一个类，每一个类称为一个策略（Strategy）。

// 代理模式：

    // 给某个对象提供一个代理或占位符，并由代理对象来控制对原对象的访问。

    // （不改变原代码情况下对对象的访问进行控制管理）

    // 远程代理：在分布式系统中，客户端可能需要访问位于远程服务器上的对象。

        // 通过使用远程代理，客户端可以通过网络请求将方法调用传递给远程对象，代理对象负责处理网络通信和数据传输。

    // 虚拟代理：在某些情况下，创建和初始化一个对象可能会非常耗时，或者对象的实例化是一种昂贵的操作。

        // 使用虚拟代理，可以在需要的时候延迟真实对象的创建和初始化，直到客户端真正需要使用该对象。

    // 安全代理：在访问真实对象之前，代理对象可以对客户端的请求进行验证和授权。

        // 安全代理用于控制对真实对象的访问权限，可以实现身份验证、访问控制等安全功能。

    // 缓存代理：在某些场景下，代理对象可以缓存真实对象的结果，以便在后续的请求中直接返回缓存的结果，而无需再次调用真实对象。

        // 缓存代理用于提高系统的性能和效率。

    // 日志代理：代理对象可以在调用真实对象的方法前后添加日志记录的功能，用于记录方法的调用时间、参数和返回值等信息。

        // 日志代理用于实现日志记录、性能监控和调试等功能。

// 观察者模式：

    // 定义对象之间的一种一对多的依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象都得到通知并被自动更新。

**模板**

// 模板就是建立通用的模具，大大提高复用性

// 模板两种模板机制：

    // 函数模板

    // 类模板

// 函数模板实现

    // template <typename/class T>// T foo(T a){}// 函数声明定义（T\E\O）

    // 显式实例化：add<int>(4,5)

    // 隐式实例化：add(4,5)

    // 显式特化：对特定类型，提供特定模板,参数个数必须相同

        // template<> // void foo<int>(int arg) {} // int类型的特定实现

// 函数模板规则

    // 普通函数优先调用于模板函数

    // 可重载（根据参数个数）

// 类模板

    //建立一个通用类，类中的成员的数据类型是不指定的（不具体），用一个或者多个虚拟的类型来表达

    // template <typename/class T>// 类声明

    // 只能使用显式实例化

    // 类模板可以有默认参数 // template <class NameType, class AgeType = int>

// 类模板的继承

    // class Son1 : public Base<int>// 父类是类模板，所以要指定类型

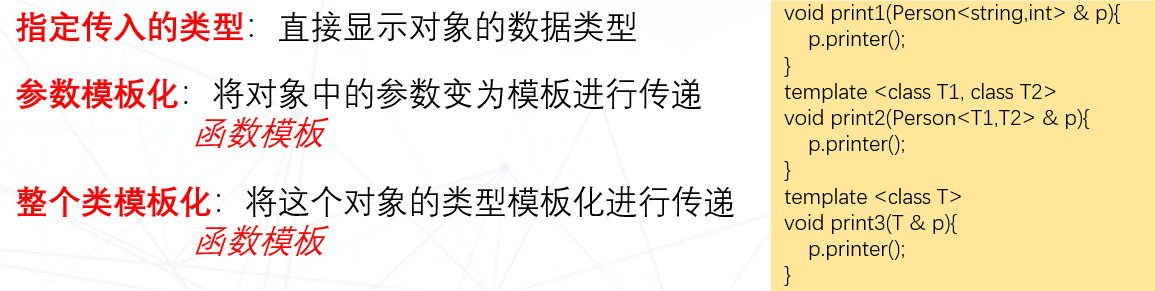
    // template <class T1, class T2> // class Son2 : public Base<T1>{} // T1替换T

    // 模板成员函数在类外实现，要加模板参数列表

// 类模板的特化

    // 完全特化

    // 偏特化

****

// 容器（实际上是一系列定义好的类模板）

    // 容器管理着为其元素分配的存储空间，并提供成员函数来直接访问或通过迭代器

    // （具有类似于指针的属性的对象）访问它们。

// 容器分类和算法

    // 顺序容器

        // 数据按插入顺序依次存储和管理，并提供按顺序访问

        // 增加/删除快，查找/修改慢

        // vector  动态的连续数组

        // list 双链表

    // 关联容器

        // 需要按照指定规则（默认升序）排序后再存储

        // 查找/修改快，增加/删除慢

        // set 唯一键的集合，按照键排序

        // map 键值对的集合，按照键排序，键是唯一的

    // 算法

        // 操作数据集合的处理逻辑，实际上是一系列定义好的函数模板

        // 迭代器

            // 是一种用于访问容器中元素的对象

            // 可以遍历容器中的元素，并提供对元素的读取、修改和删除等操作

//  动态数组vector

    // 提供一种高效的动态数组，可以方便地进行元素的插入、删除和访问操作。

    // 动态大小：vector的大小可以在运行时动态调整，可以根据需要自动增长或缩小。

    // 随机访问：vector支持通过下标快速访问任意位置的元素，时间复杂度为O(1)。

    // 连续存储：vector中的元素在内存中是连续存储的，这样可以提高数据的访问效率。

    // 动态内存管理：vector内部自动处理了动态内存的申请和释放，无需手动管理内存。

// 查找和排序

    // 标准模板库STL, 头文件<algorithm>中的查找算法find, 排序算法sort

    // #define LOG ( cout << "[" << \_\_LINE\_\_ << " " <<\_\_PRETTY\_FUNCTION\_\_ << "] " )

        // \_\_LINE\_\_是一个内置宏，表示当前代码所在的行号。

        // \_\_PRETTY\_FUNCTION\_\_是一个编译器特定的内置宏，表示当前函数的名称

|  |  |
| --- | --- |
| **迭代器操作** | **元素访问** |
| **begin()：返回容器的起始位置的迭代器(指向第一个元素)**  **end()：返回容器结束位置的迭代器(指向最后一个元素的后面)**  **advance()：将迭代器向前移动指定的距离**  **next()：返回当前迭代器的下一个迭代器**  **prev()：返回当前迭代器的前一个迭代器** | **\*it：返回迭代器it 指向的元素**  **it->：通过迭代器it访问元素的成员**  **at()：返回容器中指定位置的元素**  **front()：返回容器的第一个元素**  **back()：返回容器的最后一个元素** |

|  |  |
| --- | --- |
| **容器修改** | **容器操作** |
| **insert()：在指定位置插入一个或多个元素**  **erase()：删除指定位置或指定范围的元素**  **push\_back()：将元素添加到容器的末尾**  **pop\_back()：删除容器的最后一个元素**  **push\_front()：将元素添加到容器的开头**  **pop\_front()：删除容器的第一个元素**  **resize()：改变容器的大小** | **sort()：对容器中的元素进行排序 (不适用于list)**  **reverse()：将容器中的元素反转**  **unique()：去除容器中相邻的重复元素**  **merge()：将两个已排序的容器合并成一个已排序的容器**  **find()：在容器中查找指定元素，并返回其位置的迭代器**  **count()：计算容器中指定元素的数量**  **remove()：从容器中删除指定元素（将要删除元素移到末尾）** |

|  |  |
| --- | --- |
| **创建vector** | |
| **vector<类型> myVector;**  **vector<类型> myVector(初始大小);**  **vector<类型> myVector(初始大小, 初始值);**  **vector<类型> myVector(另一个vector);** | **// 创建一个空的vector**  **// 创建一个具有初始大小的vector**  **// 创建一个具有初始大小和初始值的vector**  **// 创建一个与另一个vector相同的vector** |
| **添加元素** | |
| **myVector.push\_back(元素);** | **// 在vector的末尾添加一个元素** |
| **修改元素** | |
| **myVector[index] = 新值;** | **// 修改vector中指定索引的元素** |
| **访问元素** | |
| **myVector[index];**  **myVector.at(index);**  **myVector.front();**  **myVector.back();** | **// 访问vector中指定索引的元**  **// 安全访问vector中指定索引的元素，会进行边界检查**  **// 访问vector中第一个元**  **// 访问vector中最后一个元素** |
| **删除元素** | |
| **myVector.pop\_back();**  **myVector.erase(迭代器);**  **myVector.erase(起始迭代器, 结束迭代器);** | **// 删除vector中最后一个元素**  **// 删除vector中指定迭代器指向的元素**  **// 删除vector中指定范围的元素** |
| **插入元素** | |
| **myVector.insert(迭代器, 元素);**  **myVector.insert(迭代器, n, 元素);**  **myVector.insert(迭代器, 起始迭代器, 结束迭代器);** | **// 在指定迭代器位置之前插入一个元素**  **// 在指定迭代器位置之前插入n个相同的元素**  **// 在指定迭代器位置之前插入另一个vector中指定元素** |
| **获取容器大小** | |
| **myVector.size();**  **myVector.empty();** | **// 返回vector中元素的个数**  **// 判断vector是否为空，返回true或false** |
| **清空vector** | |
| **myVector.clear();** | **// 清空vector中的所有元素** |

文本

低可信度描述已自动生成文本

描述已自动生成**LIST**

// vector的优点包括：

    // 高效的随机访问：可以通过下标快速访问任意位置的元素。

    // 动态调整大小：可以根据需要动态增长或缩小，灵活性较高。

    // 内存管理方便：vector内部自动管理内存，无需手动申请和释放。

// vector的缺点包括：

    // 插入和删除元素的效率较低：在中间位置插入或删除元素需要移动后续的元素。

    // 无法直接插入和删除指定位置以外的元素：需要先移动元素到指定位置再进行插入或删除。

    // 当容量不够时，重新分配内存并拷贝元素的开销较大。

//list的优点包括：

    // 插入和删除元素的效率高，不会造成元素的移动。

    // 支持在任意位置进行插入和删除操作。

    // 不需要预先指定容器的大小。

// list的缺点包括：

    // 访问元素的效率较低，因为需要遍历链表。

    // 需要额外的内存空间存储链表节点的指针。

// std::list 链表，顺序容器

    // 链式分配内存：一个个节点动态分配内存，然后通过指针链接成一串

    // template< class T, class Allocator = std::allocator<T> > class list

// set

    // set 是一个有序关联容器，其中的元素按照升序排列，且不允许重复元素。

    // set 中的元素是唯一的，即任意两个元素不能相等。

|  |  |  |
| --- | --- | --- |
| **创建set容器** | | |
| **std::set<int> mySet;**  **std::set<int> mySet2 = {1, 2, 3};**  **std::set<int> mySet3(mySet2);** | **// 创建一个空的set**  **// 创建一个具有初始值的set**  **// 创建一个与另一个set相同的set** | |
| **添加元素** | | |
| **mySet.insert(5);**  **std::set<int> anotherSet = {1, 2, 3};** | **// 向set中插入元**  **// 向set中插入一段迭代器范围内的元素** | |
| **mySet.insert(anotherSet.begin(), anotherSet.end());** | | |
| **访问元素** | | |
| **for (auto it = mySet.begin(); it != mySet.end(); ++it) {**  **std::cout << \*it << " ";**  **}** | **// set不支持通过索引访问元**  **// 遍历set中的元素** | |
| **删除元素** | | |
| **mySet.erase(5);**  **auto it = mySet.find(10);**  **if (it != mySet.end()) {**  **mySet.erase(it);**  **}**  **mySet.erase(mySet.begin(), mySet.end());**  **mySet.clear();** | | **// 删除set中的元素**  **// 删除set中指定迭代器指向的元素**  **// 删除set中指定范围的元**  **// 清空set中的所有元素** |

insert函数的返回值是一个 std::pair<iterator, bool>, 其中 iterator 指向插入的元素，bool 表示插入是否成功（如果元素已存在，则返回 false）

// 关联容器，集合set

    // 存储数据时，自动排序（默认升序排序）后存储，值不能重复（自动去重）

    // template< class Key, class Compare = std::less<Key>,

            // class Allocator = std::allocator<Key> > class set;

    // Key是元素类型，

    // Compare是比较类型（用来比较存储的元素数据），

        // 默认使用标准库的小于比较类模板less传递元素类型Key所特例化的类型，

        // 即内部针对元素进行小于比较后存储

        // 所以元素类型应提供小于比较的能力（自定义类型需重载小于比较运算符），

        // 也可以手动传递比较类型，处理元素类型数据的比较

    // Allocator是分配类型，使用默认提供的版本即可

// 优点：

    // set 具有自动排序的特性，可以方便地对元素进行有序操作。

    // set 内部使用红黑树实现，插入、删除、查找等操作效率较高。

    // set 内部的元素唯一，可以用来对元素进行去重操作。

// 缺点：

    // set 不支持随机访问，即不能通过下标来访问元素，只能通过迭代器进行访问。

    // set 的插入、删除等操作可能会导致整个红黑树的调整，因此在大量数据插入或删除时效率可能较低。

    // set 中的元素是唯一的，如果需要存储重复元素，就不能使用 set。

// map

    // 是一种关联容器，它存储键值对，并根据键来进行快速查找。根据键值排序

    // map=set+键

|  |  |
| --- | --- |
| **创建map容器** |  |
| **std::map<int, std::string> myMap** | **// 创建一个空的map** |
| **插入元素** | |
| **myMap.insert(std::make\_pair(1, "one"));**  **myMap[2] = "two";** | **//插入单个元素，前键后值**  **//使用下标操作符插入元素** |
| **删除元素** | |
| **myMap.erase(2);**  **myMap.clear();** | **// 删除指定键的元**  **// 清空map** |
| **访问元素** | |
| **// 使用迭代器遍历map**  **for (auto it = myMap.begin(); it != myMap.end(); ++it) {**  **std::cout << it->first << ": " << it->second << std::endl;**  **}**  **std::cout << myMap[1] << std::endl;**  **auto it = myMap.find(2);**  **if (it != myMap.end()) {**  **std::cout << it->first << ": " << it->second << std::endl;**  **}** | **// 使用下标操作符访问元**  **//查找元素** |

// 关联容器    键值对容器/映射表 map

    // template<class Key,class T,class Compare = std::less<Key>,

            // class Allocator =std::allocator<std::pair<const Key, T>>> class map;

    // 内部存储std::pair<Key, Value>类型的数据，一个键值对元素，pair内部包含两个数据{first, second}

    // template< class T1, class T2> struct pair; 内部形如{ T1 first; T2 second; 其它代码}

    // map类似于set，自动按Key排序（默认升序排序），键值对中的Key的数据值不能重复（自动去重）

    // -> 键值对元素中的"键"一一对应/关联/映射到某个"值"

// 优点：

    // 快速查找：由于内部使用红黑树实现，map具有较快的查找速度，适用于大规模数据的存储和查找。

    // 自动排序：map会自动根据键的大小进行排序，省去了手动排序的麻烦。

// 缺点：

    // 内存占用较大：由于需要存储键和值，map的内存占用较大，不适合存储大量的数据。

    // 插入和删除效率较低：由于需要维护红黑树的平衡性，插入和删除操作的效率相对较低。

enum fmtflags {

skipws = 1L << 0,

left = 1L << 1,

right = 1L << 2,

internal = 1L << 3,

dec = 1L << 4,

oct = 1L << 5,

hex = 1L << 6,

showbase = 1L << 7,

showpoint = 1L << 8,

uppercase = 1L << 9,

showpos = 1L << 10,

scientific = 1L << 11,

fixed = 1L << 12,

unitbuf = 1L << 13,

stdio = 1L << 14,

boolalpha = 1L << 15,

adjustfield = left | right | internal,

basefield = dec | oct | hex,

floatfield = scientific | fixed,

};

枚举类型fmtflags定义了一系列用于格式化输出的标志位