C++

零、STL

1.vector容器

```
#include<vector>
#include<algotithm>
// 定义
vector<int> v = { 2, 1, 3, 3, 2 };
vector<int> v(10,1); //包含10个为1的元素
vector<vector<int>> dp(4, vector<int>(5, 0)); //定义4×5的二维容器
// 插入
v.push_pack(1); // 向vector插入数据
v.emplace_back(1);//更高效
v.insert(index,elem);
// 访问数据
int n = v.front();  int n = v.back();
// 遍历
for(int i=0; i<v.sise(); i++)
for(int i : v)
// 删除
v.erase(v.begin() + 2);  // 删除索引为2的元素
v.pop_back();//删除最后的元素
// 排序
sort(v.begin(), v.end()); // 从小到大
// 自定义排序函数：从大到小
static bool compare(int a, int b)
    return a > b; // 如果a应该排在b之前，返回true
sort(v.begin(), v.end(), compare);//v存储int，compare就传int
// 从大到小2
sort(v.begin(), v.end(), [](int a, int b) {return a > b; });
// v存储节点，并按照节点的值升序排序
sort(v.begin(),v.end(),[](ListNode* a, ListNode* b) {return a->val < b->val;});
// 拼接
vec1.insert(vec1.end(), vec2.begin(), vec2.end());
// 查找
if(v.find(elem) != v.end())
```

```
if(find(v.begin(),v.end(),elem)!=v.end()))
// 交换
swap(v[a], v[b]);
// 取一部分
vector<int> v = {0,1,2,3,4,5};
vector<int> subv(v.begin()+a,v.begin()+b);//左闭右开原则
subv(v.begin(),v.begin()+3);   // {0,1,2}
subv(v.begin()+1,v.begin()+3); // {1,2}
subv(v.begin()+1,v.end());     // {1,2,3,4,5}，因为end实际上指向最后一个元素的后面
subv(v.begin()+1,v.end()-1);   // {1,2,3,4}
```

2.容器存放自定义类型:

```
#include<vector>
class Person
{
	public:
		Person(string name, int age)
		{
			p_name = name;
			p_age = age;
		}
		string p_name;
		int p_age;
};
void test1()
{
	vector<Person> v;
	Person p1("aaa", 10);
	Person p2("bbb", 20);
	Person p3("ccc", 30);
	v.push_back(p1);
	v.push_back(p2);
	v.push_back(p3);
	for(vector<Person>::iterator it = v.begin(); it != v.end(); it++)
	{
		cout << "Name:" << (*it).p_name<< endl;
		cout << "Age:" << it->p_age << endl;
	}
}
void test2() //保存地址
{
	vector<Person*> v;
	Person p1("aaa", 10);
	Person p2("bbb", 20);
	Person p3("ccc", 30);
	v.push_back(&p1);
	v.push_back(&p2);
	v.push_back(&p3);
	for(vector<Person>::iterator it = v.begin(); it != v.end(); it++)
	{
		cout << "Name:" << (*it)->p_name<< endl;
		cout << "Age:" << (*it)->p_age << endl;
	}
}
```

3.容器嵌套

```
#include<vector>
void test()
```

```
{
	// 大容器
	vector<vector<int>> v;
	// 小容器
	vector<int> v1;
	vector<int> v2;
	vector<int> v3;
	// 向小容器添加数据
	for(int i = 0; i < 4; i++)
	{
		v1.push_back(i+1);
		v1.push_back(i+2);
		v1.push_back(i+3);
	}
	// 将小容器添加到大容器
	v.push_back(v1);
	v.push_back(v2);
	v.push_back(v3);
	// 遍历
	for(vector<vector<int>>::iterator it = v.begin(); it != v.end(); it++)
	{
		// (*it)--容器
		for(vector<int>::iterator itv=(*it).begin();itv!=(*it).end();itv++)
		{
			cout << *itv << endl;
		}
	}

}
```

2.string容器

```
#include<string>
// 赋值
	string str1, str2, str3, str4, str5;
	str1 = "Hello";
	str2 = 'a';
	str3.assign("Hello");
	str3.assign("Hello", 3); // 取前三个
	str5.assign(10, 'w');  // 10个w组成
// 拼接
	string str1 = "我是";
	str1 += "ZZ";
	cout << str1 << endl; // 我是ZZ
	str1 += 'p';
	cout << str1 << endl; // 我是ZZP
	sring str2 = "I";
	str2.append("Love");
	str2.append("Love", 3);
// 查找
	string str1 = "abcdefg";
	int index = str1.find("de");// index = 3，出现的位置
	index = rfind("de"); // 从后向前找
// 替换
	string str1 = "abcdefg";
	str1.replace(1,3,"1111");
	// 位置1开始3个字符，替换为1111
	// bcd -> 1111, a1111efg
// 比较
	// 比较ASCII码
```

```
        // 相等返回0，大于返回1，小于返回-1
// 插入删除
        string str1 = "hello";
        str1.insert(1, "111"); //h111ello
        str1.erase(1, 3);      //ho
// 求子串
        string str = "abcdef";
        string subStr = str.substr(1,3); // bcd
        string subStr = str.substr(1); // bcdef
// 排序
        string s = "cba";
        sort(s.begin(), s.end());
```

3.map/set容器

`map`：存储的是`键值对`，键唯一

```
map<char, int> m =
   {
      {'I', 1},
      {'V', 5},
      {'X', 10},
cout << m['X']; // 10
m.count(I);    // 返回true
m['G'];        // 会自动创建一个键值对，默认为0
```

`set`：只存储`键`，键唯一

```
set<int> s;
s.emplace(10);
s.insert(10);
```

4.哈希表

`键的唯一性`：每个键在哈希表中是唯一的，如果插入重复的键，则插入失败。

```
// 定义
unordered_map<int, int> m;
// 插入元素
m[1] = "one";
m.insert({2, "two"});
// 查找元素
if (m.find(ch) != m.end())
// 访问元素
for (auto pair : m)
        pair.first; pair.second; // 键.值
```

5.堆栈

```
stack<char> stk; //创建栈
stk.push(ch);    //进
stk.pop();       //出
stk.top()        //最前
```

6.链表

```
// 定义一个链表节点
struct ListNode
{
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(nullptr) {}
};
// 第一个
ListNode* cur = head;
cur->next;      // 下一个
cur->next->next;// 下两个
// 注：如果移动head，cur也会跟着移动
// 创建新节点，作为新的头结点
ListNode* hair = new ListNode(0);
hair->next = head;
```

7.二叉树

```
// 定义二叉树节点
struct TreeNode
{
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x): val(x), left(nullptr), right(nullptr){}
};
// 实例化节点
TreeNode* root = new TreeNode(x);
```

8.deque

```
#include <deque>
deque<int> d;
d.push_back();
d.push_front();
d.pop_front() ;
d.pop_back();
```

9.queue

q.push();

q.pop();

```
struct compare {
    bool operator()(int a, int b) const {
        return a < b; // 从大到小排列(top为最大)
    }
};
int main() {
    priority_queue<int, vector<int>, compare> pq;
```

一、数据类型

1. 整型

1. 短整型 `short` ：2字节
2. 整型 `int` ：4字节
3. 长整型 `long` ：4字节
4. 长长整型 `longlong` ：8字节

2.浮点型

1. 单精度`float` ：4字节，7位有效数字
2. 双精度`double` ：8字节，15~16位有效数字

```
float a = 3.14f; double b = 3.14;
```

3.字符型

1. 单个字符

```
char ch = 'a';
```

2.转义：`\n` 换行 `\t` 制表符 `\\` 反斜杠

3.ASCII:

数字48-57，大写字母65-90，小写字母97-122

4.转换

```
string s = "13";
int num = stoi(13);
#include <cctype>
char ch = 'A';
tolower(ch);
toupper(ch);
```

4.字符串

```
//C风格，以空字符结尾
char str[] = "hola";
// C++风格
#include <string>;
char str = "hola";
// 转换为字符串
str = to_string(x);
// 求长度
str.size(); str.length();
// 颠倒
reverse(s.begin(), s.end());
// 求子串
```

```
string str = "abcdef";
string subStr = str.substr(1,3); // bcd
string subStr = str.substr(3); // def
// 删除
str.pop_back();//删除最后一个字符
// 排序
sort(str.begin(), str.end());
// 读取
getchar();//读取一个字符
getline(cin, str);//从键盘读取一个字符串到str
```

5.布尔类型

```
bool flag = true;
```

6.数组

```
int arr1[5];
int arr2[5] = {1,2,3,4,5}
int arr3[] = {1,2,3,4,5}
int arr4[5] = {0};
sizeof(arr)/sizeof(arr[0]);  //大小
```

7.二维数组

```
int arr1[2][3];  //2行3列
int arr2[2][3] = {{1,2,3}, {4,5,6}};
int arr3[2][3] = {1,2,3,4,5,6};
int arr4[][3] = {1,2,3,4,5,6};  //行可省略，列不可
```

8.位运算

符号	描述	运算规则
&	与	两位都为1，那么结果为1
|	或	有一位为1，那么结果为1
~	非	~0 = 1， ~1 = 0
^	异或	两位不相同，结果为1
<<	左移	各二进位全部左移若干位，高位丢弃，低位补0
>>	右移	各二进位全部右移若干位，对无符号数，高位补0，有符号数，各编译器处理方法不一样，有的补符号位（算术右移），有的补0（逻辑右移）

二、流程结构

1.if

在C++中，`if`语句是一种条件控制结构，用于根据特定条件执行或跳过代码块。`if`语句的基本语法如下：

```
int num = 0;
cout << "输入数字：" << endl;
cin >> num;

if (condition)
{
    // 当条件为真（true）时执行的代码块
}
else
{
    // 当条件为假（false）时执行的代码块（可选）
}
```

此外，C++还支持使用逻辑运算符（如`&&`、`||`和`!`）组合多个条件。例如：

```
if (condition1 && condition2)
```

```
{
    // 当condition1和condition2都为true时执行的代码块
}
else if (condition3 || condition4)
 {
    // 当condition3或condition4至少有一个为true时执行的代码块
}
else {
    // 当所有条件都为false时执行的代码块（可选）
}
```

2.switch

switch语句是一种多分支选择结构，它根据一个表达式的值来选择执行相应的代码块，基本语法如下：

```
switch (表达式)
{
    case 常量1:   // 当表达式的值等于常量1时执行的代码
    break;
    case 常量2:   // 当表达式的值等于常量2时执行的代码
    break;
    ...
    default:      // 当表达式的值不等于任何case常量时执行的代码
}
```

3.while

```
while(循环条件)
{
        循环语句
}
```

4.do...while

不管while条件，先执行一次循环

```
do
{
        循环语句
}
while(条件)
```

4.for

```
for(int i = 0; i < 5; i++)
{
        cout << i << endl;
}
```

范围for语句：一种简单的遍历方法

```
int arr[5] = {1,2,3,4,5};
for(int x : arr)    //优先选用auto，若需修改：auto &x : arr
{
```

```
    cout << x << endl;
}
```

三、函数

1.函数定义

```
int add(int a, int b)  //int：返回值的类型
{
    int sum = a + b;
    return sum;
}
void add(){}
```

2.函数调用

```
int add(int a, int b)
{
    ……
}
int main()
{
    int num1 = 1;
    int num2 = 2;
    int sum = add(num1, num2);
    cout << "sum =" << sum << endl;
}
```

3. 引用

1. 引用：给变量起别名，语法：`数据类型 &别名 = 原名`

```
int a = 10;
int &b = a;
```

2.注意事项：1.必须初始化2.初始化后不可更改

```
int a = 10;
int &b;
&b = a;//错误
int c = 20;
b = c;//此时abc都为20
```

3.函数引用传递

```
void swap(int *A, int *B)
{
    int temp = A;
    A = B;
    B = temp;
}
int main()
{
```

```
    int a = 10;
    int b = 20;
    swap(a,b);
}
```

4.引用的本质：指针常量

```
int a = 10;
int &b = a;
// 相当于下面的代码：
int * const b = &a;//指针指向不可改
```

4.函数的默认参数

```
int add(int a, int b = 20) //默认参数必须放到结尾
{
    return a+b;
}
int main()
{
    add(10);    //结果：30
    add(10,10); //结果：20
}
```

如果函数声明有默认参数，函数实现就不能有

```
int add(int a = 10, int b = 20)
//.....
int add(int a, int b)
{
    return a+b;
}
```

5.函数的占位参数

```
void func(int a, int)
{}
int main()
{
    func(10, 10);
}
```

6.函数重载

作用：函数名可以相同，提高复用性

前提条件：

- 同一个作用域
- 函数名相同
- 函数参数的类型不同或个数不同或顺序不同

```
void func()
{...}
void func(int a)
{...}
int main()
{
	func();
	func(10);
}
```

四、指针

1.定义

指针可以保存一个`地址`，指针就是一个地址

```
int a = 10;
//方法1
int *p;
p = &a;  //p是内存，*p是对应内存所存储的数据
//方法2
int *p = &a;
```

2.空指针

用作初始化指针变量，但其指向的内存不可访问

```
int *p = NULL;
cout << *p <<endl; //报错
```

3.野指针

指向非法内存的指针

```
int *p = (int *)0x1100;
```

4.const修饰指针

```
const int *p = &a;  //*p不可改，p可改
int * const p = &a;  //p不可改，*p可改
const int * const p = &a;  //p、*p都不可改
```

5.指针与数组

```
int arr[10] = {1,2,3,4,5,6,7,8,9,10};
int *p = arr;
// *p指向第一个元素
p++；
// *p指向第二个元素
```

五、结构体

1.定义

```
struct Student
{
	string name;
	int age;
	int score;
};
```

2.使用

使用时struct关键字可以省略

```
// 方法1
struct Student s1;
s1.name = "ZZP";
s1.age = 18;
s1.score = 90;
// 方法2
struct Student s2 = {"zzp", 18, 90};
```

3.结构体数组

```
struct Student students[3] =
{
	{"zzp", 18, 90},
	{"zzp", 18, 90},
	{"zzp", 18, 90}
};
for(int i = 0, i < 3, i++)
{
	cout << "姓名：" << students[i].name << endl;
};
```

4.结构体指针

```
struct Student s = {"zzp", 18, 90};
struct Student *p = &s;
cout << "姓名：" << p->name << endl;
```

5.结构体嵌套

```
struct Student
{
	string name;
	int age;
};
struct Teacher
```

```
{
	string name;
	struct Student s1;
};
int main(){
	struct Teacher t;
	t.name = "Wei Li";
	t.s1.name = "zzp";
	t.s1.age = 22;
};
```

6.结构体作函数的参数

```
struct Student
{
	string name;
	int age;
};
void printStudent1(struct student s)  // 值传递，不可改变实参
{
	cout << "姓名：" << s.name << endl;
};
void printStudent2(struct student *p)  // 地址传递，可改变实参
{
	cout << "姓名：" << p->name << endl;
};
int main(){
	struct student s;
	s.name = "zzp";
	s.age = 22;
	// 不想改变主函数中的数据，用值传递；反之，地址传递
	printStudent1(s);
	printStudent2(&s);
};
```

7.const的使用

```
void printStudent(const Student *s1)
{
	// 只能读不能写，防止误操作
};
```

六、程序四区

1. 代码区

存放CPU执行的机器指令，特点：`共享`和`只读`

2. 全局区

存放全局变量、静态变量、常量

变量：包括`const修饰的全局/局部变量`和`字符串常量`

```
int a = 10 ;            //全局变量：main函数外定义的变量
int main()
{
        int b = 10;     //局部变量
        static s_b = 10;//静态变量
        string c = "hello";
}
```

3. 堆

由程序员分配释放，主要使用`new`在堆区开辟内存

```
int* func()
{
        int *p = new int(10); //开辟
        int *arr = new int[10];//10个元素
        return p;
}
int main()
{
        int *p = func();
        delete p;           //释放
        delete[] arr;
}
```

4. 栈

由编译器分配释放，局部变量、形参

```
int* func()
{
        int a = 10;
        return &a; //不要这样做
}
```

七、类和对象

1.封装

属性和行为，称为成员

示例：学生类

```
class Student
{
public:
        //属性（成员属性、成员变量）
        string name;
        int ID;
        //行为（成员函数、成员方法）
        void showStudent()
        {
                cout<<"Name:"<<name<<"ID:"<<ID<<endl;
        }
        void setName(string n_name)
```

```
    {
        name = n_name;
    }
};
int main()
{
    Student stu1;
    stu1.name = "ZZP";
    stu1.ID = 001;
    stu1.showStudent();
    stu1.setName("张三");
}
```

2.权限控制

- 公共权限 `public`

  类内、类外都可以访问

- 保护权限 `protected`

  类内可以访问，类外不可访问（子类可访问）

- 私有权限 `private`

  类内可以访问，类外不可访问（子类不可访问）

```
class Person
{
public:
    string name;
protected:
    string location;
private:
    int password;
public:
    void func()
    {
        name = "ZZP";
        location = "南二舍";
        password = 123456;
    }
};
int main()
{
    Person p1;
    p1.name = "李四";
    p1.location = "南三舍"; //出错！
    p1.password = 123;    //出错！
}
```

3.类与结构体的区别

- `struct`：默认权限**公共public**
- `class`：默认权限**私有private**

4.构造函数和析构函数

- 构造函数：不写void，函数名与类相同，创建对象时自动调用
- 析构函数：不写void，函数名与类相同，名称前加~，对象销毁前自动调用

```
class Person
{
public:
	Person()
	{
		cout<<"这是构造函数"<<endl;
	}
	~Person()
	{
		cout<<"这是析构函数"<<endl;
	}
};
```

5.初始化列表

```
class Person
{
public:
	Person(int a, int b):m_A(a),m_B(b)
	{}
	int m_A;
	int m_B;
};
void test()
{
	Person p(10, 20);
}
```

6.对象成员

```
class Stu1{};
class Stu2
{
	Stu1 s1;
};
```

7.静态成员

所有类共享一份数据，两种访问方式：

- 直接用类名访问
- 创建对象访问

静态成员函数只能访问静态成员变量

```
class Person
{
public:
	static int age;
	static void func()
	{
		cout<<"调用静态成员函数"<<endl;
```

```
    }
};
int Person::age = 20; //直接通过类名访问
int main
{
    Person p1;
    cout<<p1.age<<endl;//输出20
    Person p2;
    p2.age = 30;
    cout<<p2.age<<endl;//输出30
    //两种访问方式
    p1.func();
    Person::func();
}
```

8.this指针

```
class Person
{
public:
    Person(int age)
    {
        this->age = age;//防止名称重复
    }
    void AddAge(Person &p)
    {
        this->age += p.age;
    }
    int age;
};
void test
{
    Person p1(10);
    Person p2(10);
    p2.AddAge(p1);//p2.age为20
}
```

8.友元

在程序里，有些私有属性也想让类外特殊的一些函数或者类进行访问，就需要用到友元技术

友元的目的就是让一个函数或者类，访问另一个类中私有成员

```
class Building
{
    friend void GoodBuddy(Building *building);//使GoodBuddy可访问private
public:
    Building()
    {
        Sittingroom = "客厅";
        Bedroom = "卧室";
    }
    string Sittingroom;
private:
    string Bedroom;
};
void GoodBuddy(Building *building)
{
    cout<<"好朋友正在访问:"<<building->Bedroom<<endl;
```

```
}
Building::Building()//类外实现
{
	Sittingroom = "客厅";
	Bedroom = "卧室";
}
```

9.运算符重载

1.加号运算符重载

2.左移运算符重载

3.递增运算符重载

4.赋值运算符重载

5.关系运算符重载

6.函数调用运算符重载

10.继承

1.继承的语法

```
class BasePage
{
Public:
	void head()
	{cout<<"公共头部"<<endl;}
	void foot()
	{cout<<"公共尾部"<<endl;}
};
class PythonPage: public BasePage //继承
{
	void cotent()
	{cout<<"Python视频"<<endl;}//单独功能
};
```

2.三种继承方式

```
class A{
public:
	int a;
protected:
	int b;
private:
	int c;
};
```

```
//公有继承
class B: public A{
public:
```

```
//保护继承
class B: protected A
{
```

```
//私有继承
class B: private A
{
```

```
    int a;
protected:
    int b;
被继承但不可访问：
    int c;
};
```

```
protected:
    int a;
    int b;
被继承但不可访问：
    int c;
};
```

```
private:
    int a;
    int b;
被继承但不可访问：
    int c;
};
```

3.继承中的构造和析构顺序

父类构造→子类构造→子类析构→父类析构

4.继承中同名函数

```
class Dad{
public:
    Dad()
    {mA = 100;}
    int mA;
    void func()
    {
        cout<<"父类中函数"<<endl;
    }
};
class Son: public Dad
{
public:
    Son()
    {mA = 200;}
    int mA;
    void func()
    {
        cout<<"子类中函数"<<endl;
    }
};
void test()
{
    Son s;
    cout <<"Son中的mA= "<< s.mA << endl;
    cout <<"Dad中的mA= "<< s.Dad::mA << endl;
    s.func();
    s.Dad::func();
}
```

5.多继承

```
class Dad1{};
class Dad2{};
class Son: public Dad1, public Dad2
{};
```

11.多态

同一个父类的指针指向不同的子类函数，实现不同的功能

多态分为两类

- 静态多态: 函数重载 和 运算符重载属于静态多态，复用函数名

- 动态多态: 派生类和虚函数实现运行时多态

静态多态和动态多态区别

- 静态多态的函数地址早绑定 - **编译阶段**确定函数地址
- 动态多态的函数地址晚绑定 - **运行阶段**确定函数地址

```
//地址早绑定
class Animal{
public:
	void speak(){
		cout<<"动物叫"<<endl;
	}
};
class Cat: public Animal{
public:
	void speak(){
		cout<<"喵喵叫"<<endl;
	}
};
void AnimalSpeak(Animal &animal)
{
	animal.speak();
}
void test()
{
	Cat cat;
	AnimalSpeak(cat);//动物叫
}
```

```
//地址晚绑定
class Animal{
public:
	virtual void speak(){
		cout<<"动物叫"<<endl;
	}
};
class Cat: public Animal{
public:
	void speak(){
		cout<<"喵喵叫"<<endl;
	}
};
void AnimalSpeak(Animal &animal)
{
	animal.speak();
}
void test()
{
	Cat cat;
	AnimalSpeak(cat);//喵喵叫
}
```

动态多态：子类重写父类的虚函数，父类引用指向子类的传入对象

```
//案例：计算器
//开闭原则：对扩展开放，对修改关闭
class AbsCalculator
{
public:
	virtual int getresult
	{
		return 0;
	}
	//纯虚函数：virtual void getresult() = 0;
	int Num1;
	int Num2;
};
class Plus: public AbsCalculator
{
public:
	int getresult()
	{
		return Num1+Num2;
	}
};
class Sub: public AbsCalculator
{
public:
	int getresult()
	{
		return Num1-Num2;
	}
```

```
};
int main()
{
	AbsCalculator *abc = new Add;
	abc->Num1 = 20;
	abc->Num2 = 10;
	abc->getresult();//输出30

	AbsCalculator *abc = new Sub;
	abc->getresult();//输出10
}
```

八、文件操作

1.写文件

- ofstream：写
- ifstream：读
- fstream：读写

```
#include<fsteam>
ofsteam ofs;
ofs.open("FileAddress", openmode);
ofs << "DataToWrite" << endl;
ofs.close();
```

打开方式	解释
ios::in	为读文件而打开文件
ios::out	为写文件而打开文件
ios::ate	初始位置:文件尾
ios::app	追加方式写文件
ios::trunc	如果文件存在先删除，再创建
ios::binary	二进制方式

2.读文件

```
#include<fsteam>
ifsteam ifs;
ifs.open("FileAddress", openmode);
if(!ifs.is_open()) //判断文件是否打开
{
	cout << "Fail to Open" << endl;
	retuen;
}
//读数据，方式一
char buf[1024] = {0};
while( ifs >> buf)
{
	cout << buf << endl;
}
//读数据，方式二
char buf[1024] = {0};
while (ifs.getline(buf, sizeof(buf)))
{
	cout << buf <<endl;
}
//读数据，方式三
string buf;
while (getline(ifs, buf))
{
	cout << buf << endl;
}
//读数据，方式四
char c;
while((c = ifs.get) != EOF)//end if file，判断是否读完
{
	cout c;
}
ifs.close();
```

3.写文件-二进制

```
class Person
{
public:
	char Name[64];
	int Age;
};
int main()
{
	ofstream ofs;
	ofs.open("person.txt", ios::out||ios::binary);
	Person p = {"ZZP", 23};
	ofs.write((const char *)&p, sizeof(Person));
	ofs.close();
}
```

4.读文件-二进制

```
int main()
{
	ifstream ifs;
	ifs.open("person,txt", ios::in||ioos::binary);
	if(!ifs.is_open())
```

```
    {
        cout<<"打开失败"<<endl;
        return;
    }
    Person p;
    ifs.read((char *)&p, sizeof(Person));
    cout<<"姓名："<<p.Name<<"年龄："<<p.Age<<<endl;
}
```

九、模版

建立通用的模具，提高复用性；两种模板：**函数模板**和**类模板**

1.函数模版

```
template<typename T> // 也可以用<class T>
void Swap(T &a, T &b)
{
    T temp = a;
    a = b;
    b = temp;
}
int main()
{
    int a = 10;
    int b = 20;
    //第一种使用方式：自动推导
    Swap(a, b);
    //第二种使用方式：指定类型
    Swap<int>(a, b);
}
```

2.类模板

```
template<class NameType, class AgeType>
class Person
{
public:
    Person(NameType name, AgeType age)
    {
        p_name = name;
        p_age = age;
    }
    NameType p_name;
    AgeType p_age;
};
int main()
{
    Person<string, int>p("ZZP", 23);
}
```

3.函数模板与类模板区别

- 类模板没有自动类型推导使用方式

- 类模板在模板参数列表中可以有默认参数

```
template<class NameType, class AgeType = int>
int main()
{
	Person<string>p("ZZP", 23);
}
```