# 编程语言

参考链接：Github：https://github.com/huihut/interview

## C/C++

### 1.1 const

#### 1.1.1作用

1. 修饰变量，说明该变量不可以被改变；
2. 修饰指针，分为指向常量的指针和指针常量；
3. 常量引用，经常用于形参类型，即避免了拷贝，又避免了函数对值的修改；
4. 修饰成员函数，说明该成员函数内不能修改成员变量。

#### 1.1.2 const 使用

// 类class A

{private:

const int a; // 常对象成员，只能在初始化列表赋值

public:

// 构造函数

A() : a(0) { };

A(int x) : a(x) { }; // 初始化列表

// const可用于对重载函数的区分

int getValue(); // 普通成员函数

int getValue() const; // 常成员函数，不得修改类中的任何数据成员的值

};

void function()

{

// 对象

A b; // 普通对象，可以调用全部成员函数、更新常成员变量

const A a; // 常对象，只能调用常成员函数

const A \*p = &a; // 常指针

const A &q = a; // 常引用

// 指针

char greeting[] = "Hello";

char\* p1 = greeting; // 指针变量，指向字符数组变量

const char\* p2 = greeting; // 指针变量，指向字符数组常量

char\* const p3 = greeting; // 常指针，指向字符数组变量

const char\* const p4 = greeting; // 常指针，指向字符数组常量

}

// 函数

void function1(const int Var); // 传递过来的参数在函数内不可变

void function2(const char\* Var); // 参数指针所指内容为常量

void function3(char\* const Var); // 参数指针为常指针

void function4(const int& Var); // 引用参数在函数内为常量

// 函数返回值

const int function5(); // 返回一个常数

const int\* function6(); // 返回一个指向常量的指针变量，使用：const int \*p = function6();

int\* const function7(); // 返回一个指向变量的常指针，使用：int\* const p = function7();

### **1.2 static**

#### 1.2.1作用

1. 修饰普通变量，修改变量的存储区域和生命周期，使变量存储在静态区，在 main 函数运行前就分配了空间，如果有初始值就用初始值初始化它，如果没有初始值系统用默认值初始化它。
2. 修饰普通函数，表明函数的作用范围，仅在定义该函数的文件内才能使用。在多人开发项目时，为了防止与他人命名空间里的函数重名，可以将函数定位为 static。
3. 修饰成员变量，修饰成员变量使所有的对象只保存一个该变量，而且不需要生成对象就可以访问该成员。
4. 修饰成员函数，修饰成员函数使得不需要生成对象就可以访问该函数，但是在 static 函数内不能访问非静态成员。Static函数只能访问static成员。

### 1.3 this指针

#### 1.3.1 作用

1. this 指针是一个隐含于每一个非静态成员函数中的特殊指针。它指向调用该成员函数的那个对象。
2. 当对一个对象调用成员函数时，编译程序先将对象的地址赋给 this 指针，然后调用成员函数，每次成员函数存取数据成员时，都隐含使用 this 指针。
3. 当一个成员函数被调用时，自动向它传递一个隐含的参数，该参数是一个指向这个成员函数所在的对象的指针。
4. this 指针被隐含地声明为: ClassName \*const this，这意味着不能给 this 指针赋值；在 ClassName 类的 const 成员函数中，this 指针的类型为：const ClassName\* const，这说明不能对 this 指针所指向的这种对象是不可修改的（即不能对这种对象的数据成员进行赋值操作）；
5. this 并不是一个常规变量，而是个右值，所以不能取得 this 的地址（不能 &this）。
6. 在以下场景中，经常需要显式引用 this 指针：
7. 为实现对象的链式引用；
8. 为避免对同一对象进行赋值操作；
9. 在实现一些数据结构时，如 list。

### 1.4 inline 内联函数

#### 1.4.1 特征

1.相当于把内联函数里面的内容写在调用内联函数处；

2.相当于不用执行进入函数的步骤，直接执行函数体；

3.相当于宏，却比宏多了类型检查，真正具有函数特性；

4.编译器一般不内联包含循环、递归、switch 等复杂操作的内联函数；

5.在类声明中定义的函数，除了虚函数的其他函数都会自动隐式地当成内联函数。

#### 1.4.2 使用

// 声明1（加 inline，建议使用）

inline int functionName(int first, int second,...);

// 声明2（不加 inline）

int functionName(int first, int second,...);

// 定义

inline int functionName(int first, int second,...) {/\*\*\*\*/};

// 类内定义，隐式内联

class A {

int doA() { return 0; } // 隐式内联

}

// 类外定义，需要显式内联

class A {

int doA();

}

inline int A::doA() { return 0; } // 需要显式内联

#### 1.4.3 编译器对 inline 函数的处理步骤

1.将 inline 函数体复制到 inline 函数调用点处；

2.为所用 inline 函数中的局部变量分配内存空间；

3.将 inline 函数的的输入参数和返回值映射到调用方法的局部变量空间中；

4.如果 inline 函数有多个返回点，将其转变为 inline 函数代码块末尾的分支（使用 GOTO）。

#### 1.4.4 优缺点

优点

1. 内联函数同宏函数一样将在被调用处进行代码展开，省去了参数压栈、栈帧开辟与回收，结果返回等，从而提高程序运行速度。
2. 内联函数相比宏函数来说，在代码展开时，会做安全检查或自动类型转换（同普通函数），而宏定义则不会。
3. 在类中声明同时定义的成员函数，自动转化为内联函数，因此内联函数可以访问类的成员变量，宏定义则不能。
4. 内联函数在运行时可调试，而宏定义不可以。

缺点

1. 代码膨胀。内联是以代码膨胀（复制）为代价，消除函数调用带来的开销。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。
2. inline 函数无法随着函数库升级而升级。inline函数的改变需要重新编译，不像 non-inline 可以直接链接。
3. 是否内联，程序员不可控。内联函数只是对编译器的建议，是否对函数内联，决定权在于编译器。

#### 1.4.5 虚函数（virtual）可以是内联函数（inline）吗？

1.虚函数可以是内联函数，内联是可以修饰虚函数的，但是当虚函数表现多态性的时候不能内联。

2.内联是在编译器建议编译器内联，而虚函数的多态性在运行期，编译器无法知道运行期调用哪个代码，因此虚函数表现为多态性时（运行期）不可以内联。

3.inline virtual 唯一可以内联的时候是：编译器知道所调用的对象是哪个类（如 Base::who()），这只有在编译器具有实际对象而不是对象的指针或引用时才会发生。

#### 1.4.6 虚函数内联使用

#include <iostream> using namespace std;class Base

{public:

inline virtual void who()

{

cout << "I am Base\n";

}

virtual ~Base() {}

};class Derived : public Base

{public:

inline void who() // 不写inline时隐式内联

{

cout << "I am Derived\n";

}

};

int main()

{

// 此处的虚函数 who()，是通过类（Base）的具体对象（b）来调用的，编译期间就能确定了，所以它可以是内联的，但最终是否内联取决于编译器。

Base b;

b.who();

// 此处的虚函数是通过指针调用的，呈现多态性，需要在运行时期间才能确定，所以不能为内联。

Base \*ptr = new Derived();

ptr->who();

// 因为Base有虚析构函数（virtual ~Base() {}），所以 delete 时，会先调用派生类（Derived）析构函数，再调用基类（Base）析构函数，防止内存泄漏。

delete ptr;

ptr = nullptr;

system("pause");

return 0;

}

### **1.5 volatile**

volatile int i = 10;

* volatile 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素（操作系统、硬件、其它线程等）更改。所以使用 volatile 告诉编译器不应对这样的对象进行优化。
* volatile 关键字声明的变量，每次访问时都必须从内存中取出值（没有被 volatile 修饰的变量，可能由于编译器的优化，从 CPU 寄存器中取值）
* const 可以是 volatile （如只读的状态寄存器）
* 指针可以是 volatile

### 1.6 **assert()**

断言，是宏，而非函数。assert 宏的原型定义在 <assert.h>（C）、<cassert>（C++）中，其作用是如果它的条件返回错误，则终止程序执行。可以通过定义 NDEBUG 来关闭 assert，但是需要在源代码的开头，include <assert.h> 之前。

#define NDEBUG // 加上这行，则 assert 不可用

#include <assert.h>

assert( p != NULL ); // assert 不可用

### 1.7 **sizeof()**

* sizeof 对数组，得到整个数组所占空间大小。
* sizeof 对指针，得到指针本身所占空间大小。

### 1.8 **#pragma pack(n)**

#pragma pack(push) // 保存对齐状态

#pragma pack(4) // 设定为 4 字节对齐

struct test

{

char m1;

double m4;

int m3;

};

#pragma pack(pop) // 恢复对齐状态

### 1.9 **位域**

Bit mode: 2; // mode 占 2 位

类可以将其（非静态）数据成员定义为位域（bit-field），在一个位域中含有一定数量的二进制位。当一个程序需要向其他程序或硬件设备传递二进制数据时，通常会用到位域。

* 位域在内存中的布局是与机器有关的
* 位域的类型必须是整型或枚举类型，带符号类型中的位域的行为将因具体实现而定
* 取地址运算符（&）不能作用于位域，任何指针都无法指向类的位域

### 1.10 **extern "C"**

* 被 extern 限定的函数或变量是 extern 类型的
* 被 extern "C" 修饰的变量和函数是按照 C 语言方式编译和链接的

extern "C" 的作用是让 C++ 编译器将 extern "C" 声明的代码当作 C 语言代码处理，可以避免 C++ 因符号修饰导致代码不能和C语言库中的符号进行链接的问题。

#ifdef \_\_cplusplusextern "C" {

#endif

void \*memset(void \*, int, size\_t);

#ifdef \_\_cplusplus

}

#endif

### **1.11 struct 和 typedef struct**

#### 1.11.1 在C 中

// ctypedef struct Student {

int age;

} S;

等价于

// cstruct Student {

int age;

};

typedef struct Student S;

此时 S 等价于 struct Student，但两个标识符名称空间不相同。

另外还可以定义与 struct Student 不冲突的 void Student() {}。

#### 1.11.2 在**C++ 中**

由于编译器定位符号的规则（搜索规则）改变，导致不同于C语言。

一、如果在类标识符空间定义了 struct Student {...};，使用 Student me; 时，编译器将搜索全局标识符表，Student 未找到，则在类标识符内搜索。

即表现为可以使用 Student 也可以使用 struct Student，如下：

// cpp

struct Student {

int age;

};

void f( Student me ); // 正确，"struct" 关键字可省略

二、若定义了与 Student 同名函数之后，则 Student 只代表函数，不代表结构体，如下：

typedef struct Student {

int age;

} S;

void Student() {} // 正确，定义后 "Student" 只代表此函数

//void S() {} // 错误，符号 "S" 已经被定义为一个 "struct Student" 的别名

int main() {

Student();

struct Student me; // 或者 "S me";

return 0;

}

### 1.12 **C++ 中 struct 和 class**

总的来说，struct 更适合看成是一个数据结构的实现体，class 更适合看成是一个对象的实现体。

#### **区别**

* 最本质的一个区别就是默认的访问控制
  1. 默认的继承访问权限。struct 是 public 的，class 是 private 的。
  2. struct 作为数据结构的实现体，它默认的数据访问控制是 public 的，而 class 作为对象的实现体，它默认的成员变量访问控制是 private 的。

### **1.13 union 联合**

联合（union）是一种节省空间的特殊的类，一个 union 可以有多个数据成员，但是在任意时刻只有一个数据成员可以有值。当某个成员被赋值后其他成员变为未定义状态。联合有如下特点：

* 默认访问控制符为 public
* 可以含有构造函数、析构函数
* 不能含有引用类型的成员
* 不能继承自其他类，不能作为基类
* 不能含有虚函数
* 匿名 union 在定义所在作用域可直接访问 union 成员
* 匿名 union 不能包含 protected 成员或 private 成员
* 全局匿名联合必须是静态（static）的

#include<iostream>

union UnionTest {

UnionTest() : i(10) {};

int i;

double d;

};

static union {

int i;

double d;

};

int main() {

UnionTest u;

union {

int i;

double d;

};

std::cout << u.i << std::endl; // 输出 UnionTest 联合的 10

::i = 20;

std::cout << ::i << std::endl; // 输出全局静态匿名联合的 20

i = 30;

std::cout << i << std::endl; // 输出局部匿名联合的 30

return 0;

}

### 1.14 C++面向对象

C 实现 C++ 的面向对象特性（封装、继承、多态）

* 封装：使用函数指针把属性与方法封装到结构体中
* 继承：结构体嵌套
* 多态：父类与子类方法的函数指针不同

#### 封装

把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。关键字：public, protected, private。不写默认为 private。

* public 成员：可以被任意实体访问
* protected 成员：只允许被子类及本类的成员函数访问
* private 成员：只允许被本类的成员函数、友元类或友元函数访问

#### 继承

* 基类（父类）——> 派生类（子类）

#### 多态

* 多态，即多种状态（形态）。简单来说，我们可以将多态定义为消息以多种形式显示的能力。
* 多态是以封装和继承为基础的。
* C++ 多态分类及实现：
  1. 重载多态（Ad-hoc Polymorphism，编译期）：函数重载、运算符重载
  2. 子类型多态（Subtype Polymorphism，运行期）：虚函数
  3. 参数多态性（Parametric Polymorphism，编译期）：类模板、函数模板
  4. 强制多态（Coercion Polymorphism，编译期/运行期）：基本类型转换、自定义类型转换

#### **静态多态（编译期/早绑定）**

函数重载

class A

{public:

void do(int a);

void do(int a, int b);

};

#### **动态多态（运行期期/晚绑定）**

* 虚函数：用 virtual 修饰成员函数，使其成为虚函数

****注意：****

* 普通函数（非类成员函数）不能是虚函数
* 静态函数（static）不能是虚函数
* 构造函数不能是虚函数（因为在调用构造函数时，虚表指针并没有在对象的内存空间中，必须要构造函数调用完成后才会形成虚表指针）
* 内联函数不能是表现多态性时的虚函数，解释见：[虚函数（virtual）可以是内联函数（inline）吗？](https://github.com/huihut/interview" \l "%E8%99%9A%E5%87%BD%E6%95%B0virtual%E5%8F%AF%E4%BB%A5%E6%98%AF%E5%86%85%E8%81%94%E5%87%BD%E6%95%B0inline%E5%90%97)

class Shape // 形状类

{public:

virtual double calcArea()

{

...

}

virtual ~Shape();

};class Circle : public Shape // 圆形类

{public:

virtual double calcArea();

...

};class Rect : public Shape // 矩形类

{public:

virtual double calcArea();

...

};int main()

{

Shape \* shape1 = new Circle(4.0);

Shape \* shape2 = new Rect(5.0, 6.0);

shape1->calcArea(); // 调用圆形类里面的方法

shape2->calcArea(); // 调用矩形类里面的方法

delete shape1;

shape1 = nullptr;

delete shape2;

shape2 = nullptr;

return 0;

}

### 1.15 explicit（显式）关键字

* explicit 修饰构造函数时，可以防止隐式转换和复制初始化
* explicit 修饰转换函数时，可以防止隐式转换，但 [按语境转换](https://zh.cppreference.com/w/cpp/language/implicit_conversion) 除外

struct A

{

A(int) { }

operator bool() const { return true; }

};

struct B

{

explicit B(int) {}

explicit operator bool() const { return true; }

};

void doA(A a) {}

void doB(B b) {}

int main()

{

A a1(1); // OK：直接初始化

A a2 = 1; // OK：复制初始化

A a3{ 1 }; // OK：直接列表初始化

A a4 = { 1 }; // OK：复制列表初始化

A a5 = (A)1; // OK：允许 static\_cast 的显式转换

doA(1); // OK：允许从 int 到 A 的隐式转换

if (a1); // OK：使用转换函数 A::operator bool() 的从 A 到 bool 的隐式转换

bool a6（a1）;// OK：使用转换函数 A::operator bool() 的从 A 到 bool 的隐式转换

bool a7 = a1;// OK：使用转换函数 A::operator bool() 的从 A 到 bool 的隐式转换

bool a8 = static\_cast<bool>(a1); // OK ：static\_cast 进行直接初始化

B b1(1); // OK：直接初始化

B b2 = 1; // 错误：被 explicit 修饰构造函数的对象不可以复制初始化

B b3{ 1 }; // OK：直接列表初始化

B b4 = { 1 }; // 错误：被 explicit 修饰构造函数的对象不可以复制列表初始化

B b5 = (B)1; // OK：允许 static\_cast 的显式转换

doB(1); // 错误：被 explicit 修饰构造函数的对象不可以从 int 到 B 的隐式转换

if (b1); // OK：被 explicit 修饰转换函数 B::operator bool() 的对象可以从 B 到 b ool 的按语境转换

bool b6(b1); // OK：被 explicit 修饰转换函数 B::operator bool() 的对象可以从 B 到 bool 的按语境转换

bool b7 = b1; // 错误：被 explicit 修饰转换函数 B::operator bool() 的对象不可以隐式转换

bool b8 = static\_cast<bool>(b1); // OK：static\_cast 进行直接初始化

return 0;

}

### 1.16 friend 友元类和友元函数

* 能访问私有成员
* 破坏封装性
* 友元关系不可传递
* 友元关系的单向性
* 友元声明的形式及数量不受限制

### 1.17 **using 声明**

一条 using 声明 语句一次只引入命名空间的一个成员。它使得我们可以清楚知道程序中所引用的到底是哪个名字。如：

using namespace\_name::name;

### 1.18 **构造函数的 using 声明**

在 C++11 中，派生类能够重用其直接基类定义的构造函数。

using namespace\_name name;

尽量少使用 using 指示 污染命名空间

一般说来，使用 using 命令比使用 using 编译命令更安全，这是由于它****只导入了指定的名称****。如果该名称与局部名称发生冲突，编译器将****发出指示****。using编译命令导入所有的名称，包括可能并不需要的名称。如果与局部名称发生冲突，则****局部名称将覆盖名称空间版本****，而编译器****并不会发出警告****。另外，名称空间的开放性意味着名称空间的名称可能分散在多个地方，这使得难以准确知道添加了哪些名称。

尽量少使用 using 指示

using namespace std;

应该多使用 using 声明

int x;

std::cin >> x ;

std::cout << x << std::endl;

或者

using std::cin;

using std::cout;

using std::endl;

int x;

cin >> x;

cout << x << endl;

### 1.19 :: 范围解析运算符

#### **分类**

1. 全局作用域符（::name）：用于类型名称（类、类成员、成员函数、变量等）前，表示作用域为全局命名空间
2. 类作用域符（class::name）：用于表示指定类型的作用域范围是具体某个类的
3. 命名空间作用域符（namespace::name）:用于表示指定类型的作用域范围是具体某个命名空间的

int count = 0; // 全局（::）的 count

class A {public:

static int count; // 类 A 的 count（A::count）

};

int main() {

::count = 1; // 设置全局的 count 的值为 1

A::count = 2; // 设置类 A 的 count 为 2

int count = 0; // 局部的 count

count = 3; // 设置局部的 count 的值为 3

return 0;

}

### 1.20 enum 枚举类型

#### **限定作用域的枚举类型**

enum class open\_modes { input, output, append };

#### **不限定作用域的枚举类型**

enum color { red, yellow, green };enum { floatPrec = 6, doublePrec = 10 };

### 1.21 **decltype**

decltype 关键字用于检查实体的声明类型或表达式的类型及值分类。语法：

// 尾置返回允许我们在参数列表之后声明返回类型

template <typename It>auto fcn(It beg, It end) -> decltype(\*beg)

{

// 处理序列

return \*beg; // 返回序列中一个元素的引用

}// 为了使用模板参数成员，必须用 typenametemplate <typename It>auto fcn2(It beg, It end) -> typename remove\_reference<decltype(\*beg)>::type

{

// 处理序列

return \*beg; // 返回序列中一个元素的拷贝

}

### 1.22 引用

#### **左值引用**

常规引用，一般表示对象的身份。

#### **右值引用**

右值引用就是必须绑定到右值（一个临时对象、将要销毁的对象）的引用，一般表示对象的值。

右值引用可实现转移语义（Move Sementics）和精确传递（Perfect Forwarding），它的主要目的有两个方面：

* 消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率。
* 能够更简洁明确地定义泛型函数。

#### **引用折叠**

* X& &、X& &&、X&& & 可折叠成 X&
* X&& && 可折叠成 X&&

### 1.23 宏

* 宏定义可以实现类似于函数的功能，但是它终归不是函数，而宏定义中括弧中的“参数”也不是真的参数，在宏展开的时候对 “参数” 进行的是一对一的替换。

### 1.24 成员初始化列表

好处

* 更高效：少了一次调用默认构造函数的过程。
* 有些场合必须要用初始化列表：
  1. 常量成员，因为常量只能初始化不能赋值，所以必须放在初始化列表里面
  2. 引用类型，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面
  3. 没有默认构造函数的类类型，因为使用初始化列表可以不必调用默认构造函数来初始化

initializer\_list 列表初始化

用花括号初始化器列表初始化一个对象，其中对应构造函数接受一个 std::initializer\_list 参数.

#include <iostream>

#include <vector>

#include <initializer\_list>

template <class T>struct S {

std::vector<T> v;

S(std::initializer\_list<T> l) : v(l) {

std::cout << "constructed with a " << l.size() << "-element list\n";

}

void append(std::initializer\_list<T> l) {

v.insert(v.end(), l.begin(), l.end());

}

std::pair<const T\*, std::size\_t> c\_arr() const {

return {&v[0], v.size()}; // 在 return 语句中复制列表初始化

// 这不使用 std::initializer\_list

}

};

template <typename T>void templated\_fn(T) {}

int main()

{

S<int> s = {1, 2, 3, 4, 5}; // 复制初始化

s.append({6, 7, 8}); // 函数调用中的列表初始化

std::cout << "The vector size is now " << s.c\_arr().second << " ints:\n";

for (auto n : s.v)

std::cout << n << ' ';

std::cout << '\n';

std::cout << "Range-for over brace-init-list: \n";

for (int x : {-1, -2, -3}) // auto 的规则令此带范围 for 工作

std::cout << x << ' ';

std::cout << '\n';

auto al = {10, 11, 12}; // auto 的特殊规则

std::cout << "The list bound to auto has size() = " << al.size() << '\n';

// templated\_fn({1, 2, 3}); // 编译错误！“ {1, 2, 3} ”不是表达式，

// 它无类型，故 T 无法推导

templated\_fn<std::initializer\_list<int>>({1, 2, 3}); // OK

templated\_fn<std::vector<int>>({1, 2, 3}); // 也 OK

}

### 1.25 虚函数

#### **1.25.1虚函数**

动态多态（运行期期/晚绑定）

虚函数：用 virtual 修饰成员函数，使其成为虚函数

****注意：****

* 普通函数（非类成员函数）不能是虚函数
* 静态函数（static）不能是虚函数
* 构造函数不能是虚函数（因为在调用构造函数时，虚表指针并没有在对象的内存空间中，必须要构造函数调用完成后才会形成虚表指针）
* 内联函数不能是表现多态性时的虚函数，解释见：[虚函数（virtual）可以是内联函数（inline）吗？](https://github.com/huihut/interview" \l "%E8%99%9A%E5%87%BD%E6%95%B0virtual%E5%8F%AF%E4%BB%A5%E6%98%AF%E5%86%85%E8%81%94%E5%87%BD%E6%95%B0inline%E5%90%97)

class Shape // 形状类

{public:

virtual double calcArea()

{

...

}

virtual ~Shape();

};class Circle : public Shape // 圆形类

{public:

virtual double calcArea();

...

};class Rect : public Shape // 矩形类

{public:

virtual double calcArea();

...

};int main()

{

Shape \* shape1 = new Circle(4.0);

Shape \* shape2 = new Rect(5.0, 6.0);

shape1->calcArea(); // 调用圆形类里面的方法

shape2->calcArea(); // 调用矩形类里面的方法

delete shape1;

shape1 = nullptr;

delete shape2;

shape2 = nullptr;

return 0;

}

#### **1.25.2虚析构函数**

虚析构函数是为了解决基类的指针指向派生类对象，并用基类的指针删除派生类对象。

class Shape

{public:

Shape(); // 构造函数不能是虚函数

virtual double calcArea();

virtual ~Shape(); // 虚析构函数

};class Circle : public Shape // 圆形类

{public:

virtual double calcArea();

...

};int main()

{

Shape \* shape1 = new Circle(4.0);

shape1->calcArea();

delete shape1; // 因为Shape有虚析构函数，所以delete释放内存时，先调用子类析构函数，再调用基类析构函数，防止内存泄漏。

shape1 = NULL;

return 0；

}

#### 1.25.3 纯虚函数

纯虚函数是一种特殊的虚函数，在基类中不能对虚函数给出有意义的实现，而把它声明为纯虚函数，它的实现留给该基类的派生类去做。

virtual int A() = 0;

#### 1.25.4 虚函数、纯虚函数

* 类里如果声明了虚函数，这个函数是实现的，哪怕是空实现，它的作用就是为了能让这个函数在它的子类里面可以被覆盖，这样的话，编译器就可以使用后期绑定来达到多态了。纯虚函数只是一个接口，是个函数的声明而已，它要留到子类里去实现。
* 虚函数在子类里面也可以不重载的；但纯虚函数必须在子类去实现。
* 虚函数的类用于 “实作继承”，继承接口的同时也继承了父类的实现。当然大家也可以完成自己的实现。纯虚函数关注的是接口的统一性，实现由子类完成。
* 带纯虚函数的类叫抽象类，这种类不能直接生成对象，而只有被继承，并重写其虚函数后，才能使用。抽象类被继承后，子类可以继续是抽象类，也可以是普通类。
* 虚基类是虚继承中的基类，具体见下文虚继承。https://blog.csdn.net/u012260238/article/details/53610462

#### 1.25.5 虚函数指针、虚函数表

* 虚函数指针：在含有虚函数类的对象中，指向虚函数表，在运行时确定。
* 虚函数表：在程序只读数据段（.rodata section，见：[目标文件存储结构](https://github.com/huihut/interview" \l "%E7%9B%AE%E6%A0%87%E6%96%87%E4%BB%B6%E5%AD%98%E5%82%A8%E7%BB%93%E6%9E%84)），存放虚函数指针，如果派生类实现了基类的某个虚函数，则在虚表中覆盖原本基类的那个虚函数指针，在编译时根据类的声明创建。

#### 1.25.6 虚继承

虚继承用于解决多继承条件下的菱形继承问题（浪费存储空间、存在二义性）。

底层实现原理与编译器相关，一般通过****虚基类指针****和****虚基类表****实现，每个虚继承的子类都有一个虚基类指针（占用一个指针的存储空间，4字节）和虚基类表（不占用类对象的存储空间）（需要强调的是，虚基类依旧会在子类里面存在拷贝，只是仅仅最多存在一份而已，并不是不在子类里面了）；当虚继承的子类被当做父类继承时，虚基类指针也会被继承。

实际上，vbptr 指的是虚基类表指针（virtual base table pointer），该指针指向了一个虚基类表（virtual table），虚表中记录了虚基类与本类的偏移地址；通过偏移地址，这样就找到了虚基类成员，而虚继承也不用像普通多继承那样维持着公共基类（虚基类）的两份同样的拷贝，节省了存储空间。

#### 1.25.7虚继承、虚函数

* 相同之处：都利用了虚指针（均占用类的存储空间）和虚表（均不占用类的存储空间）
* 不同之处：
  + 虚继承
    - 虚基类依旧存在继承类中，只占用存储空间
    - 虚基类表存储的是虚基类相对直接继承类的偏移
  + 虚函数
    - 虚函数不占用存储空间
    - 虚函数表存储的是虚函数地址

#### 1.25.8 模板类、成员模板、虚函数

* 模板类中可以使用虚函数
* 一个类（无论是普通类还是类模板）的成员模板（本身是模板的成员函数）不能是虚函数

#### 1.25.9 抽象类、接口类、聚合类

* 抽象类：含有纯虚函数的类
* 接口类：仅含有纯虚函数的抽象类
* 聚合类：用户可以直接访问其成员，并且具有特殊的初始化语法形式。满足如下特点：
  + 所有成员都是 public
  + 没有定义任何构造函数
  + 没有类内初始化
  + 没有基类，也没有 virtual 函数

### 1.26 **内存分配和管理**

#### 1.26.1 malloc、calloc、realloc、alloca

1. malloc：申请指定字节数的内存。申请到的内存中的初始值不确定。
2. calloc：为指定长度的对象，分配能容纳其指定个数的内存。申请到的内存的每一位（bit）都初始化为 0。
3. realloc：更改以前分配的内存长度（增加或减少）。当增加长度时，可能需将以前分配区的内容移到另一个足够大的区域，而新增区域内的初始值则不确定。
4. alloca：在栈上申请内存。程序在出栈的时候，会自动释放内存。但是需要注意的是，alloca 不具可移植性, 而且在没有传统堆栈的机器上很难实现。alloca 不宜使用在必须广泛移植的程序中。C99 中支持变长数组 (VLA)，可以用来替代 alloca。

#### 1.26.2 malloc、free

用于分配、释放内存

char \*str = (char\*) malloc(100);

assert(str != nullptr);

free(p);

p = nullptr;

#### 1.26.3 new、delete

1. new / new[]：完成两件事，先底层调用 malloc 分配了内存，然后调用构造函数（创建对象）。
2. delete/delete[]：也完成两件事，先调用析构函数（清理资源），然后底层调用 free 释放空间。
3. new 在申请内存时会自动计算所需字节数，而 malloc 则需我们自己输入申请内存空间的字节数。

int main()

{

T\* t = new T(); // 先内存分配 ，再构造函数

delete t; // 先析构函数，再内存释放

return 0;

}

#### 1.26.4 定位 new

定位 new（placement new）允许我们向 new 传递额外的地址参数，从而在预先指定的内存区域创建对象。

new (place\_address) typenew (place\_address) type (initializers)new (place\_address) type [size]new (place\_address) type [size] { braced initializer list }

* place\_address 是个指针
* initializers 提供一个（可能为空的）以逗号分隔的初始值列表

delete this 合法吗？合法，但：

1. 必须保证 this 对象是通过 new（不是 new[]、不是 placement new、不是栈上、不是全局、不是其他对象成员）分配的
2. 必须保证调用 delete this 的成员函数是最后一个调用 this 的成员函数
3. 必须保证成员函数的 delete this 后面没有调用 this 了
4. 必须保证 delete this 后没有人使用了

### 1.27 **在堆上（栈上）生成对象的类**

#### **只能在堆上**

方法：将析构函数设置为私有

原因：C++ 是静态绑定语言，编译器管理栈上对象的生命周期，编译器在为类对象分配栈空间时，会先检查类的析构函数的访问性。若析构函数不可访问，则不能在栈上创建对象。

#### **只能在栈上**

方法：将 new 和 delete 重载为私有

原因：在堆上生成对象，使用 new 关键词操作，其过程分为两阶段：第一阶段，使用 new 在堆上寻找可用内存，分配给对象；第二阶段，调用构造函数生成对象。将 new 操作设置为私有，那么第一阶段就无法完成，就不能够在堆上生成对象。

### 1.28 智能指针

#### **C++ 标准库（STL）中**

头文件：#include <memory>

#### **C++ 98**

std::auto\_ptr<std::string> ps (new std::string(str))；

#### **C++ 11**

1. shared\_ptr
2. unique\_ptr
3. weak\_ptr
4. auto\_ptr（被 C++11 弃用）

* Class shared\_ptr 实现共享式拥有（shared ownership）概念。多个智能指针指向相同对象，该对象和其相关资源会在 “最后一个 reference 被销毁” 时被释放。为了在结构较复杂的情景中执行上述工作，标准库提供 weak\_ptr、bad\_weak\_ptr 和 enable\_shared\_from\_this 等辅助类。
* Class unique\_ptr 实现独占式拥有（exclusive ownership）或严格拥有（strict ownership）概念，保证同一时间内只有一个智能指针可以指向该对象。你可以移交拥有权。它对于避免内存泄漏（resource leak）——如 new 后忘记 delete ——特别有用。

##### **shared\_ptr**

多个智能指针可以共享同一个对象，对象的最末一个拥有着有责任销毁对象，并清理与该对象相关的所有资源。

* 支持定制型删除器（custom deleter），可防范 Cross-DLL 问题（对象在动态链接库（DLL）中被 new 创建，却在另一个 DLL 内被 delete 销毁）、自动解除互斥锁

##### **weak\_ptr**

weak\_ptr 允许你共享但不拥有某对象，一旦最末一个拥有该对象的智能指针失去了所有权，任何 weak\_ptr 都会自动成空（empty）。因此，在 default 和 copy 构造函数之外，weak\_ptr 只提供 “接受一个 shared\_ptr” 的构造函数。

* 可打破环状引用（cycles of references，两个其实已经没有被使用的对象彼此互指，使之看似还在 “被使用” 的状态）的问题

##### **unique\_ptr**

unique\_ptr 是 C++11 才开始提供的类型，是一种在异常时可以帮助避免资源泄漏的智能指针。采用独占式拥有，意味着可以确保一个对象和其相应的资源同一时间只被一个 pointer 拥有。一旦拥有着被销毁或编程 empty，或开始拥有另一个对象，先前拥有的那个对象就会被销毁，其任何相应资源亦会被释放。

* unique\_ptr 用于取代 auto\_ptr

##### **auto\_ptr**

被 c++11 弃用，原因是缺乏语言特性如 “针对构造和赋值” 的 std::move 语义，以及其他瑕疵。

##### **auto\_ptr 与 unique\_ptr 比较**

* auto\_ptr 可以赋值拷贝，复制拷贝后所有权转移；unqiue\_ptr 无拷贝赋值语义，但实现了move 语义；
* auto\_ptr 对象不能管理数组（析构调用 delete），unique\_ptr 可以管理数组（析构调用 delete[] ）；

### **1.29强制类型转换运算符**

#### **static\_cast**

* 用于非多态类型的转换
* 不执行运行时类型检查（转换安全性不如 dynamic\_cast）
* 通常用于转换数值数据类型（如 float -> int）
* 可以在整个类层次结构中移动指针，子类转化为父类安全（向上转换），父类转化为子类不安全（因为子类可能有不在父类的字段或方法）

向上转换是一种隐式转换。

#### **dynamic\_cast**

* 用于多态类型的转换
* 执行行运行时类型检查
* 只适用于指针或引用
* 对不明确的指针的转换将失败（返回 nullptr），但不引发异常
* 可以在整个类层次结构中移动指针，包括向上转换、向下转换

#### **const\_cast**

* 用于删除 const、volatile 和 \_\_unaligned 特性（如将 const int 类型转换为 int 类型 ）

#### **reinterpret\_cast**

* 用于位的简单重新解释
* 滥用 reinterpret\_cast 运算符可能很容易带来风险。 除非所需转换本身是低级别的，否则应使用其他强制转换运算符之一。
* 允许将任何指针转换为任何其他指针类型（如 char\* 到 int\* 或 One\_class\* 到 Unrelated\_class\* 之类的转换，但其本身并不安全）
* 也允许将任何整数类型转换为任何指针类型以及反向转换。
* reinterpret\_cast 运算符不能丢掉 const、volatile 或 \_\_unaligned 特性。
* reinterpret\_cast 的一个实际用途是在哈希函数中，即，通过让两个不同的值几乎不以相同的索引结尾的方式将值映射到索引。

#### **bad\_cast**

* 由于强制转换为引用类型失败，dynamic\_cast 运算符引发 bad\_cast 异常。

try {

Circle& ref\_circle = dynamic\_cast<Circle&>(ref\_shape);

} catch (bad\_cast b) {

cout << "Caught: " << b.what();

}

### 1.30运行时类型信息 (RTTI)

#### **dynamic\_cast**

* 用于多态类型的转换

#### **typeid**

* typeid 运算符允许在运行时确定对象的类型
* type\_id 返回一个 type\_info 对象的引用
* 如果想通过基类的指针获得派生类的数据类型，基类必须带有虚函数
* 只能获取对象的实际类型

#### **type\_info**

* type\_info 类描述编译器在程序中生成的类型信息。 此类的对象可以有效存储指向类型的名称的指针。 type\_info 类还可存储适合比较两个类型是否相等或比较其排列顺序的编码值。 类型的编码规则和排列顺序是未指定的，并且可能因程序而异。
* 头文件：typeinfo

class Flyable // 能飞的

{public:

virtual void takeoff() = 0; // 起飞

virtual void land() = 0; // 降落

};class Bird : public Flyable // 鸟

{public:

void foraging() {...} // 觅食

virtual void takeoff() {...}

virtual void land() {...}

};class Plane : public Flyable // 飞机

{public:

void carry() {...} // 运输

virtual void take off() {...}

virtual void land() {...}

};

class type\_info

{public:

const char\* name() const;

bool operator == (const type\_info & rhs) const;

bool operator != (const type\_info & rhs) const;

int before(const type\_info & rhs) const;

virtual ~type\_info();private:

...

};

class doSomething(Flyable \*obj) // 做些事情

{

obj->takeoff();

cout << typeid(\*obj).name() << endl; // 输出传入对象类型（"class Bird" or "class Plane"）

if(typeid(\*obj) == typeid(Bird)) // 判断对象类型

{

Bird \*bird = dynamic\_cast<Bird \*>(obj); // 对象转化

bird->foraging();

}

obj->land();

};

### 1.31 **Effective C++**

1. 视 C++ 为一个语言联邦（C、Object-Oriented C++、Template C++、STL）
2. 宁可以编译器替换预处理器（尽量以 const、enum、inline 替换 #define）
3. 尽可能使用 const
4. 确定对象被使用前已先被初始化（构造时赋值（copy 构造函数）比 default 构造后赋值（copy assignment）效率高）
5. 了解 C++ 默默编写并调用哪些函数（编译器暗自为 class 创建 default 构造函数、copy 构造函数、copy assignment 操作符、析构函数）
6. 若不想使用编译器自动生成的函数，就应该明确拒绝（将不想使用的成员函数声明为 private，并且不予实现）
7. 为多态基类声明 virtual 析构函数（如果 class 带有任何 virtual 函数，它就应该拥有一个 virtual 析构函数）
8. 别让异常逃离析构函数（析构函数应该吞下不传播异常，或者结束程序，而不是吐出异常；如果要处理异常应该在非析构的普通函数处理）
9. 绝不在构造和析构过程中调用 virtual 函数（因为这类调用从不下降至 derived class）
10. 令 operator= 返回一个 reference to \*this （用于连锁赋值）
11. 在 operator= 中处理 “自我赋值”
12. 赋值对象时应确保复制 “对象内的所有成员变量” 及 “所有 base class 成分”（调用基类复制构造函数）
13. 以对象管理资源（资源在构造函数获得，在析构函数释放，建议使用智能指针，资源取得时机便是初始化时机（Resource Acquisition Is Initialization，RAII））
14. 在资源管理类中小心 copying 行为（普遍的 RAII class copying 行为是：抑制 copying、引用计数、深度拷贝、转移底部资源拥有权（类似 auto\_ptr））
15. 在资源管理类中提供对原始资源（raw resources）的访问（对原始资源的访问可能经过显式转换或隐式转换，一般而言显示转换比较安全，隐式转换对客户比较方便）
16. 成对使用 new 和 delete 时要采取相同形式（new 中使用 [] 则 delete []，new 中不使用 [] 则 delete）
17. 以独立语句将 newed 对象存储于（置入）智能指针（如果不这样做，可能会因为编译器优化，导致难以察觉的资源泄漏）
18. 让接口容易被正确使用，不易被误用（促进正常使用的办法：接口的一致性、内置类型的行为兼容；阻止误用的办法：建立新类型，限制类型上的操作，约束对象值、消除客户的资源管理责任）
19. 设计 class 犹如设计 type，需要考虑对象创建、销毁、初始化、赋值、值传递、合法值、继承关系、转换、一般化等等。
20. 宁以 pass-by-reference-to-const 替换 pass-by-value （前者通常更高效、避免切割问题（slicing problem），但不适用于内置类型、STL迭代器、函数对象）
21. 必须返回对象时，别妄想返回其 reference（绝不返回 pointer 或 reference 指向一个 local stack 对象，或返回 reference 指向一个 heap-allocated 对象，或返回 pointer 或 reference 指向一个 local static 对象而有可能同时需要多个这样的对象。）
22. 将成员变量声明为 private（为了封装、一致性、对其读写精确控制等）
23. 宁以 non-member、non-friend 替换 member 函数（可增加封装性、包裹弹性（packaging flexibility）、机能扩充性）
24. 若所有参数（包括被this指针所指的那个隐喻参数）皆须要类型转换，请为此采用 non-member 函数
25. 考虑写一个不抛异常的 swap 函数
26. 尽可能延后变量定义式的出现时间（可增加程序清晰度并改善程序效率）
27. 尽量少做转型动作（旧式：(T)expression、T(expression)；新式：const\_cast<T>(expression)、dynamic\_cast<T>(expression)、reinterpret\_cast<T>(expression)、static\_cast<T>(expression)、；尽量避免转型、注重效率避免 dynamic\_casts、尽量设计成无需转型、可把转型封装成函数、宁可用新式转型）
28. 避免使用 handles（包括 引用、指针、迭代器）指向对象内部（以增加封装性、使 const 成员函数的行为更像 const、降低 “虚吊号码牌”（dangling handles，如悬空指针等）的可能性）
29. 为 “异常安全” 而努力是值得的（异常安全函数（Exception-safe functions）即使发生异常也不会泄露资源或允许任何数据结构败坏，分为三种可能的保证：基本型、强列型、不抛异常型）
30. 透彻了解 inlining 的里里外外（inlining 在大多数 C++ 程序中是编译期的行为；inline 函数是否真正 inline，取决于编译器；大部分编译器拒绝太过复杂（如带有循环或递归）的函数 inlining，而所有对 virtual 函数的调用（除非是最平淡无奇的）也都会使 inlining 落空；inline 造成的代码膨胀可能带来效率损失；inline 函数无法随着程序库的升级而升级）
31. 将文件间的编译依存关系降至最低（如果使用 object references 或 object pointers 可以完成任务，就不要使用 objects；如果能过够，尽量以 class 声明式替换 class 定义式；为声明式和定义式提供不同的头文件）
32. 确定你的 public 继承塑模出 is-a（是一种）关系（适用于 base classes 身上的每一件事情一定适用于 derived classes 身上，因为每一个 derived class 对象也都是一个 base class 对象）
33. 避免遮掩继承而来的名字（可使用 using 声明式或转交函数（forwarding functions）来让被遮掩的名字再见天日）
34. 区分接口继承和实现继承（在 public 继承之下，derived classes 总是继承 base class 的接口；pure virtual 函数只具体指定接口继承；非纯 impure virtual 函数具体指定接口继承及缺省实现继承；non-virtual 函数具体指定接口继承以及强制性实现继承）
35. 考虑 virtual 函数以外的其他选择（如 Template Method 设计模式的 non-virtual interface（NVI）手法，将 virtual 函数替换为 “函数指针成员变量”，以 tr1::function 成员变量替换 virtual 函数，将继承体系内的 virtual 函数替换为另一个继承体系内的 virtual 函数）
36. 绝不重新定义继承而来的 non-virtual 函数
37. 绝不重新定义继承而来的缺省参数值，因为缺省参数值是静态绑定（statically bound），而 virtual 函数却是动态绑定（dynamically bound）
38. 通过复合塑模 has-a（有一个）或 “根据某物实现出”（在应用域（application domain），复合意味 has-a（有一个）；在实现域（implementation domain），复合意味着 is-implemented-in-terms-of（根据某物实现出））
39. 明智而审慎地使用 private 继承（private 继承意味着 is-implemented-in-terms-of（根据某物实现出），尽可能使用复合，当 derived class 需要访问 protected base class 的成员，或需要重新定义继承而来的时候 virtual 函数，或需要 empty base 最优化时，才使用 private 继承）
40. 明智而审慎地使用多重继承（多继承比单一继承复杂，可能导致新的歧义性，以及对 virtual 继承的需要，但确有正当用途，如 “public 继承某个 interface class” 和 “private 继承某个协助实现的 class”；virtual 继承可解决多继承下菱形继承的二义性问题，但会增加大小、速度、初始化及赋值的复杂度等等成本）
41. 了解隐式接口和编译期多态（class 和 templates 都支持接口（interfaces）和多态（polymorphism）；class 的接口是以签名为中心的显式的（explicit），多态则是通过 virtual 函数发生于运行期；template 的接口是奠基于有效表达式的隐式的（implicit），多态则是通过 template 具现化和函数重载解析（function overloading resolution）发生于编译期）
42. 了解 typename 的双重意义（声明 template 类型参数是，前缀关键字 class 和 typename 的意义完全相同；请使用关键字 typename 标识嵌套从属类型名称，但不得在基类列（base class lists）或成员初值列（member initialization list）内以它作为 base class 修饰符）
43. 学习处理模板化基类内的名称（可在 derived class templates 内通过 this-> 指涉 base class templates 内的成员名称，或藉由一个明白写出的 “base class 资格修饰符” 完成）
44. 将与参数无关的代码抽离 templates（因类型模板参数（non-type template parameters）而造成代码膨胀往往可以通过函数参数或 class 成员变量替换 template 参数来消除；因类型参数（type parameters）而造成的代码膨胀往往可以通过让带有完全相同二进制表述（binary representations）的实现类型（instantiation types）共享实现码）
45. 运用成员函数模板接受所有兼容类型（请使用成员函数模板（member function templates）生成 “可接受所有兼容类型” 的函数；声明 member templates 用于 “泛化 copy 构造” 或 “泛化 assignment 操作” 时还需要声明正常的 copy 构造函数和 copy assignment 操作符）
46. 需要类型转换时请为模板定义非成员函数（当我们编写一个 class template，而它所提供之 “与此 template 相关的” 函数支持 “所有参数之隐式类型转换” 时，请将那些函数定义为 “class template 内部的 friend 函数”）
47. 请使用 traits classes 表现类型信息（traits classes 通过 templates 和 “templates 特化” 使得 “类型相关信息” 在编译期可用，通过重载技术（overloading）实现在编译期对类型执行 if...else 测试）
48. 认识 template 元编程（模板元编程（TMP，template metaprogramming）可将工作由运行期移往编译期，因此得以实现早期错误侦测和更高的执行效率；TMP 可被用来生成 “给予政策选择组合”（based on combinations of policy choices）的客户定制代码，也可用来避免生成对某些特殊类型并不适合的代码）
49. 了解 new-handler 的行为（set\_new\_handler 允许客户指定一个在内存分配无法获得满足时被调用的函数；nothrow new 是一个颇具局限的工具，因为它只适用于内存分配（operator new），后继的构造函数调用还是可能抛出异常）
50. 了解 new 和 delete 的合理替换时机（为了检测运用错误、收集动态分配内存之使用统计信息、增加分配和归还速度、降低缺省内存管理器带来的空间额外开销、弥补缺省分配器中的非最佳齐位、将相关对象成簇集中、获得非传统的行为）
51. 编写 new 和 delete 时需固守常规（operator new 应该内涵一个无穷循环，并在其中尝试分配内存，如果它无法满足内存需求，就应该调用 new-handler，它也应该有能力处理 0 bytes 申请，class 专属版本则还应该处理 “比正确大小更大的（错误）申请”；operator delete 应该在收到 null 指针时不做任何事，class 专属版本则还应该处理 “比正确大小更大的（错误）申请”）
52. 写了 placement new 也要写 placement delete（当你写一个 placement operator new，请确定也写出了对应的 placement operator delete，否则可能会发生隐微而时断时续的内存泄漏；当你声明 placement new 和 placement delete，请确定不要无意识（非故意）地遮掩了它们地正常版本）
53. 不要轻忽编译器的警告
54. 让自己熟悉包括 TR1 在内的标准程序库（TR1，C++ Technical Report 1，C++11 标准的草稿文件）
55. 让自己熟悉 Boost（准标准库）

### **1.32 Google C++ Style Guide**

英文：Google C++ Style Guide

https://google.github.io/styleguide/cppguide.html

中文：C++ 风格指南

<https://zh-google-styleguide.readthedocs.io/en/latest/google-cpp-styleguide/contents/>



<https://blog.csdn.net/voidccc/article/details/37599203>

## STL

### 2.1参考链接

网站：<https://github.com/huihut/interview/blob/master/STL/STL.md>

### 2.2组成

* 容器（containers）
* 算法（algorithms）
* 迭代器（iterators）
* 仿函数（functors）
* 配接器（adapters）
* 空间配置器（allocator）

### 2.3 容器（containers）

array

array 是固定大小的顺序容器，它们保存了一个以严格的线性顺序排列的特定数量的元素。

Vector

vector 是表示可以改变大小的数组的序列容器。

deque

deque（['dek]）（双端队列）是double-ended queue 的一个不规则缩写。deque是具有动态大小的序列容器，可以在两端（前端或后端）扩展或收缩。

forward\_list

forward\_list（单向链表）是序列容器，允许在序列中的任何地方进行恒定的时间插入和擦除操作。

list

list，双向链表，是序列容器，允许在序列中的任何地方进行常数时间插入和擦除操作，并在两个方向上进行迭代。

stack

stack 是一种容器适配器，用于在LIFO（后进先出）的操作，其中元素仅从容器的一端插入和提取。

queue

queue 是一种容器适配器，用于在FIFO（先入先出）的操作，其中元素插入到容器的一端并从另一端提取。

priority\_queue

set

set 是按照特定顺序存储唯一元素的容器。

multiset

map

map 是关联容器，按照特定顺序存储由 key value (键值) 和 mapped value (映射值) 组合形成的元素。

#### STL 容器

| 容器 | 底层数据结构 | 时间复杂度 | 有无序 | 可不可重复 | 其他 |
| --- | --- | --- | --- | --- | --- |
| [array](https://github.com/huihut/interview/tree/master/STL" \l "array) | 数组 | 随机读改 O(1) | 无序 | 可重复 | 支持快速随机访问 |
| [vector](https://github.com/huihut/interview/tree/master/STL" \l "vector) | 数组 | 随机读改、尾部插入、尾部删除 O(1) 头部插入、头部删除 O(n) | 无序 | 可重复 | 支持快速随机访问 |
| [list](https://github.com/huihut/interview/tree/master/STL" \l "list) | 双向链表 | 插入、删除 O(1) 随机读改 O(n) | 无序 | 可重复 | 支持快速增删 |
| [deque](https://github.com/huihut/interview/tree/master/STL" \l "deque) | 双端队列 | 头尾插入、头尾删除 O(1) | 无序 | 可重复 | 一个中央控制器 + 多个缓冲区，支持首尾快速增删，支持随机访问 |
| [stack](https://github.com/huihut/interview/tree/master/STL" \l "stack) | deque / list | 顶部插入、顶部删除 O(1) | 无序 | 可重复 | deque 或 list 封闭头端开口，不用 vector 的原因应该是容量大小有限制，扩容耗时 |
| [queue](https://github.com/huihut/interview/tree/master/STL" \l "queue) | deque / list | 尾部插入、头部删除 O(1) | 无序 | 可重复 | deque 或 list 封闭头端开口，不用 vector 的原因应该是容量大小有限制，扩容耗时 |
| [priority\_queue](https://github.com/huihut/interview/tree/master/STL" \l "priority_queue) | vector + max-heap | 插入、删除 O(log2n) | 有序 | 可重复 | vector容器+heap处理规则 |
| [set](https://github.com/huihut/interview/tree/master/STL" \l "set) | 红黑树 | 插入、删除、查找 O(log2n) | 有序 | 不可重复 |  |
| [multiset](https://github.com/huihut/interview/tree/master/STL" \l "multiset) | 红黑树 | 插入、删除、查找 O(log2n) | 有序 | 可重复 |  |
| [map](https://github.com/huihut/interview/tree/master/STL" \l "map) | 红黑树 | 插入、删除、查找 O(log2n) | 有序 | 不可重复 |  |
| [multimap](https://github.com/huihut/interview/tree/master/STL" \l "multimap) | 红黑树 | 插入、删除、查找 O(log2n) | 有序 | 可重复 |  |
| hash\_set | 哈希表 | 插入、删除、查找 O(1) 最差 O(n) | 无序 | 不可重复 |  |
| hash\_multiset | 哈希表 | 插入、删除、查找 O(1) 最差 O(n) | 无序 | 可重复 |  |
| hash\_map | 哈希表 | 插入、删除、查找 O(1) 最差 O(n) | 无序 | 不可重复 |  |
| hash\_multimap | 哈希表 | 插入、删除、查找 O(1) 最差 O(n) | 无序 | 可重复 |  |

#### Stl 算法

// 简单查找算法，要求输入迭代器（input iterator）

find(beg, end, val); // 返回一个迭代器，指向输入序列中第一个等于 val 的元素，未找到返回 end

find\_if(beg, end, unaryPred); // 返回一个迭代器，指向第一个满足 unaryPred 的元素，未找到返回 end

find\_if\_not(beg, end, unaryPred); // 返回一个迭代器，指向第一个令 unaryPred 为 false 的元素，未找到返回 end

count(beg, end, val); // 返回一个计数器，指出 val 出现了多少次

count\_if(beg, end, unaryPred); // 统计有多少个元素满足 unaryPred

all\_of(beg, end, unaryPred); // 返回一个 bool 值，判断是否所有元素都满足 unaryPred

any\_of(beg, end, unaryPred); // 返回一个 bool 值，判断是否任意（存在）一个元素满足 unaryPred

none\_of(beg, end, unaryPred); // 返回一个 bool 值，判断是否所有元素都不满足 unaryPred

// 查找重复值的算法，传入向前迭代器（forward iterator）

adjacent\_find(beg, end); // 返回指向第一对相邻重复元素的迭代器，无相邻元素则返回 end

adjacent\_find(beg, end, binaryPred); // 返回指向第一对相邻重复元素的迭代器，无相邻元素则返回 end

search\_n(beg, end, count, val); // 返回一个迭代器，从此位置开始有 count 个相等元素，不存在则返回 end

search\_n(beg, end, count, val, binaryPred); // 返回一个迭代器，从此位置开始有 count 个相等元素，不存在则返回 end

// 查找子序列算法，除 find\_first\_of（前两个输入迭代器，后两个前向迭代器） 外，都要求两个前向迭代器

search(beg1, end1, beg2, end2); // 返回第二个输入范围（子序列）在爹一个输入范围中第一次出现的位置，未找到则返回 end1

search(beg1, end1, beg2, end2, binaryPred); // 返回第二个输入范围（子序列）在爹一个输入范围中第一次出现的位置，未找到则返回 end1

find\_first\_of(beg1, end1, beg2, end2); // 返回一个迭代器，指向第二个输入范围中任意元素在第一个范围中首次出现的位置，未找到则返回end1

find\_first\_of(beg1, end1, beg2, end2, binaryPred); // 返回一个迭代器，指向第二个输入范围中任意元素在第一个范围中首次出现的位置，未找到则返回end1

find\_end(beg1, end1, beg2, end2); // 类似 search，但返回的最后一次出现的位置。如果第二个输入范围为空，或者在第一个输入范围为空，或者在第一个输入范围中未找到它，则返回 end1

find\_end(beg1, end1, beg2, end2, binaryPred); // 类似 search，但返回的最后一次出现的位置。如果第二个输入范围为空，或者在第一个输入范围为空，或者在第一个输入范围中未找到它，则返回 end1

// 其他只读算法，传入输入迭代器

for\_each(beg, end, unaryOp); // 对输入序列中的每个元素应用可调用对象 unaryOp，unaryOp 的返回值被忽略

mismatch(beg1, end1, beg2); // 比较两个序列中的元素。返回一个迭代器的 pair，表示两个序列中第一个不匹配的元素

mismatch(beg1, end1, beg2, binaryPred); // 比较两个序列中的元素。返回一个迭代器的 pair，表示两个序列中第一个不匹配的元素

equal(beg1, end1, beg2); // 比较每个元素，确定两个序列是否相等。

equal(beg1, end1, beg2, binaryPred); // 比较每个元素，确定两个序列是否相等。

// 二分搜索算法，传入前向迭代器或随机访问迭代器（random-access iterator），要求序列中的元素已经是有序的。通过小于运算符（<）或 comp 比较操作实现比较。

lower\_bound(beg, end, val); // 返回一个非递减序列 [beg, end) 中的第一个大于等于值 val 的位置的迭代器，不存在则返回 end

lower\_bound(beg, end, val, comp); // 返回一个非递减序列 [beg, end) 中的第一个大于等于值 val 的位置的迭代器，不存在则返回 end

upper\_bound(beg, end, val); // 返回一个非递减序列 [beg, end) 中第一个大于 val 的位置的迭代器，不存在则返回 end

upper\_bound(beg, end, val, comp); // 返回一个非递减序列 [beg, end) 中第一个大于 val 的位置的迭代器，不存在则返回 end

equal\_range(beg, end, val); // 返回一个 pair，其 first 成员是 lower\_bound 返回的迭代器，其 second 成员是 upper\_bound 返回的迭代器

binary\_search(beg, end, val); // 返回一个 bool 值，指出序列中是否包含等于 val 的元素。对于两个值 x 和 y，当 x 不小于 y 且 y 也不小于 x 时，认为它们相等。

// 只写不读算法，要求输出迭代器（output iterator）

fill(beg, end, val); // 将 val 赋予每个元素，返回 void

fill\_n(beg, cnt, val); // 将 val 赋予 cnt 个元素，返回指向写入到输出序列最有一个元素之后位置的迭代器

genetate(beg, end, Gen); // 每次调用 Gen() 生成不同的值赋予每个序列，返回 void

genetate\_n(beg, cnt, Gen); // 每次调用 Gen() 生成不同的值赋予 cnt 个序列，返回指向写入到输出序列最有一个元素之后位置的迭代器

// 使用输入迭代器的写算法，读取一个输入序列，将值写入到一个输出序列（dest）中

copy(beg, end, dest); // 从输入范围将元素拷贝所有元素到 dest 指定定的目的序列

copy\_if(beg, end, dest, unaryPred); // 从输入范围将元素拷贝满足 unaryPred 的元素到 dest 指定定的目的序列

copy\_n(beg, n, dest); // 从输入范围将元素拷贝前 n 个元素到 dest 指定定的目的序列

move(beg, end, dest); // 对输入序列中的每个元素调用 std::move，将其移动到迭代器 dest 开始始的序列中

transform(beg, end, dest, unaryOp); // 调用给定操作（一元操作），并将结果写到dest中

transform(beg, end, beg2, dest, binaryOp); // 调用给定操作（二元操作），并将结果写到dest中

replace\_copy(beg, end, dest, old\_val, new\_val); // 将每个元素拷贝到 dest，将等于 old\_val 的的元素替换为 new\_val

replace\_copy\_if(beg, end, dest, unaryPred, new\_val); // 将每个元素拷贝到 dest，将满足 unaryPred 的的元素替换为 new\_val

merge(beg1, end1, beg2, end2, dest); // 两个输入序列必须都是有序的，用 < 运算符将合并后的序列写入到 dest 中

merge(beg1, end1, beg2, end2, dest, comp); // 两个输入序列必须都是有序的，使用给定的比较操作（comp）将合并后的序列写入到 dest 中

// 使用前向迭代器的写算法，要求前向迭代器

iter\_swap(iter1, iter2); // 交换 iter1 和 iter2 所表示的元素，返回 void

swap\_ranges(beg1, end1, beg2); // 将输入范围中所有元素与 beg2 开始的第二个序列中所有元素进行交换。返回递增后的的 beg2，指向最后一个交换元素之后的位置。

replace(beg, end, old\_val, new\_val); // 用 new\_val 替换等于 old\_val 的每个匹配元素

replace\_if(beg, end, unaryPred, new\_val); // 用 new\_val 替换满足 unaryPred 的每个匹配元素

// 使用双向迭代器的写算法，要求双向选代器（bidirectional iterator）

copy\_backward(beg, end, dest); // 从输入范围中拷贝元素到指定目的位置。如果范围为空,则返回值为 dest；否则，返回值表示从 \*beg 中拷贝或移动的元素。

move\_backward(beg, end, dest); // 从输入范围中移动元素到指定目的位置。如果范围为空,则返回值为 dest；否则,返回值表示从 \*beg 中拷贝或移动的元素。

inplace\_merge(beg, mid, end); // 将同一个序列中的两个有序子序列合并为单一的有序序列。beg 到 mid 间的子序列和 mid 到 end 间的子序列被合并，并被写入到原序列中。使用 < 比较元素。

inplace\_merge(beg, mid, end, comp); // 将同一个序列中的两个有序子序列合并为单一的有序序列。beg 到 mid 间的子序列和 mid 到 end 间的子序列被合并，并被写入到原序列中。使用给定的 comp 操作。

// 划分算法，要求双向选代器（bidirectional iterator）

is\_partitioned(beg, end, unaryPred); // 如果所有满足谓词 unaryPred 的元素都在不满足 unarypred 的元素之前，则返回 true。若序列为空，也返回 true

partition\_copy(beg, end, dest1, dest2, unaryPred); // 将满足 unaryPred 的元素拷贝到到 dest1，并将不满足 unaryPred 的元素拷贝到到 dest2。返回一个迭代器 pair，其 first 成员表示拷贝到 dest1 的的元素的末尾，second 表示拷贝到 dest2 的元素的末尾。

partitioned\_point(beg, end, unaryPred); // 输入序列必须是已经用 unaryPred 划分过的。返回满足 unaryPred 的范围的尾后迭代器。如果返回的迭代器不是 end，则它指向的元素及其后的元素必须都不满足 unaryPred

stable\_partition(beg, end, unaryPred); // 使用 unaryPred 划分输入序列。满足 unaryPred 的元素放置在序列开始，不满足的元素放在序列尾部。返回一个迭代器，指向最后一个满足 unaryPred 的元素之后的位置如果所有元素都不满足 unaryPred，则返回 beg

partition(beg, end, unaryPred); // 使用 unaryPred 划分输入序列。满足 unaryPred 的元素放置在序列开始，不满足的元素放在序列尾部。返回一个迭代器，指向最后一个满足 unaryPred 的元素之后的位置如果所有元素都不满足 unaryPred，则返回 beg

// 排序算法，要求随机访问迭代器（random-access iterator）

sort(beg, end); // 排序整个范围

stable\_sort(beg, end); // 排序整个范围（稳定排序）

sort(beg, end, comp); // 排序整个范围

stable\_sort(beg, end, comp); // 排序整个范围（稳定排序）

is\_sorted(beg, end); // 返回一个 bool 值，指出整个输入序列是否有序

is\_sorted(beg, end, comp); // 返回一个 bool 值，指出整个输入序列是否有序

is\_sorted\_until(beg, end); // 在输入序列中査找最长初始有序子序列，并返回子序列的尾后迭代器

is\_sorted\_until(beg, end, comp); // 在输入序列中査找最长初始有序子序列，并返回子序列的尾后迭代器

partial\_sort(beg, mid, end); // 排序 mid-beg 个元素。即，如果 mid-beg 等于 42，则此函数将值最小的 42 个元素有序放在序列前 42 个位置

partial\_sort(beg, mid, end, comp); // 排序 mid-beg 个元素。即，如果 mid-beg 等于 42，则此函数将值最小的 42 个元素有序放在序列前 42 个位置

partial\_sort\_copy(beg, end, destBeg, destEnd); // 排序输入范围中的元素，并将足够多的已排序元素放到 destBeg 和 destEnd 所指示的序列中

partial\_sort\_copy(beg, end, destBeg, destEnd, comp); // 排序输入范围中的元素，并将足够多的已排序元素放到 destBeg 和 destEnd 所指示的序列中

nth\_element(beg, nth, end); // nth 是一个迭代器，指向输入序列中第 n 大的元素。nth 之前的元素都小于等于它，而之后的元素都大于等于它

nth\_element(beg, nth, end, comp); // nth 是一个迭代器，指向输入序列中第 n 大的元素。nth 之前的元素都小于等于它，而之后的元素都大于等于它

// 使用前向迭代器的重排算法。普通版本在输入序列自身内部重拍元素，\_copy 版本完成重拍后写入到指定目的序列中，而不改变输入序列

remove(beg, end, val); // 通过用保留的元素覆盖要删除的元素实现删除 ==val 的元素，返回一个指向最后一个删除元素的尾后位置的迭代器

remove\_if(beg, end, unaryPred); // 通过用保留的元素覆盖要删除的元素实现删除满足 unaryPred 的元素，返回一个指向最后一个删除元素的尾后位置的迭代器

remove\_copy(beg, end, dest, val); // 通过用保留的元素覆盖要删除的元素实现删除 ==val 的元素，返回一个指向最后一个删除元素的尾后位置的迭代器

remove\_copy\_if(beg, end, dest, unaryPred); // 通过用保留的元素覆盖要删除的元素实现删除满足 unaryPred 的元素，返回一个指向最后一个删除元素的尾后位置的迭代器

unique(beg, end); // 通过对覆盖相邻的重复元素（用 == 确定是否相同）实现重排序列。返回一个迭代器，指向不重复元素的尾后位置

unique (beg, end, binaryPred); // 通过对覆盖相邻的重复元素（用 binaryPred 确定是否相同）实现重排序列。返回一个迭代器，指向不重复元素的尾后位置

unique\_copy(beg, end, dest); // 通过对覆盖相邻的重复元素（用 == 确定是否相同）实现重排序列。返回一个迭代器，指向不重复元素的尾后位置

unique\_copy\_if(beg, end, dest, binaryPred); // 通过对覆盖相邻的重复元素（用 binaryPred 确定是否相同）实现重排序列。返回一个迭代器，指向不重复元素的尾后位置

rotate(beg, mid, end); // 围绕 mid 指向的元素进行元素转动。元素 mid 成为为首元素，随后是 mid+1 到到 end 之前的元素，再接着是 beg 到 mid 之前的元素。返回一个迭代器，指向原来在 beg 位置的元素

rotate\_copy(beg, mid, end, dest); // 围绕 mid 指向的元素进行元素转动。元素 mid 成为为首元素，随后是 mid+1 到到 end 之前的元素，再接着是 beg 到 mid 之前的元素。返回一个迭代器，指向原来在 beg 位置的元素

// 使用双向迭代器的重排算法

reverse(beg, end); // 翻转序列中的元素，返回 void

reverse\_copy(beg, end, dest);; // 翻转序列中的元素，返回一个迭代器，指向拷贝到目的序列的元素的尾后位置

// 使用随机访问迭代器的重排算法

random\_shuffle(beg, end); // 混洗输入序列中的元素，返回 void

random\_shuffle(beg, end, rand); // 混洗输入序列中的元素，rand 接受一个正整数的随机对象，返回 void

shuffle(beg, end, Uniform\_rand); // 混洗输入序列中的元素，Uniform\_rand 必须满足均匀分布随机数生成器的要求，返回 void

// 最小值和最大值，使用 < 运算符或给定的比较操作 comp 进行比较

min(val1, va12); // 返回 val1 和 val2 中的最小值，两个实参的类型必须完全一致。参数和返回类型都是 const的引引用，意味着对象不会被拷贝。下略

min(val1, val2, comp);

min(init\_list);

min(init\_list, comp);

max(val1, val2);

max(val1, val2, comp);

max(init\_list);

max(init\_list, comp);

minmax(val1, val2); // 返回一个 pair，其 first 成员为提供的值中的较小者，second 成员为较大者。下略

minmax(vall, val2, comp);

minmax(init\_list);

minmax(init\_list, comp);

min\_element(beg, end); // 返回指向输入序列中最小元素的迭代器

min\_element(beg, end, comp); // 返回指向输入序列中最小元素的迭代器

max\_element(beg, end); // 返回指向输入序列中最大元素的迭代器

max\_element(beg, end, comp); // 返回指向输入序列中最大元素的迭代器

minmax\_element(beg, end); // 返回一个 pair，其中 first 成员为最小元素，second 成员为最大元素

minmax\_element(beg, end, comp); // 返回一个 pair，其中 first 成员为最小元素，second 成员为最大元素

// 字典序比较，根据第一对不相等的元素的相对大小来返回结果。如果第一个序列在字典序中小于第二个序列，则返回 true。否则，返回 fa1se。如果个序列比另一个短，且所有元素都与较长序列的对应元素相等，则较短序列在字典序中更小。如果序列长度相等，且对应元素都相等，则在字典序中任何一个都不大于另外一个。

lexicographical\_compare(beg1, end1, beg2, end2);

lexicographical\_compare(beg1, end1, beg2, end2, comp);

# 二．算法

## 2.1排序

| 排序算法 | 平均时间复杂度 | 最差时间复杂度 | 空间复杂度 | 数据对象稳定性 |
| --- | --- | --- | --- | --- |
| [冒泡排序](https://github.com/huihut/interview/blob/master/Algorithm/BubbleSort.h) | O(n2) | O(n2) | O(1) | 稳定 |
| [选择排序](https://github.com/huihut/interview/blob/master/Algorithm/SelectionSort.h) | O(n2) | O(n2) | O(1) | 数组不稳定、链表稳定 |
| [插入排序](https://github.com/huihut/interview/blob/master/Algorithm/InsertSort.h) | O(n2) | O(n2) | O(1) | 稳定 |
| [快速排序](https://github.com/huihut/interview/blob/master/Algorithm/QuickSort.h) | O(n\*log2n) | O(n2) | O(log2n) | 不稳定 |
| [堆排序](https://github.com/huihut/interview/blob/master/Algorithm/HeapSort.cpp) | O(n\*log2n) | O(n\*log2n) | O(1) | 不稳定 |
| [归并排序](https://github.com/huihut/interview/blob/master/Algorithm/MergeSort.h) | O(n\*log2n) | O(n\*log2n) | O(n) | 稳定 |
| [希尔排序](https://github.com/huihut/interview/blob/master/Algorithm/ShellSort.h) | O(n\*log2n) | O(n2) | O(1) | 不稳定 |
| [计数排序](https://github.com/huihut/interview/blob/master/Algorithm/CountSort.cpp) | O(n+m) | O(n+m) | O(n+m) | 稳定 |
| [桶排序](https://github.com/huihut/interview/blob/master/Algorithm/BucketSort.cpp) | O(n) | O(n) | O(m) | 稳定 |
| [基数排序](https://github.com/huihut/interview/blob/master/Algorithm/RadixSort.h) | O(k\*n) | O(n2) |  | 稳定 |

* 均按从小到大排列
* k：代表数值中的 “数位” 个数
* n：代表数据规模
* m：代表数据的最大值减最小值
* 来自：[wikipedia . 排序算法](https://zh.wikipedia.org/wiki/%E6%8E%92%E5%BA%8F%E7%AE%97%E6%B3%95)

## 2.2查找

| 查找算法 | 平均时间复杂度 | 空间复杂度 | 查找条件 |
| --- | --- | --- | --- |
| [顺序查找](https://github.com/huihut/interview/blob/master/Algorithm/SequentialSearch.h) | O(n) | O(1) | 无序或有序 |
| [二分查找（折半查找）](https://github.com/huihut/interview/blob/master/Algorithm/BinarySearch.h) | O(log2n) | O(1) | 有序 |
| [插值查找](https://github.com/huihut/interview/blob/master/Algorithm/InsertionSearch.h) | O(log2(log2n)) | O(1) | 有序 |
| [斐波那契查找](https://github.com/huihut/interview/blob/master/Algorithm/FibonacciSearch.cpp) | O(log2n) | O(1) | 有序 |
| [哈希查找](https://github.com/huihut/interview/blob/master/DataStructure/HashTable.cpp) | O(1) | O(n) | 无序或有序 |
| [二叉查找树（二叉搜索树查找）](https://github.com/huihut/interview/blob/master/Algorithm/BSTSearch.h) | O(log2n) |  |  |
| [红黑树](https://github.com/huihut/interview/blob/master/DataStructure/RedBlackTree.cpp) | O(log2n) |  |  |
| 2-3树 | O(log2n - log3n) |  |  |
| B树/B+树 | O(log2n) |  |  |

## **2.3图搜索算法**

| 图搜索算法 | 数据结构 | 遍历时间复杂度 | 空间复杂度 |
| --- | --- | --- | --- |
| [BFS广度优先搜索](https://zh.wikipedia.org/wiki/%E5%B9%BF%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2) | 邻接矩阵 邻接链表 | O(|v|2) O(|v|+|E|) | O(|v|2) O(|v|+|E|) |
| [DFS深度优先搜索](https://zh.wikipedia.org/wiki/%E6%B7%B1%E5%BA%A6%E4%BC%98%E5%85%88%E6%90%9C%E7%B4%A2) | 邻接矩阵 邻接链表 | O(|v|2) O(|v|+|E|) | O(|v|2) O(|v|+|E|) |

## 2.4 其他算法

| **算法** | **思想** | **应用** |
| --- | --- | --- |
| [分治法](https://zh.wikipedia.org/wiki/%E5%88%86%E6%B2%BB%E6%B3%95) | 把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并 | [循环赛日程安排问题](https://github.com/huihut/interview/tree/master/Problems/RoundRobinProblem)、排序算法（快速排序、归并排序） |
| [动态规划](https://zh.wikipedia.org/wiki/%E5%8A%A8%E6%80%81%E8%A7%84%E5%88%92) | 通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法，适用于有重叠子问题和最优子结构性质的问题 | [背包问题](https://github.com/huihut/interview/tree/master/Problems/KnapsackProblem)、斐波那契数列 |
| [贪心法](https://zh.wikipedia.org/wiki/%E8%B4%AA%E5%BF%83%E6%B3%95) | 一种在每一步选择中都采取在当前状态下最好或最优（即最有利）的选择，从而希望导致结果是最好或最优的算法 | 旅行推销员问题（最短路径问题）、最小生成树、哈夫曼编码 |

# 三．操作系统

## 3.1 进程与线程

对于有线程系统：

* 进程是资源分配的独立单位
* 线程是资源调度的独立单位

对于无线程系统：

* 进程是资源调度、分配的独立单位

### 3.1.1进程之间的通信方式以及优缺点

* 管道（PIPE）
  + 有名管道：一种半双工的通信方式，它允许无亲缘关系进程间的通信
    1. 优点：可以实现任意关系的进程间的通信
    2. 缺点：
       1. 长期存于系统中，使用不当容易出错
       2. 缓冲区有限
  + 无名管道：一种半双工的通信方式，只能在具有亲缘关系的进程间使用（父子进程）
    1. 优点：简单方便
    2. 缺点：
       1. 局限于单向通信
       2. 只能创建在它的进程以及其有亲缘关系的进程之间
       3. 缓冲区有限
* 信号量（Semaphore）：一个计数器，可以用来控制多个线程对共享资源的访问
  + 优点：可以同步进程
  + 缺点：信号量有限
* 信号（Signal）：一种比较复杂的通信方式，用于通知接收进程某个事件已经发生
* 消息队列（Message Queue）：是消息的链表，存放在内核中并由消息队列标识符标识
  + 优点：可以实现任意进程间的通信，并通过系统调用函数来实现消息发送和接收之间的同步，无需考虑同步问题，方便
  + 缺点：信息的复制需要额外消耗 CPU 的时间，不适宜于信息量大或操作频繁的场合
* 共享内存（Shared Memory）：映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问
  + 优点：无须复制，快捷，信息量大
  + 缺点：
    1. 通信是通过将共享空间缓冲区直接附加到进程的虚拟地址空间中来实现的，因此进程间的读写操作的同步问题
    2. 利用内存缓冲区直接交换信息，内存的实体存在于计算机中，只能同一个计算机系统中的诸多进程共享，不方便网络通信
* 套接字（Socket）：可用于不同及其间的进程通信
  + 优点：
    1. 传输数据为字节级，传输数据可自定义，数据量小效率高
    2. 传输数据时间短，性能高
    3. 适合于客户端和服务器端之间信息实时交互
    4. 可以加密,数据安全性强
  + 缺点：需对传输的数据进行解析，转化成应用级的数据。

### 3.1.2 线程之间的通信方式

* 锁机制：包括互斥锁/量（mutex）、读写锁（reader-writer lock）、自旋锁（spin lock）、条件变量（condition）
  + 互斥锁/量（mutex）：提供了以排他方式防止数据结构被并发修改的方法。
  + 读写锁（reader-writer lock）：允许多个线程同时读共享数据，而对写操作是互斥的。
  + 自旋锁（spin lock）与互斥锁类似，都是为了保护共享资源。互斥锁是当资源被占用，申请者进入睡眠状态；而自旋锁则循环检测保持者是否已经释放锁。
  + 条件变量（condition）：可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。
* 信号量机制(Semaphore)
  + 无名线程信号量
  + 命名线程信号量
* 信号机制(Signal)：类似进程间的信号处理
* 屏障（barrier）：屏障允许每个线程等待，直到所有的合作线程都达到某一点，然后从该点继续执行。

线程间的通信目的主要是用于线程同步，所以线程没有像进程通信中的用于数据交换的通信机制

### 3.1.3 进程之间私有和共享的资源

* 私有：地址空间、堆、全局变量、栈、寄存器
* 共享：代码段，公共数据，进程目录，进程 ID

### 3.1.4 线程之间私有和共享的资源

* 私有：线程栈，寄存器，程序寄存器
* 共享：堆，地址空间，全局变量，静态变量

### 3.1.5多进程与多线程间的对比、优劣与选择

##### **对比**

| 对比维度 | 多进程 | 多线程 | 总结 |
| --- | --- | --- | --- |
| 数据共享、同步 | 数据共享复杂，需要用 IPC；数据是分开的，同步简单 | 因为共享进程数据，数据共享简单，但也是因为这个原因导致同步复杂 | 各有优势 |
| 内存、CPU | 占用内存多，切换复杂，CPU 利用率低 | 占用内存少，切换简单，CPU 利用率高 | 线程占优 |
| 创建销毁、切换 | 创建销毁、切换复杂，速度慢 | 创建销毁、切换简单，速度很快 | 线程占优 |
| 编程、调试 | 编程简单，调试简单 | 编程复杂，调试复杂 | 进程占优 |
| 可靠性 | 进程间不会互相影响 | 一个线程挂掉将导致整个进程挂掉 | 进程占优 |
| 分布式 | 适应于多核、多机分布式；如果一台机器不够，扩展到多台机器比较简单 | 适应于多核分布式 | 进程占优 |

##### **优劣**

| 优劣 | 多进程 | 多线程 |
| --- | --- | --- |
| 优点 | 编程、调试简单，可靠性较高 | 创建、销毁、切换速度快，内存、资源占用小 |
| 缺点 | 创建、销毁、切换速度慢，内存、资源占用大 | 编程、调试复杂，可靠性较差 |

##### **选择**

* 需要频繁创建销毁的优先用线程
* 需要进行大量计算的优先使用线程
* 强相关的处理用线程，弱相关的处理用进程
* 可能要扩展到多机分布的用进程，多核分布的用线程
* 都满足需求的情况下，用你最熟悉、最拿手的方式

## 3.2 Linux 内核的同步方式

#### **原因**

在现代操作系统里，同一时间可能有多个内核执行流在执行，因此内核其实象多进程多线程编程一样也需要一些同步机制来同步各执行单元对共享数据的访问。尤其是在多处理器系统上，更需要一些同步机制来同步不同处理器上的执行单元对共享的数据的访问。