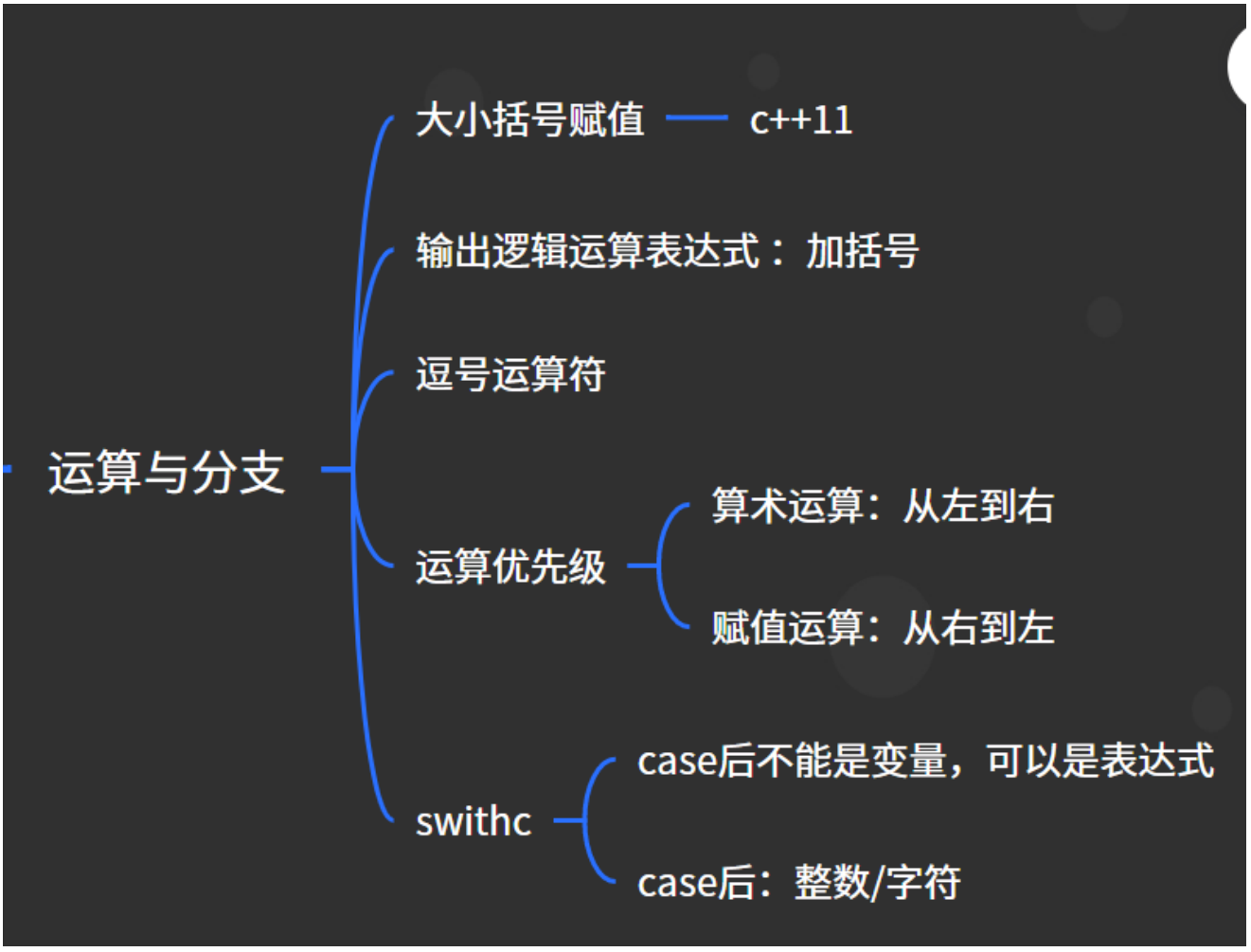
C++

#CS语言

c++项目多windows开发，linux下部署

基础内容

运算与分支



C++ 11 初始化赋值

以下式等价:

```
int a = 10;
int b = (10);
int c(10);
int e = {10};
int f{10};
```

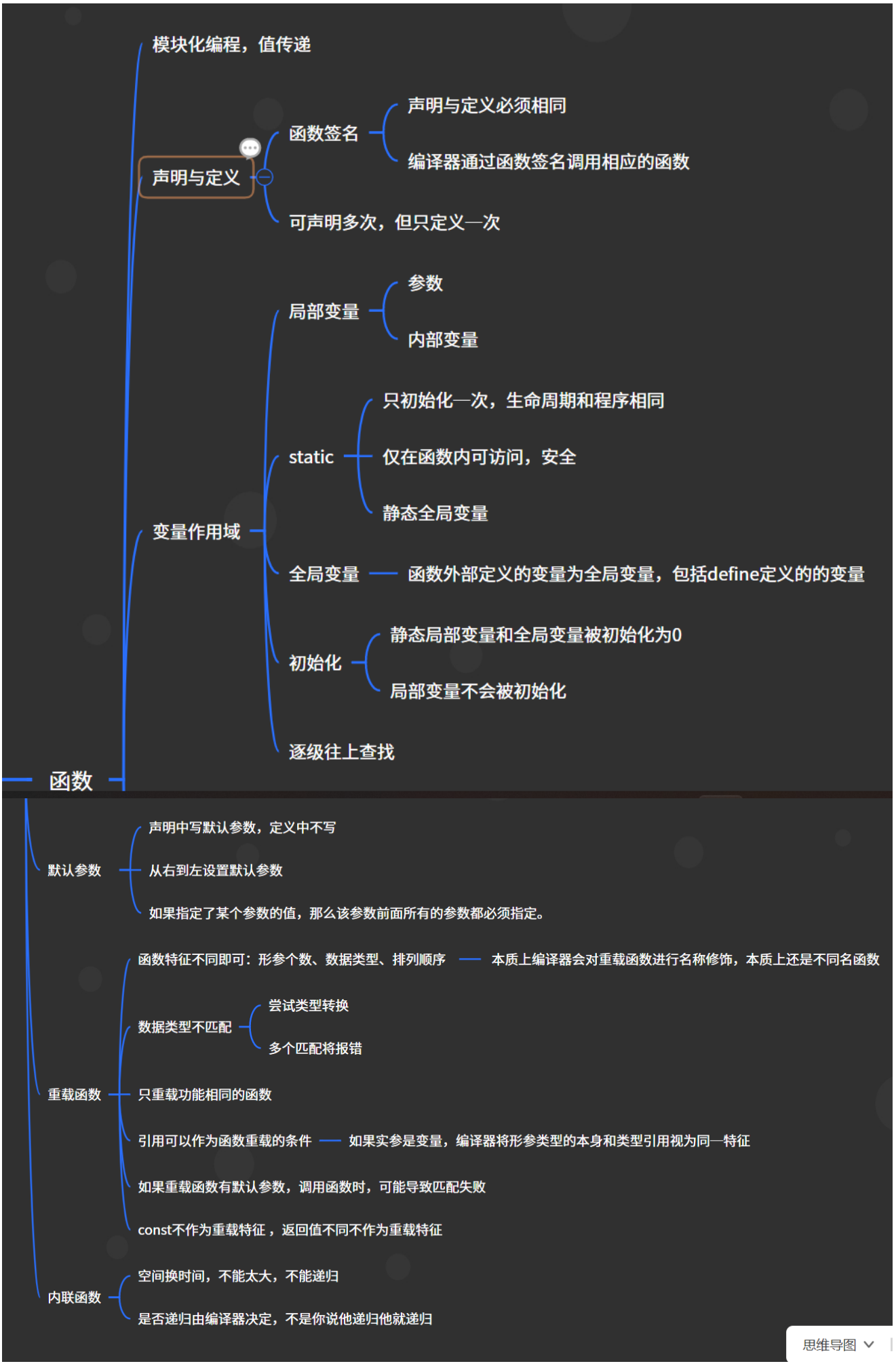
注意：在 Linux 平台下，编译需要加-std=c++11 参数。

输出逻辑运算表达式

——加括号

```
cout << "a&&b=" << (a && b) << endl;
```

函数


函数
模块化编程，值传递
声明与定义
函数签名
声明与定义必须相同
编译器通过函数签名调用相应的函数
可声明多次，但只定义一次
变量作用域
局部变量
参数
内部变量
static
只初始化一次，生命周期和程序相同
仅在函数内可访问，安全
静态全局变量
全局变量
函数外部定义的变量为全局变量，包括define定义的的变量
初始化
静态局部变量和全局变量被初始化为0
局部变量不会被初始化
逐级往上查找
默认参数
声明中写默认参数，定义中不写
从右到左设置默认参数
如果指定了某个参数的值，那么该参数前面所有的参数都必须指定。
重载函数
函数特征不同即可：形参个数、数据类型、排列顺序
本质上编译器会对重载函数进行名称修饰，本质上还是不同名函数
数据类型不匹配
尝试类型转换
多个匹配将报错
只重载功能相同的函数
引用可以作为函数重载的条件
如果实参是变量，编译器将形参类型的本身和类型引用视为同一特征
如果重载函数有默认参数，调用函数时，可能导致匹配失败
const不作为重载特征，返回值不同不作为重载特征
内联函数
空间换时间，不能太大，不能递归
是否递归由编译器决定，不是你说他递归他就递归
思维导图

默认参数

```
void func(int bh,const string &name="x", const string& message="sth.")
{
cout << "亲爱的"<<name<<"（"<<bh<<"）：" << message << endl;
}
```

重载函数

引用可以作为函数重载的条件，但是，调用重载函数的时候，如果实参是变量，编译器将形参类型的本身和类型引用视为同一特征

```
void myswap(int& a, int& b)
{
int tmp = a; a = b; b = tmp;
}
void myswap(int a, int& b)
{
int tmp = a; a = b; b = tmp;
}
myswap(10, cnt); //调用不明确
myswap(a, b); //可以编译
```

如果重载函数有默认参数，调用函数时，可能导致匹配失败。

```
void myswap(int& a, int& b)
{
int tmp = a; a = b; b = tmp;
}
void myswap(int a, int& b, int gift = 10)
{
int tmp = a; a = b; b = tmp;
}
myswap(a, b);//调用不明确
```

const不作为重载特征

```
void print(const int x) {
std::cout << "Const Integer: " << x << std::endl;
}

void print(int x) {
std::cout << "Integer: " << x << std::endl;
}//被视为同一个函数，定义了两次
```

C++数据类型

C++ 11 long long

不同系统的变量大小不一，跨平台需要做特殊的处理

long long大小 >= long

- vs中long为4字节，long long为8字节
- linux中long为8字节，long long也为8字节

浮点数

在 VS 和 Linux 中，long double 占用的内存空间分别是 8 和 16 字节。long double >= double。

在实际开发中，用整数代替浮点数，整数的运算更快，精度更高。

原始字面量

——无需转移和连接

```
string path1 = R"abcd(C:\Program Files\Microsoft OneDrive\tail\nation)abcd";
```

字符串

1. C 风格字符串的本质是字符数组
2. C++风格字符串的本质是类，它封装了 C 风格字符串，可以进行拼接和=赋值，支持关系运算符

bool

占1字节

数据类型转换

不同类型的数据进行混合运算经常出现，某些类型的转换编译器可以隐式的进行，不需程序员干预，有些类型的转换需要程序员显式指定。

- char → short → int → long → long long → 浮点
- float → double → long double

自动由低→高转换，右值自动转换为左值--存在截断现象

typedef

——提高可读性/提高兼容性

```
typedef short int16_t;
using short = int16_t;
```

结构体、共用体和枚举

结构体

1. 一般定义在函数之前 或 头文件中
2. 成员可以是类、函数，但不提倡(其实结构体就是类)
3. C++ 11结构体可以指定默认值
4. 结构体大小——按字长对齐，对结构体使用sizeof可能没有意义
5. 结构体变量名并没有被解释为地址，作为参数的结构体会被整个复制过去

初始化

```
struct people ppl1 = {"a", 18};
```

结构体指针

```
(*arr).member;//()不能省略
pst->name;
*pst->age;
```

结构体中的指针

赋初始值不多，清空变量经常用——memset()

而对结构体用 memset()函数可能会造成内存泄露

1. 清理了结构体但未清理结构体指针指向的内存
2. 应该清空指针指向的内容，而非清空指针
3. 如果成员为string等C++类，memset是不可靠的——memset只适用于基本数据类型

```
struct st_boy
{
string name; //超女姓名。
};
int main()
{
st_boy boy; //创建结构体变量。
boy.name = "西";
cout << "boy.name=" << boy.name << endl;
memset(&boy, 0, sizeof(boy));
boy.name = "好大的西瓜";
cout << "boy.name=" << boy.name << endl;
}
```

共用体

```
union
{
int a;
double b;
char c[25];
} data;
```

1. 它能存储不同的数据类型，但是，在同一时间只能存储其中的一种类型。
2. 全部的成员使用同一块内存，共同体中的值为最后被赋值的那个成员的值
3. 占用内存的大小是它最大的成员占用内存的大小（内存对齐）。

经常以匿名的形式出现

```
struct st_boy
{
int no;
union
{
int a;
double b;
char c[21];
};
}boy;
...
cout << "boy.a 的地址是：" << (void*) &boy.a << endl;//注意
```

枚举

常用创建符号常量，枚举类型值域只能取定义的符号常量，作用域和普通变量相同，即便有默认值也不能通过默认值直接赋值

```
enum colors { red , yellow , blue };//默认012...
enum colors
{
red = 1,
yellow = 2,
blue = 3
};
enum colors
{
red, //0
yellow = 10,
blue //11
};

colors cc = yellow;
colors cc = 1;//错误
colors cc = colors(1); //可以

switch (cc)
{
case red: cout << "红色。\n"; break;
case yellow: cout << "黄色。\n"; break;
case blue: cout << "蓝色。\n"; break;
default: cout << "未知。\n";
}
```

指针、内存和数组

——占一个字长

跟踪：数据存储物理地址、数据类型、数据值

减少拷贝，提升性能；函数中修改实参

形形色色的指针

const与指针

```
char *const p1;
const char *p2;
char const *p3;
const char *const p4;
```

```
//指针指向不变，指向对象的内容可变，常用于硬件
//指向对象的值不可变，指针指向可变
//指向对象的值不可变，指针指向可变
//指向不变，指向对象不变，常用于ROM只读存储器
```

void与指针

void*也称万能指针，不能解引用，必须转换为其他类型的指针使用

其他类型指针可以直接赋值给它，无需转换

空指针

- 0和NULL均可表示空指针
- 解引用空指针程序会崩溃——0x00000000 到 0x0000FFFF均会崩溃，无物理存储器与之对应
- delete空指针会被忽略
- 动态分配内存时，应判断形参是否为空指针

C++11 nullptr

——为了防止作为(void*)0的NULL会引起的误解：

- NULL课已被隐式转化为整型，因此在重载函数时，编译器无法确定此处的参数为整型还是指针类型
- nullptr不会被隐式转换为整数类型
- Linux下使用 nullptr，编译需要加-std=c++11 参数。

野指针

指针指向非合法地址，常见的可能导致野指针出现的行为有：

- 未初始化
- 动态分配的内存被释放后，其指针未被置空
- 指针指向超出作用域内存：如指向函数局部变量的指针，函数被回收后该内存仍被指针指向

函数指针与回调函数

- 由于函数名就是函数指针，对其进行取地址、解引用亦或不作都会被编译器当成 函数名处理

```
void say()
{
cout << "不该如此" << endl;
}
int main()
{
void (*func_ptr)(); // 函数指针
func_ptr = say; // 函数名就是函数指针，&say同样可以；C++，C
func_ptr(); // C++
(*func_ptr)(); // C++，C
}
```

回调函数

- 调用者函数提供主体的流程和框架，将回调函数嵌入到调用者函数中实现<特定>的功能

传递参数：

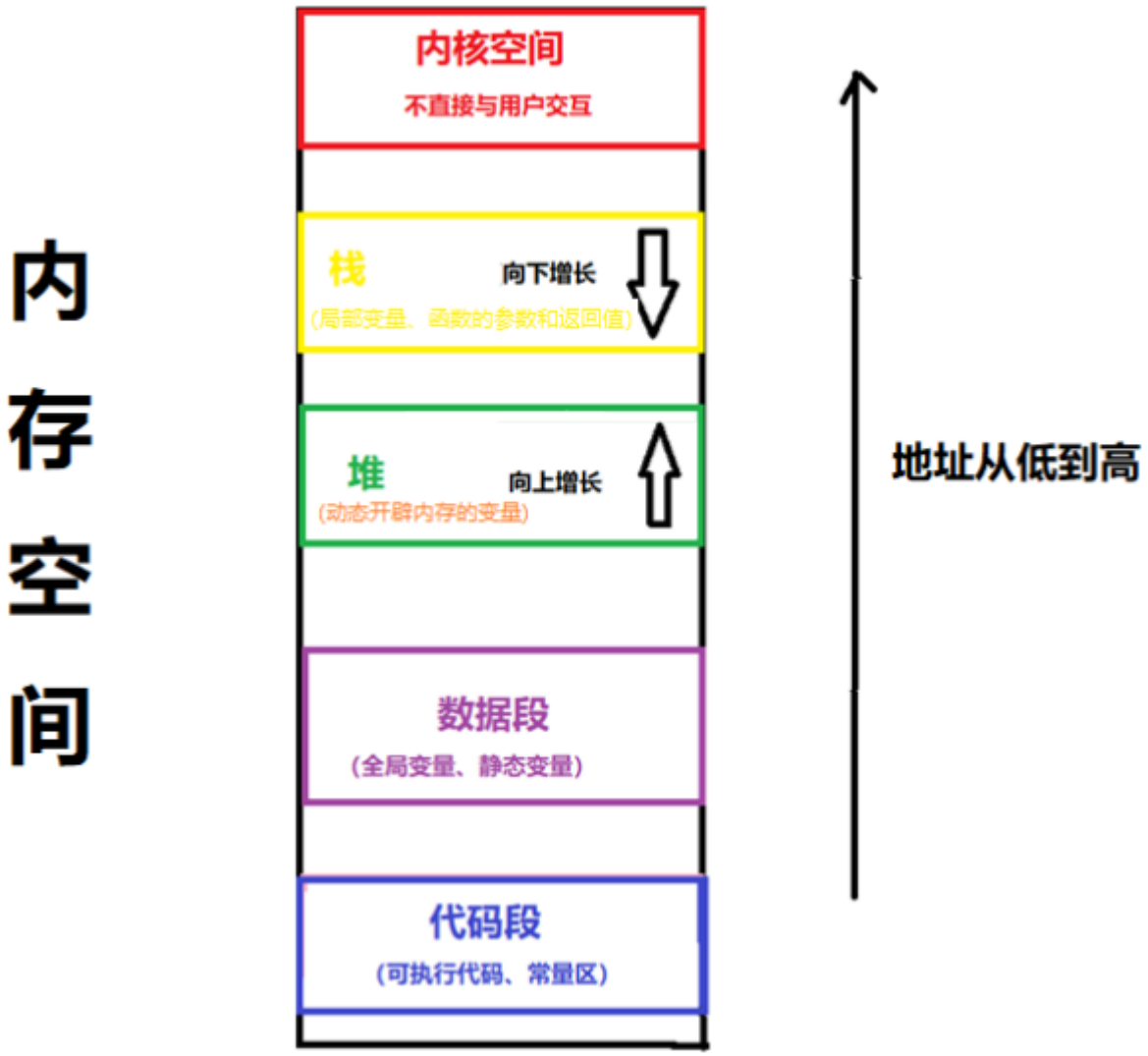
```
void show_something(void (*sth)(int))
{
cout << "我想说的是" << endl;
(*sth)(3);
cout << "仅此而已" << endl;
}
```

或

```
void show_something(void (*sth)(int), int num)
{
cout << "我想说的是" << endl;
(*sth)(num);
cout << "仅此而已" << endl;
}
...
show_something(say，3);
```

C++ 内存模型

1. 堆由程序员动态进行管理
   - 整个程序结束时，即便有未释放的内存，也会被OS回收
2. 栈存储：局部变量、函数参数、返回值，实现自动管理：返回时自动释放
3. 堆大小受限于物理内存空间，栈大小一般只有8M(可以修改)



动态分配内存

- 注意指针别跟丢

```
int *ptr = new int;
delete ptr;

int *arr = new int[10];
delete[] arr;
```

数组

```
sizeof(arr);//整个数组的大小

//C++11初始化数组可不写等于号

void *memset(void *s, int c, size_t n);//<string.h>
void *memcpy(void *dest, const void *src, size_t n);
```

1. C++编译器把 数组名[下标] 解释为 *(数组首地址+下标)
2. 数组名被解释为数组第 0 个元素的地址
3. 数组名不一定会被解释为地址--sizeof
4. memset可以对数组使用，指针应该不行(?)

作为参数的数组

完完全全的变成了指针--sizeof会失效，一般同时传入数组地址和数组长度

```
void func(int (*p)[3],int len);
void func(int p[][3],int len);//定义，前面的[]可以不写任何值
```

动态创建的数组

1. 动态创建的数组没有数组名，不能用 sizeof 运算符
2. 必须使用 delete[]来释放动态数组的内存（不能只用 delete）

引用

1. 可以理解为别名，但本质是指针常量的伪装，只是相对于指针更加安全
2. 主要用作形参和返回值
3. 没有为数组or指针建引用的说法，因为在传递时这没有任何意义，二者的设计目的明显不同
4. 引用建立的初衷仅仅是使代码更加简洁、可读
5. 指针则是为了各种复杂操作而建立
6. 二者已经涵盖所有需求，没有必要为数组和指针建立引用

```
int a = 1;
int &b = a;//数据类型相同
```

作函数参数

1. 更简洁
2. 不必使用高级指针
3. const + &

```
void func3(int &no, string &str) //更简洁
{
no = 8;
str = "我有一只小小鸟。";
cout << "亲爱的" << no << "号： " << str << endl;
}
//不必使用二级指针
void func1(int** p) // 传地址，实参是指针的地址，形参是二级指针。
{
*p = new int(3); // p 是二级指针，存放指针的地址。
cout << "func1 内存的地址是： " << *p << "， 内存中的值是： " << **p << endl;
}
void func2(int*& p) // 传引用，实参是指针，形参是指针的别名。
{
p = new int(3); // p 是指针的别名。
cout << "func2 内存的地址是： " << p << "， 内存中的值是： " << *p << endl;
}
```

引用和const

向函数传递常量参数是经常出现的场景， 而：

```
int z = 1;
int &a = 8; // 报错，引用的本质是指针的伪装
const int &b = 8; // 可行，类型正确，不是左值，创建临时变量
const int &c = 'x'; // 可行，类型不正确，可被转化为正确的类型，创建临时变量
const int &d = "x"; // 报错，类型不正确，不可被转化为正确的类型，不创建临时变量
const int &e = z; // 可行，类型正确，但是左值，不创建临时变量
```

即：

1. 如果引用的数据对象类型不匹配
2. 但可被转化为正确的类型 或类型正确且不是左值
3. 且引用为 const 时

C++将创建临时变量，让引用指向临时变量。指针并没有这种待遇。

注：左值是可以被引用的数据对象，可以通过地址访问它们

作函数返回值

1. 传统的函数返回机制：函数的返回值被拷贝到一个临时位置（寄存器或栈），然后调用者程序再使用这个值。
2. 返回引用不会拷贝内存,但会拷贝指针
3. 可以返回：引用形参、类成员、全局变量、静态变量

- 注意不能返回局部变量的引用——野指针

const type&

```
const int &func()
{
static int a = 1;
return a;
}
int main()
{
const int &num1 = func(); // 别名，不可修改
int &num2 = func(); // 禁止，非const类型不能成为const类型的别名
int num3 = func(); // 单纯赋值,可修改
const int num4 = func(); // 单纯赋值,不可修改

return 0;
}
```

形参使用原则

传参三大方法：传值，传引用，传地址；以下是形参使用的一些基本原则，但也很可能有充分的理由做出其他的选择

不需要在函数中修改实参

```
void func1(int *); // 实参很小
void func2(const struct people *); // 实参较大
void func3(const struct people &); // 实参较大
void func4(const int *array, int size); // 实参是数组
void func5(const Person &); // 实参是类
```

需要在函数中修改实参

```
void func1(int *); // 内置数据类型
void func2(struct pepole *); // 结构体
void func3(struct pepole &); // 结构体
void func3(pepole &); // 类

int main()
{
int x, *a;
func1(a); // 无法确定语义
func1(&x); // 函数将修改x
}
```

类与对象

——OOP

类的访问权限

1. 类内部成员不受权限限制，而外部只能访问public成员
2. 结构体的成员缺省为 public，类的成员缺省为 private

类使用的简单规则

1. oop思想，一般**不直接访问成员变量**
2. 对象一般**不用 memset()清空成员变量**，可以写一个专用于清空成员变量的成员函数。
3. C++ 中在类声明中定义的函数会自动成为**内联函数**

- 减少了函数调用的开销
- 对于较复杂的函数或者需要频繁调用但代码较长的函数，编译器可能会选择不进行内联。

6. 为了**区分**类的成员变量和成员函数的形参，把成员变量名加m_前缀或_后缀，如 m*name 或 name*。
7. 类的**分文件编写**，可以提高代码的可维护性和可读性

```
// Person.h

#ifndef PERSON_H
#define PERSON_H

#include <string>

class Person {
private:
std::string name;
int age;

public:
// Constructor
Person(const std::string& name, int age);

// Getter methods
std::string getName() const;
int getAge() const;

// Setter methods
void setName(const std::string& name);
void setAge(int age);
};

#endif // PERSON_H

// Person.cpp

#include "Person.h"

// Constructor
Person::Person(const std::string& name, int age) : name(name), age(age) {}

// Getter methods
std::string Person::getName() const {
return name;
}
...
```

```
// main.cpp

#include <iostream>
#include "Person.h"

int main() {
...
return 0;
}
```

构造函数和析构函数

1. 构造函数：在创建对象时，自动的进行初始化工作。
- 可以有参数、重载、默认参数
- 禁止手动调用，只允许创建对象时的一次自动调用
- 然而，可以使用委托构造函数，或者说，在一个构造函数中调用另一个构造函数来达到间接调用构造函数的效果
2. 析构函数：在销毁对象前，自动的完成清理工作。
- 无参数，不重载
- 可以手动调用，在销毁对象前会自动调用一次

```
class Person {
public:
// Constructor 必须是public，无返回值，不写void，名称与类名相同
Person(){...}

// Destructor 必须是public，无返回值，不写void，名称=~+类名
~Person(){...}
};
```

细节

1. **空实现**：如果没有提供构造/析构函数，编译器将提供空实现的构造/析构函数；否则不提供空~

2. **歧义**

```
ppl xm();//如果没有构造函数、构造函数没有参数、构造函数的参数都有默认参数，应去掉()
```

3. **委托构造函数 匿名对象**

```
class ppl {
private:
int *ptr_;

public:
ppl() { ptr_ = nullptr; }

ppl(int a = 1) {
ppl();//创建匿名对象，而非调用构造函数，未将ptr_置空
}

//正确做法
ppl(int a = 1) : ppl() {
cout << words;
}

~ppl() {
if (ptr_ != nullptr)
delete ptr_;//可能导致出现野指针
}
};
```

4. **对象初始化为一个值**

```
Person person1 = 30;
//接受一个参数的构造函数允许使用赋值语法将对象初始化为一个值
//但如果不清晰地使用，可能会导致不必要的隐式类型转换
```

5. **多次创建对象**

```
CGirl girl = CGirl("西施"20); // 显式创建对象--一次构造一次析构
CGirl girl; // 创建对象。
girl = CGirl("西施"20); // 创建匿名对象，然后给现有的对象赋值--两次构造两次析构
```

6. 用 **new/delete** 创建/销毁对象时，也会调用构造/析构函数。
7. **工程经验**：不建议在构造/析构函数中写太多的代码

- 可读性和可维护性
- 构造函数和析构函数中的代码应该保持完全正确运行，即使在发生异常的情况下，也应该能够正确地创建/清理资源。
- 构造函数和析构函数的性能对于对象的创建和销毁过程至关重要。过多的复杂逻辑可能会导致构造和析构过程变慢，影响整体性能。

8. **类的成员也是类**

- 先构造成员类，后构造自身类；先析构自身类，后析构成员类
- 对象A包含对象B，A肯定持有B的引用，所以必须先解决引用的问题，不然B析构了会有**无效引用**。A的析构完成，A对B的引用全部失效。再析构B，就比较稳妥。
- 构造过程正好相反，构造函数执行时，子对象已经构造完成了，引用自然是有效的

9. **统一初始化列表**

```
Person(const std::string& name, int age) : name(name), age(age) {}
```

拷贝构造函数

1. 用一个**已存在的对象创建新的**对象
2. **不会调用（普通）构造函数**，而是调用拷贝构造函数。
3. 缺省、未定义默认拷贝构造函数时，**编译器会提供**拷贝构造函数

```
class Person {
public:
// 必须是public，无返回值，不写void，名称与类名相同
Person(const Person& p){}
};

Person p2(p1);
Person p2 = p1;
```

可重载，可有默认参数

浅拷贝

- 两个指针指向同一块堆内存：

1. 一个指针释放资源，另一个指针就会变成野指针
2. 一个指针的修改会体现在另一个指针上

- 成员有指针，使用默认拷贝函数就会出现这种情况

深拷贝

- 解决浅拷贝造成的问题，重载复制构造函数，重新为指针成员分配内存

初始化列表--非C++ 11

构造函数执行分两个阶段：初始化阶段 → 计算阶段

```
class Person
{
private:
string name_;
int age_;

public:
Person(string n, int a) : name_("@" + n), age_(a)
{
// 不应再次给初始化列表成员赋值，会覆盖掉原值
cout << "name: " << name_ << " age: " << age_ << endl;
}
};
```

提升效率

值得注意的是，初始化列表并非简单赋值那么简单

```
CBoy boy("子都");
CGirl g1("冰冰",18,boy);
```

1. 传统的函数中赋值，程序的调用顺序为：boy构造--boy拷贝构造--boy构造--girl构造
2. 若girl构造参数改为CBoy& boy，则结果为：boy构造--boy构造--girl构造
3. 若使用初始化列表：boy构造--boy拷贝构造--girl构造

构造赋值会额外进行不必要的初始化操作，创建对象等；而拷贝构造只调用拷贝函数，且符合语义，节省资源——对象无默认构造函数时也需要用初始化列表

成员为常量和引用的构造

——类和引用只能在被创建时初始化

```
class Person
{
private:
string name_;
const int age_;

public:
Person(string n, int a) : name_("@" + n), age_(a)
{
cout << "name: " << name_ << " age: " << age_ << endl;
}
};
```

const函数

函数中不会修改成员变量

```
class Person
{
private:
string name_;
const int age_;

public:
//一些函数

show() const
{
cout << "name: " << name_ << " age: " << age_ << endl;
}
};
```

1. mutable可以突破const限制，在该函数中被修改——灵活性+语义
2. const函数只能调用const函数
3. const对象只能调用const函数——构造函数和析构函数不在此列，他们的作用是创建和销毁，而非修改

this指针

可读性upup

1. 作为成员函数的隐藏参数，指向调用者对象
2. 解决重名歧义，当然_方法更好

e.g. pk

```
const person& pk(const person& p) const
{
if (p.age < age) return p;
return *this; //对象
}
...
const person &winner = p1.pk(p2).pk(p3).pk(p4).....
```

静态成员 static

1. 用于多个实例对象共享数据，不创建对象也可以访问 -
2. 不在创建对象的时候初始化，在全局区初始化
3. 静态成员是全局的，不是对象的，程序中仅此一份

- 相对应的，静态成员函数只能访问静态成员，而对象可以访问静态成员和对象成员
- 也因此，该函数无this指针

```
class Person
{
private:
static int age_sum_; // 私有的静态成员在类外无法访问
static const int whatever = 1; // C++11特性，静态成员变量的内联初始化；可读性upup,避免重定义

public:
static void show_age()
{
cout << age_sum_ << "\n";
}
};
// 定义&初始化，非访问，故不会违背私有性
// 为避免重定义，经常放在源文件cpp中
int Person::age_sum_ = 1; // 缺省为0

int main()
{
Person::show_age();

return 0;
}
```

C++对象模型

1. OOP思想，C语言只有变量和函数，没有什么成员变量和成员函数
   - C++没什么新的，还是通过指针联系成员变量和成员函数，不过隐藏了实现方法罢了
2. 对象中维护了多个指针表，表中放了成员与地址的对应关系

3. 对象内存包括：

4. 非静态成员变量

5. 内存对齐的额外消耗

6. 为了支持 virtual 成员而产生的额外负担

7. 无论是什么函数，在内存中仅一份，放在代码段，通过指针使用

- 静态成员变量也一份，放在全局区

5. 用空指针可以调用没有用到 this 指针的非静态成员函数

- 编译器会自动为成员函数中的成员变量添加this→

```
class Person
{
public:
void show_age()
{
if (this == nullptr) // 常见规避空指针访问方法
return;
cout << "whatever\n";
}
};

int main()
{
Person *p;
p->show_age(); // 未创建对象时，this指针为空，如果成员函数中有成员变量，此时等于访问空指针，访问前的部分都可以正常运行

return 0;
}
```

6. 对象的地址是第一个非静态成员变量的地址，如果类中没有非静态成员变量，编译器会隐含的增加一个 1 字节的占位成员

友元

为访问类的私有成员提供方法

友元全局函数

只要访问私有成员的 函数/对象 是friend就行

```
class Person
{
friend int main(); // main函数可以访问Person的全部成员

private:
int age;

public:
Person() : age(1) {}
};
```

友元类

1. 同上，不过需要注意，友元关系不能继承，且是单向的
2. 也即：我把你当朋友，你可以访问我的私有成员，你不把我当朋友，我访问不了你的私有成员

友元成员函数

1. 我把你当朋友，但我是一个有原则的人，你只能在某些情况下看我的私有成员
2. 用法同上，不过需要注意声明顺序：

```
// 类B的成员函数需要使用类A的私有成员

class A; // 先声明A——B要用到


class B
{
public:
void show_A_info(const A &a);
};


class A
{
friend void B::show_A_info(const A &a); // 友元函数

private:
int age;

public:
A() : age(0) {}
};


void B::show_A_info(const A &a) // 定义A后才能定义该友元函数，因为要用到A中信息
{
cout << a.age << endl;
}
```

运算符重载

重载函数的返回值应与其含义与用法一致，如加法：

```
class Person
{
private:
int score_;

public:
Person() : score_(0) {}

friend Person &operator+(int, Person &);
friend Person &operator+(Person &, int);
friend Person &operator+(Person &, Person &);
};

Person &operator+(int score, Person &p)
{
p.score_ += score;
return p;
}
Person &operator+(Person &p, int score)
{
p.score_ += score;
return p;
}
Person &operator+(Person &p1, Person &p2)
{
p1.score_ += p2.score_;
return p1;
}
```

1. 运算符重载函数的参数，至少有一个自定义类型
2. 无法创建新的运算符，部分运算符无法重载，部分运算符只能重载为成员函数

==、!=、>、>=、<、<=

```
class Actor
{
private:
int perfomance_;
int fame_;
int age_;

public:
bool operator==(const Actor &actor)
{ // 自定义比较价值方式，其他关系运算符同理
return (perfomance_ + fame_ + age_ - actor.age_ - actor.fame_ - actor.perfomance_) ==
0;
}
};
```

<<

1. 主要用于写日志，方便调试，类似于cout<<
2. 只能使用非成员函数版本——成员函数的第一个参数一定是本对象
3. 可配合友元函数使用

```
class Actor
{
friend ostream &operator<<(ostream &cout, const Actor &actor);

private:
int perfomance_;
int fame_;
int age_;

public:
Actor() : perfomance_(0), fame_(0), age_(0) {}
};

ostream &operator<<(ostream &cout, const Actor &actor)
{
cout << "年龄： " << actor.age_
<< " 演技： " << actor.perfomance_
<< " 声望： " << actor.fame_ << "\n";
return cout;
}
...
cout << g << endl;
```

[]

- 操纵对象中的数组
- 需要提供两种，以适应**常对象**

```
class fansName
{
private:
string fans_name_[3];

public:
fansName() : fans_name_({"a", "b", "c"}) {}

string &operator[](int i)
{
return fans_name_[i];
}

const string &operator[](int i) const
{
return fans_name_[i];
}
};
```

=

1. 为**默认赋值函数**，缺省时编译器会提供--多对多分别赋值，且为**浅拷贝**——对象中有堆内存空间时，无法满足要求，需要**重载进行深拷贝**
2. **赋值运算和拷贝构造函数有本质不同**：

- 赋值运算：存在两个对象，一个对象的成员的值给另一个对象
- 拷贝构造：用已存在的对象创建新的对象

```
class fansName
{
public:
string *fans_name_;
int id_;

fansName() : fans_name_(nullptr), id_(0) {}
~fansName()
{
if (fans_name_)
delete fans_name_;
}

fansName &operator=(const fansName &fn)
{
// 自身
if (this == &fn)
return *this;

// 对方为空
if (fn.fans_name_ == nullptr)
{
if (this->fans_name_ != nullptr) // 自身不为空删除自身
{
delete this->fans_name_;
this->fans_name_ = nullptr;
}
}
else
{
if (this->fans_name_ == nullptr) // 自身为空先分配内存
this->fans_name_ = new string;
memcpy(fans_name_, fn.fans_name_, sizeof(string));
}

id_ = fn.id_; // 赋值其他
return *this;
```

```
}
};
```

new delete

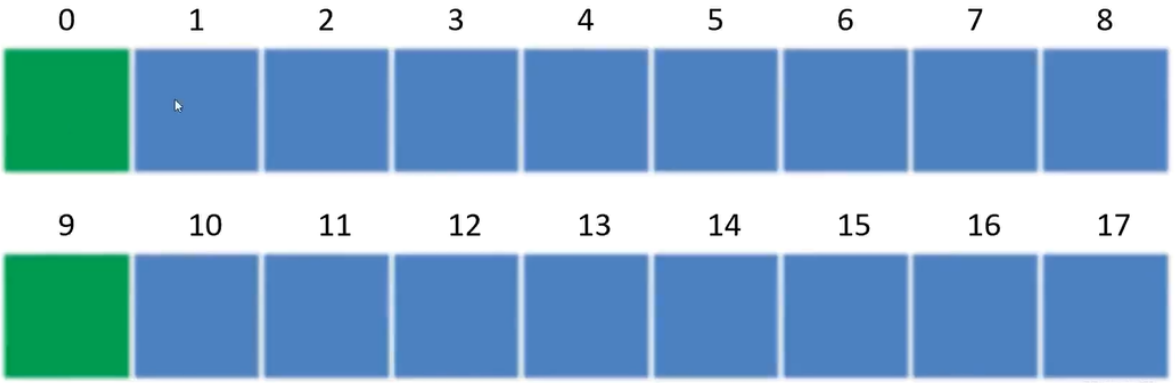
1. new: 先分配内存，再构造
2. delete: 先析构，再free
3. 默认为static类型，不必显示写出，不能访问非static成员
4. new[]和delete[]重载少见，暂放

```
class fansName
{
public:
void *operator new(size_t size)
{
void *ptr = malloc(size);
return ptr;
}
void operator delete(void *ptr)
{
if (ptr == nullptr)
return;
free(ptr);
}
};
```

内存池

1. 预先分配一大块连续空间，使用时从内存池中借还
2. 可以提升分配和归还速度，并减少内存碎片

简单的内存池



如果没有被占用，就把标志位后面这个内存的地址返回

```
class Pool
{
private:
int a, b; // 类大小为8B
static char *pool_; // 书写方便

public:
bool initPool()
{
pool_ = (char *)malloc(18); //(1+8)*2
if (pool_ == 0) // 分配失败
return false;
memset(pool_, 0, 18);
return true;
}

void freePool()
{
if (pool_ == nullptr)
return;
free(pool_);
}

void *operator new(size_t size)
{
if (pool_[0] == 0)
{ // 分配第一块内存
pool_[0] = 1;
return pool_ + 1;
}

if (pool_[9] == 0)
{ // 分配第二块内存
pool_[9] = 1;
return pool_ + 9 + 1;
}

// 均不可用，三种做法：扩展、直接申请、返回空
void *ptr = malloc(size);
```

```
return ptr;
}

void operator delete(void *ptr)
{
if (ptr == nullptr)
return;
if (ptr == pool_ + 1)
{
pool_[0] = 0; // 不使用free，提高速度
return;
}
if (ptr == pool_ + 10)
{
pool_[9] = 0;
return;
}
free(ptr);
}
};
```

()

1. 对象当成函数用：函数对象，仿函数
2. 只能用成员函数重载
3. 具备普通函数全部特征，调用函数时根据作用域选择--想调用全局的怎么办？手动加::

++ -- ! & ~ * 一元加减

1. 一元运算符都没有参数
2. ++后置没必要嵌套
3. 深坑

```
class Person
{
private:
int rank_; // 类大小为8B

public:
Person &operator++()
{ // 前置
rank_++;
return *this;
}

Person operator++(int)
{ // 后置，维护语义，无引用！
Person tmp = *this; // 保存当前状态，深坑
rank_++;
return tmp;
}
};
```

类型转换

自动类型转换

1. 只会自动转换兼容的类型，如不允许：int* p = 8;，但可强制转换
2. 类型转化为类类型是有意义的，如：

```
string str = "哈哈";
```

3. 通过一个参数的**构造函数实现**——可重载；多参数情况下，其他参数有默认值也可作转换函数
4. 转换的各种情形如下：

```
class Person
{
private:
int rank_;

public:
Person() : rank_(0) {}
Person(int rank) : rank_(rank) {}
};
Person func(Person p)
{
return 'c'; // 也是ok的
}

int main()
{
Person p = 1;
Person p2;
p2 = 2;
func(8);

return 0;
}
```

但是注意，自动转换若出现二义性则会报错

5. **不希望自动转换**，但允许显示转换

```
class Person
{
private:
int rank_;

public:
explicit Person(int rank) : rank_(rank) {}
};

int main()
{
Person p1 = (Person)1;
Person p2(1);
Person p3 = Person(1);
// Person p4 = 1; // 不允许隐式转换

return 0;
}
```

强调构造时，一般加上explicit，强调转换则不加

转换函数

- 必须是类的成员函数，无参数，无返回值
- 可用于隐式or显示转换，隐式转换有二义性的时候会报错，可以通过显示转换的方式规避；可设置函数使其只能进行显示转换，也可通过成员函数的方法实现
- 警惕隐式转换，极易导致意图与实际实现不符

```
class Person
{
private:
int rank_;
double score_;
string name_;

public:
explicit Person(int rank) : rank_(rank) {}
Person(int rank, double score) : rank_(rank), score_(score), name_("") {}

operator int() { return rank_; }
operator double() { return score_; }

explicit operator string() { return name_; } // 只能显示转换

int to_int() { return rank_; } // 成员函数实现只能显示转换
};

int main()
{
int a = Person(1); // 隐式

Person g(2);
int b = (int)g; // 显示也ok

// short c = Person(1, 1.1); // 二义性时报错
short c = (int)Person(1, 1.1); // 显示转换规避二义性

return 0;
}
```

继承

- 代码复用，获取基类的成员变量和函数，在基类的基础上进行扩展

继承方式

1. **public 成员在类外可以访问**；protected **可以在子类中访问**，不能在类外访问；**private 成员只能在类内访问**，但是也可以通过有权限的函数在类外访问--也占用内存
2. 继承时，基类成员在派生类中的**访问权限不得高于**继承方式中指定的权限

- 高的会降级，低的不变 —— public、protected、private 是用来指明基类成员在派生类中的最高访问权限的

3. 由于 private 和 protected 继承方式会改变基类成员在派生类中的访问权限，导致继承关系复杂， 所以，**在实际开发中，一般使用 public**
4. **using**可以随意改变基类public、protected成员在子类中的权限，但private碰都不能碰

继承对象模型

1. 调用**构造函数--先父后子**；调用**析构函数--先子后父**
2. 对于构造，子类很可能依赖于父类，故先构造父类
3. 对于析构，同理，必须先销毁依赖的部分，再销毁被依赖的部分
4. **创建子类对象只会申请一块内存**，其大小等于子类新成员+父类成员的大小(包含父类的私有成员，尽管它在子类中不可见)
5. 不同继承方式的访问权限只是语法上的处理，可以**通过指针突破权限限制**

```
class A
{
public:
int b, c; // 变量地址与声明顺序有关
A() : a(0), b(0), c(0) {}
void showInfo()
{
cout << "a = " << a << endl;
cout << "b = " << b << endl;
cout << "c = " << c << endl;
}

private:
int a;
};
class B : public A
{
public:
int d;
B() : d(0) {}
void showMyInfo()
{
cout << "d = " << d << endl;
}
};

int main()
{
B obj; // a地址相对于对象地址偏移2个int
*((int *)&obj + 2) = 2;
obj.showInfo();
obj.showMyInfo();

return 0;
}
```

也因此memset等函数也会操纵私有成员的值

基类构造函数

1. 如若未指定基类构造函数，使用默认基类构造函数
2. 也可以使用**初始化列表指定**基类的构造函数
3. **谁的成员谁初始化**：代码重用+私有无法访问

```
class A
{
public:
int a_;
A() : a_(0) {}
A(int a) : a_(a) {}
A(const A &o) : a_(o.a_) {}
};
class B : public A
{
public:
int b_;
B() : b_(0) {}
B(int a, int b) : A(a), b_(b) {}
B(const B &o) : A(o), b_(o.b_) {}
};
```

名字遮蔽与类作用域

1. 子类成员与父类**成员重名**时，**不会覆盖**父类成员，二者**同时存在**，**函数不形成重载**，在子类成员函数中使用时**优先使用子类的成员**
2. 通过**域解析符**使用父类被遮蔽的成员

```
class A
{
public:
void func() { cout << "a"; }
};

class B : public A
{
public:
void func() { cout << "b"; }
};

class C : public B
{
public:
void func() { cout << "c"; }
};

int main()
{
C c;

c.A::func();
c.B::A::func(); // 建议写法，展现关系

c.B::func();

c.func();

return 0;
}
```

继承的特殊关系

数据类型决定数据的操纵方法

1. **B is an A**:
2. 可以把子类对象赋值给父类对象(包括私有成员)，但会**舍弃非基类**成员
3. **基类指针**可以在不进行显式转换的情况下**指向派生类对象**，但会舍弃~

4. **基类引用**可以在不进行显式转换的情况下**引用派生类对象**，但会舍弃~
5. 也因此，可以用**派生类构造基类**，可以用**派生类作实参赋给基类的形参**
6. C++要求指针和引用类型与赋给的类型匹配，这一规则对继承来说是例外。但是，这种例外只 是单向的，不可以将基类对象和地址赋给派生类引用和指针

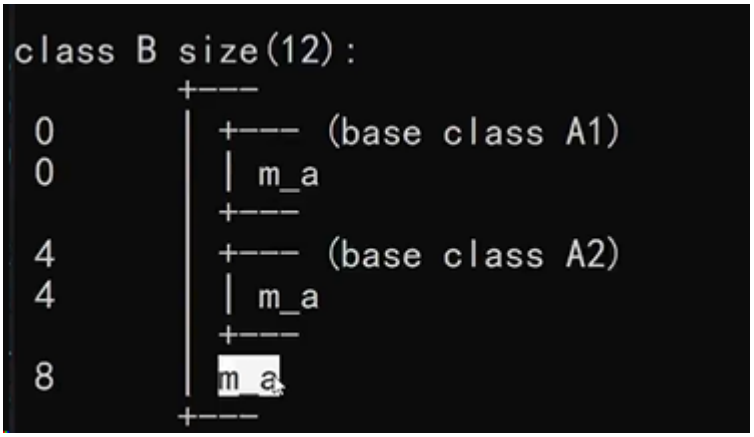
```
class A
{
public:
int a_;
A() : a_(0) {}
};

class B : public A
{
public:
int b_;
B() : A(), b_(0) {}
};

void func(A a) {}

int main()
{
A a;
B b;
a = b; // b is an a
// a.b_; // 报错，舍弃非基类成员

A *aPtr = &a;
B *bPtr = &b;
aPtr = bPtr; // 允许
// bPtr = aPtr; // 错误

// aPtr->b_; // 没有成员b_，报错

func(b); // 派生类作实参赋给基类的形参
A ao(b); // 派生类对象构造基类对象

return 0;
```

多继承与虚继承

99%情况下的多继承：

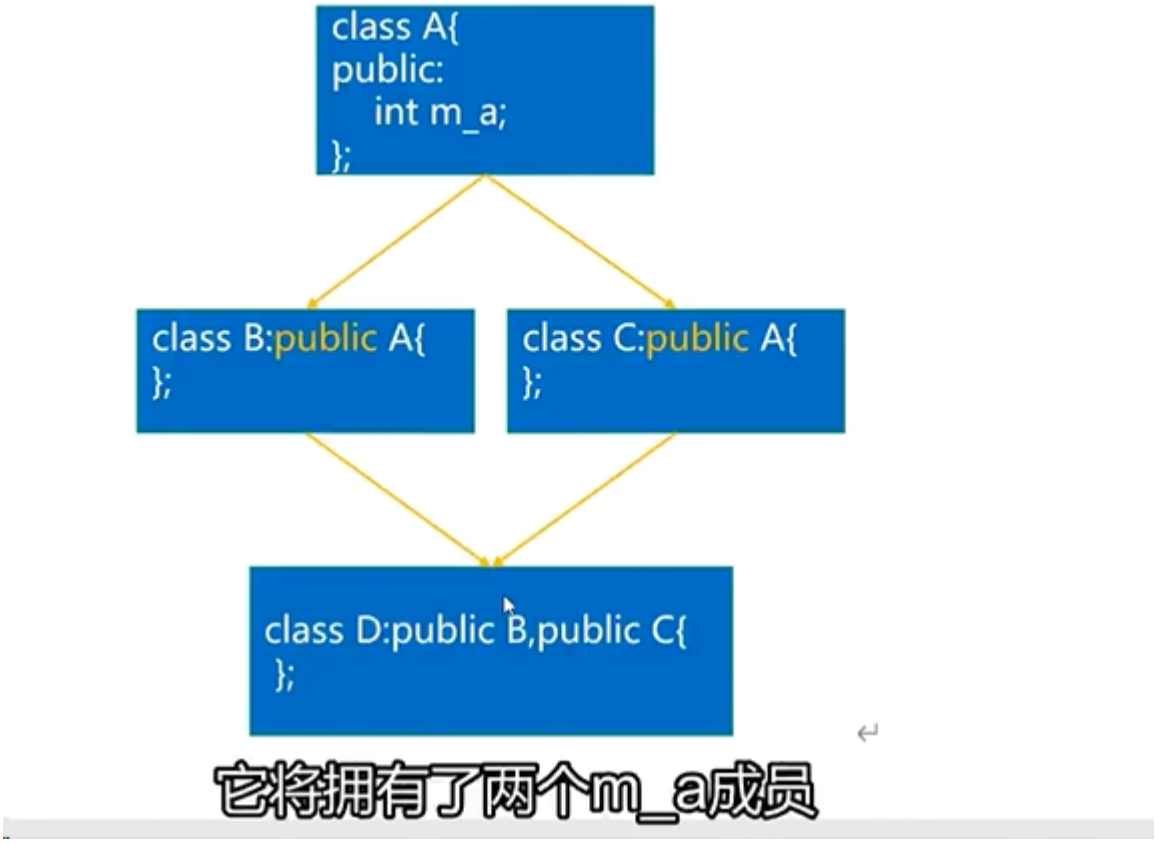
```
class A1
{
public:
int a1_;
A1() : a1_(0) {}
};

class A2
{
public:
int a2_;
A2() : a2_(0) {}
};
class B : public A1, public A2
{
public:
int b_;
B() : b_(0) {}
};
```

有例外就用域解析符



菱形继承

- **造成数据冗余以及二义性**



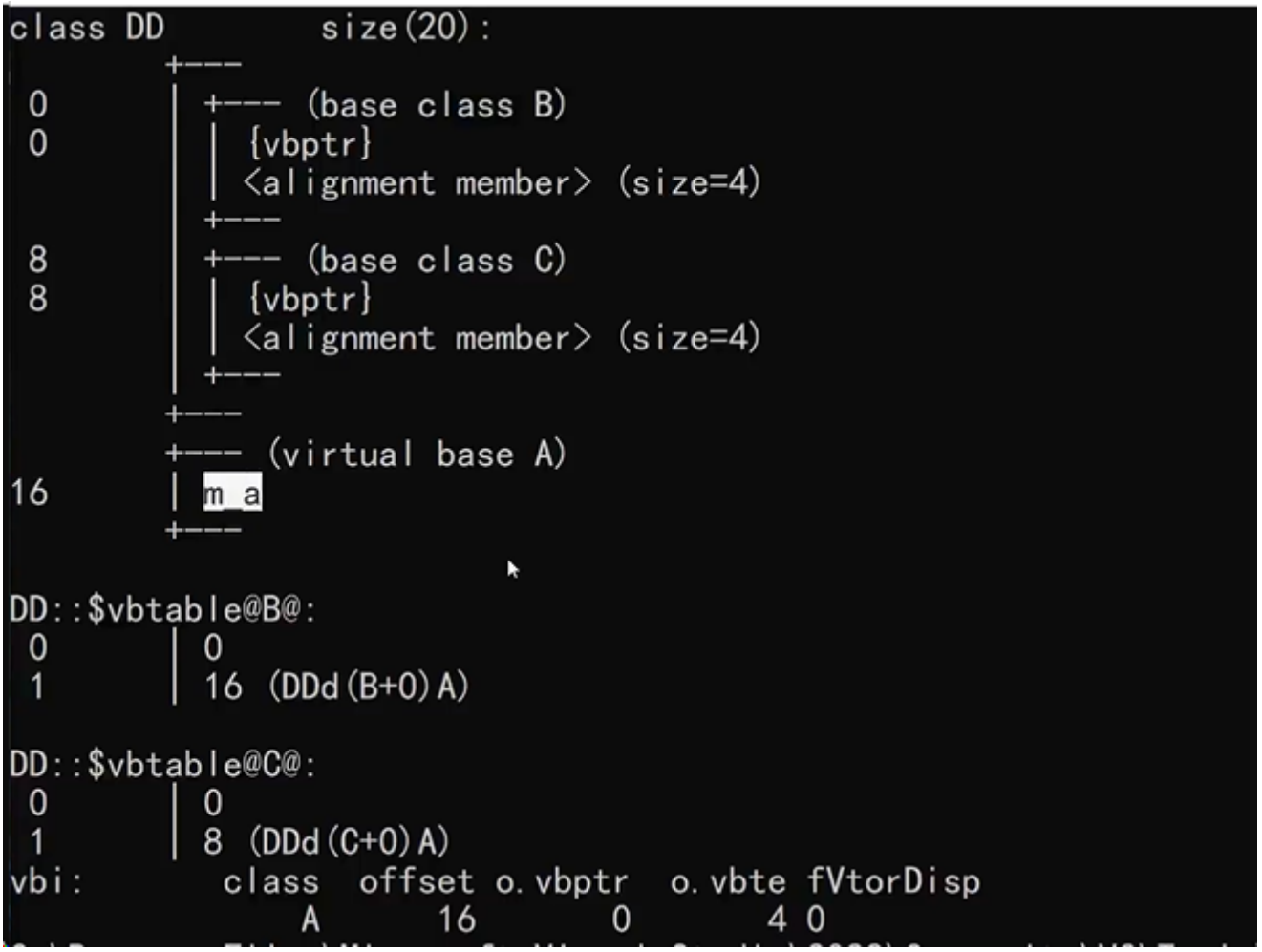

```
D d;
// d.m_a; // 不明确，需要使用域解析符指明
d.B::m_a;
d.C::m_a;
```

但其实是不需要两个m_a的，这是一种**数据冗余**，也造成了**二义性**

虚继承

- 解决上述问题，此时两个m_a是同一个东西
- 不提倡使用多继承，过于复杂，且bug很多

```
class A
{
public:
int m_a = 10;
};
class B : virtual public A {};
class C : virtual public A {};
class DD : public B, public C {};
int main()
{
DD d;
// d.B::m_a = 30;
// d.C::m_a = 80;
d.m_a = 80;
cout << "B::m_a 的地址是： " << &d.B::m_a << "，值是： " << d.B::m_a << endl;
cout << "C::m_a 的地址是： " << &d.C::m_a << "，值是： " << d.C::m_a << endl;
}
```



多态

使基类指针可以使用子类函数——呈现出多种形式，即多态；引用也同理

1. 有函数声明则函数定义不能加virtual，无则可以
2. **函数特征需要相同**，才能够成多态；如果没有重定义，则使用基类函数
3. 即便定义了，也可以通过**域解析符**调用基类函数
4. **语义要求**：只有当需要重定义时，才声明为virtual，否则不声明——而且不声明的效率也更高

应用场景

王者荣耀举例：

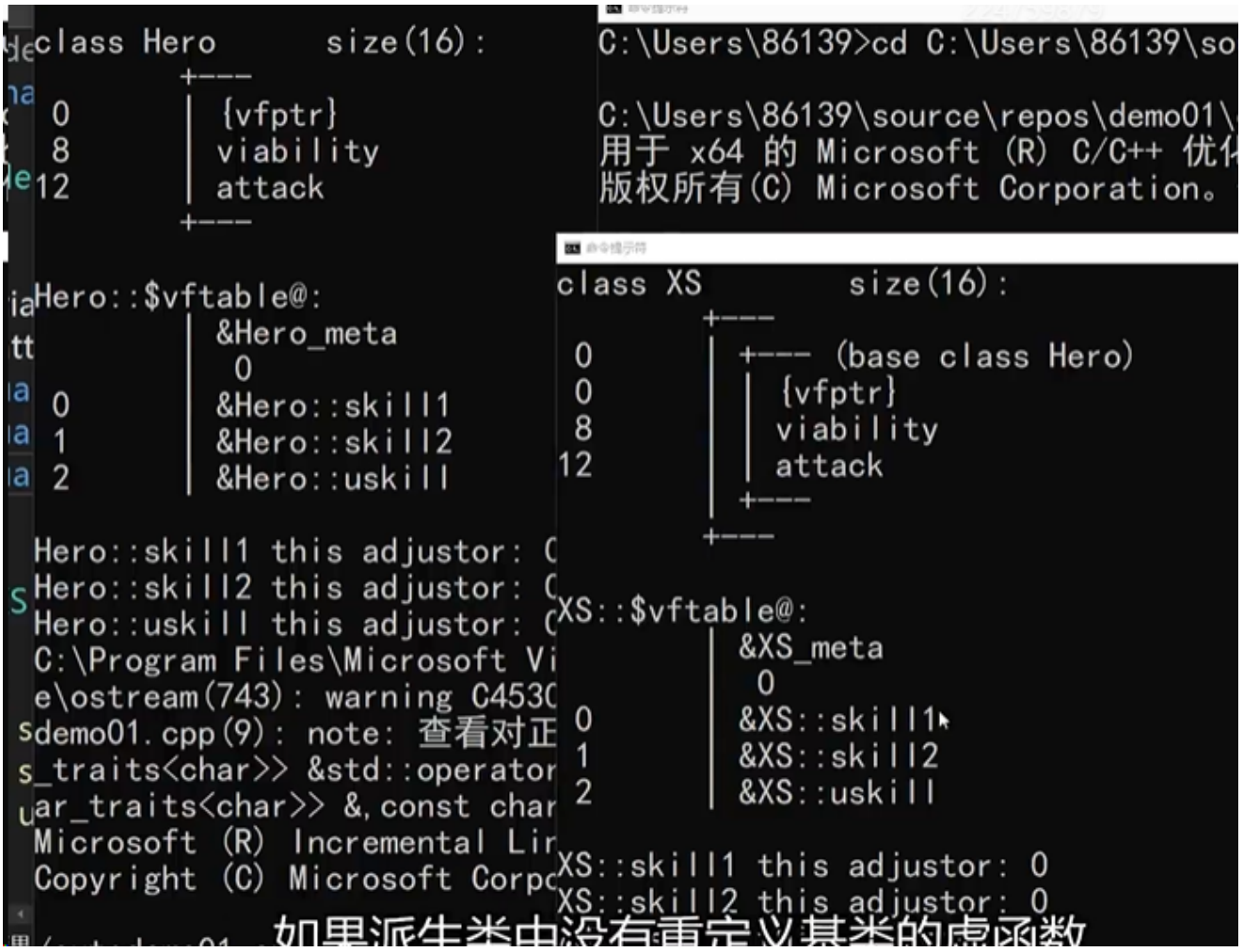
```
class Hero
{
private:
int damage_;
int viability;

public:
virtual void skill1() { cout << "英雄一技能" << endl; } // 虚函数
virtual void skill2() { cout << "英雄二技能" << endl; }
virtual void skill3() { cout << "英雄三技能" << endl; }
};

class LiBai : public Hero
{
public:
void skill1() { cout << "李白一技能" << endl; }
void skill2() { cout << "李白二技能" << endl; }
void skill3() { cout << "李白三技能" << endl; }
};

class Xishi : public Hero
{
public:
void skill1() { cout << "西施一技能" << endl; }
void skill2() { cout << "西施二技能" << endl; }
void skill3() { cout << "西施三技能" << endl; }
};

class Mengya : public Hero
{
public:
void skill1() { cout << "蒙犴一技能" << endl; }
void skill2() { cout << "蒙犴二技能" << endl; }
void skill3() { cout << "蒙犴三技能" << endl; }
};

int main()
{
Hero *ptr;
```

```
int herold;
cin >> herold;

if (herold == 1)
ptr = new LiBai;
else if (herold == 2)
ptr = new Xishi;
else if (herold == 3)
ptr = new Mengya;

ptr->skill1(); // 重点
}
```

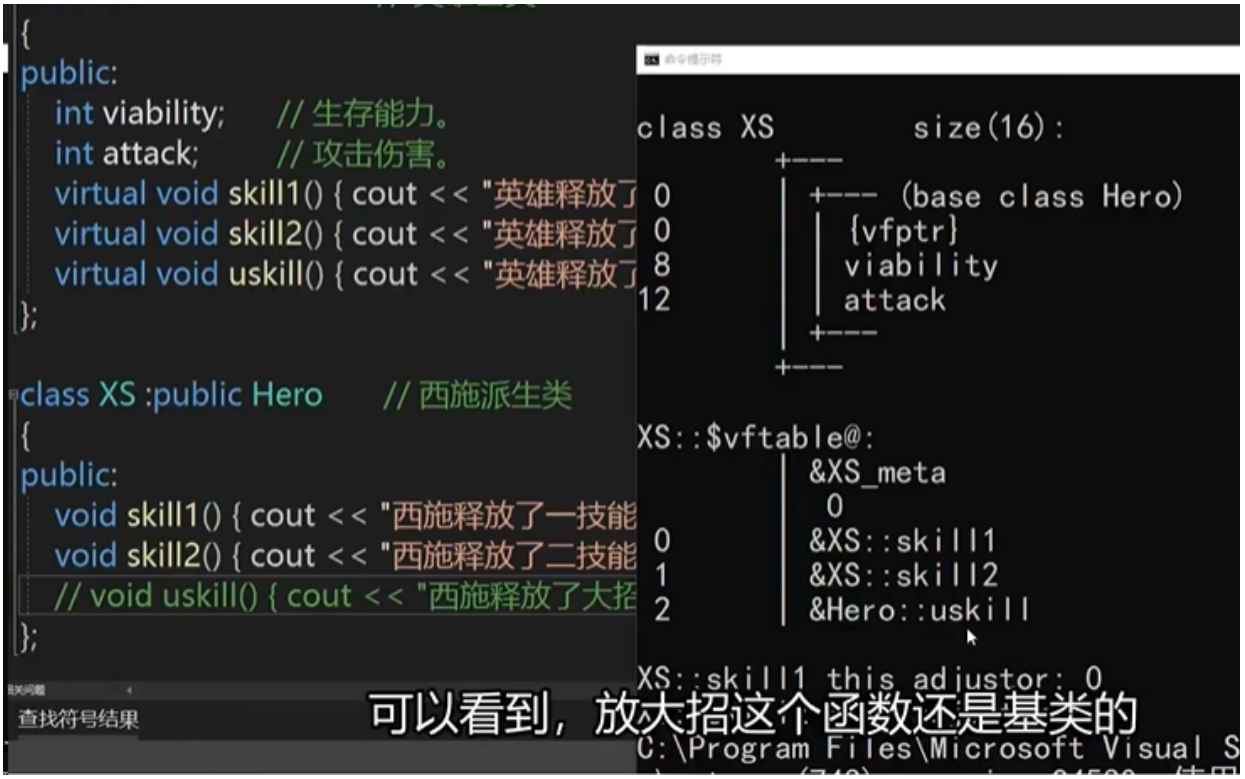
多态对象模型

1. 对象内存多了个隐藏的**虚函数指针**，指向**虚函数表**，虚函数表中有基类的函数名和地址
- 普通成员函数的地址早已链接进二进制文件中，可以直接执行；虚函数还要查虚表才能执行，效率低

2. **多态原理**：派生类会从基类继承这个vfptr，但是会用派生类的虚函数表替代虚函数表，于是调用虚函数的时候便通过虚函数表调用到了派生类的函数



3. 也因此，如果**派生类中没有重定义**，这个虚表还是基类的虚表



4. C语言就是这样，使用**函数指针实现多态**

5. 静态多态：在编译时期就已经确定要执行了的函数地址了，如有函数重载 和函数模板

6. 动态多态：运行时确定~

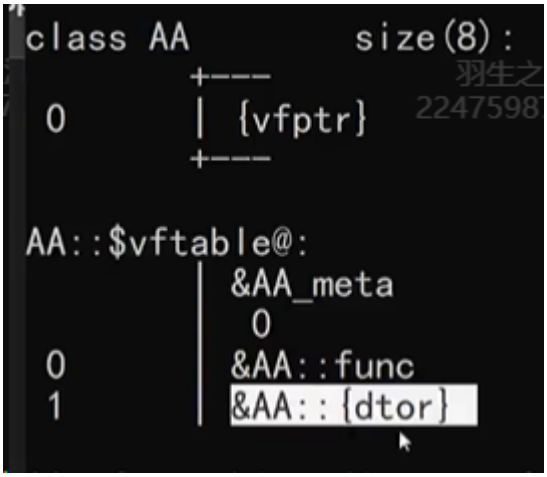
派生类析构

1. 执行完派生类的析构函数后(哪怕是手动调用)，会强制**自动调用基类的析构函数**

2. 在多态情境下，**使用delete时，并不会调用派生类的析构函数**，解决方法：

```
class A
{
public:
virtual ~A() {}
};
class B : public A
{
public:
~B() {}
};
...
A *a = new B;
delete a;
```

虽然这俩析构函数不重名，但是**编译器对他们做特别的处理**--这还是虚函数！编译器会调用~B，然后自动调用~A



其他细节

1. 构造函数、析构函数、友元函数、赋值运算符函数**不能继承**
2. 析构函数可以手动调用，会delete动态分配的内存，同时也**必须置空ptr**，否则在自动调用析构函数时，会delete野指针，导致程序崩溃
3. **基类即便不需要析构函数，也应提供空虚析构函数**，否则在多态情境下，销毁对象时，将无法调用析构函数

纯虚函数和抽象类

有一种场景，基类完全不需要实现虚函数——完全用不到，比如说英雄联盟，必须要选择一个英雄

这种情况下，基类中的该函数给个名字就好了

```
virtual void func(int a) = 0;
virtual void func(int a) = 0 {} // 也可以实现
virtual ~A() = 0 {} // 纯虚析构函数必须实现，但好像是多余的...直接用析构不就得了...原因是想使一个类成为抽象类，不能实例化
```

1. 而这种函数，**必须实现后才可以实例化**，否则就是抽象类，不允许实例化
2. **纯虚析构函数必须实现，但好像是多余的**...直接用析构不就得了...原因是想**使一个类成为抽象类**，不能实例化

运行阶段类型识别 dynamic_cast

多态产生了一个问题：如何识别基类指针用的是哪个子类？

C++提供了dynamic_cast，用指向基类的指针来生成派生类的指针，它不能回答“指针指向的是什么类的对象”的问题，但能回答“是否可以**安全的将对象的地址赋给特定类的**

指针”的问题。

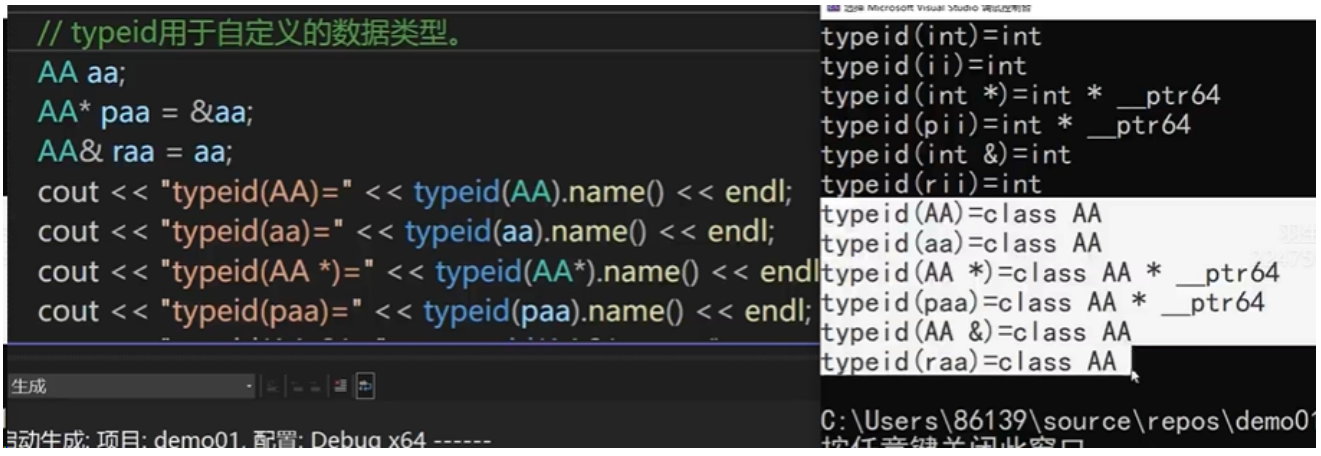
1. 转换失败则返回**空地址**
2. **只适用于多态情景**，也即有虚函数的情景
3. 派生类指针可以直接转换为基类指针，无需cast
4. 可以用于引用，但是失败会有异常

```
Libai *lbptr = dynamic_cast<Libai*> (ptr);
if (lbptr != nullptr){
//xxx
}
```

typeid 和 type_info

typeid 运算符返回 type_info 类（在头文件中定义）的对象的引用。

type_info 类的实现随编译器而异，但至少有 name()成员函数，该函数返回一个字符串，通常是类 名。



类型比较

```
if (typeid(AA) == typeid(aa)) cout << "ok1\n";
```

1. type_info 类的构造函数是 private 属性，也没有拷贝构造函数，所以**不能直接实例化**，只能由 编译器在内部实例化。
2. **不建议用 name()**成员函数返回的字符串作为判断数据类型的依据。（编译器可能会转换类型名）
3. typeid 运算符可以**用于多态**的场景，在运行阶段识别对象的数据类型，但注意，基类指针指向派生类对象，其**类型还是基类指针**
4. 注意验证指针不是**空指针**再使用typeid(*ptr)

模板

自动推导类型

C++11，根据等号右边的类型自动推导类型

1. auto类型必须**定义时初始化**
2. auto不能做形参，不能直接声明数组，不能定义类的非静态成员变量(static的类外定义，可以确定大小)

- **其原因均是，编译器无法确定声明变量的大小**

3. 不要**滥用auto**，应用于

- 代替冗长复杂的变量声明
- 在模板中，用于声明依赖模板参数的变量
- 函数模板依赖模板参数的返回值
- 用于 lambda 表达式中

```
// 自动推导函数指针
double func(double b, const char* c, float d, short e, long f)
{
cout << ",b=" << b << ",c=" << c << ",d=" << d << ",e=" << e << ",f=" << f << endl;
return 5.5;
}
auto pf = func;
// 对比
// double (*pf)( double , const char* , float , short , long );
// pf = func;
```

函数模板

编译的时候，编译器**推导实参的数据类型**，根据实参的数据类型和函数模板，**生成**该类型的函数定义。

```
template <typename T>
void Swap(T &a, T &b)
{
T tmp;
tmp = a;
a = b;
b = tmp;
}
// 手动指定模板类型
Swap<string>(a, b);
```

建议函数模板用typename，类模板用class

1. 可以为类的成员函数创建模板，但**不能是虚函数和析构函数**，析构函数没有参数不需要模板，虚函数无法确定正确的虚表
2. 模板也需要参数，须确保**实参与函数模板能匹配上**(int char匹配一个T，无参匹配T :x:)
3. **数据类型应符合模板函数的的逻辑**(如不能进行加法的类型，传递给加法函数)

4. **自动推导不隐式转换，指定模板参数会发生隐式转换**

5. 多模板

```
template<typename T1, typename T2>
void printPair(T1 first, T2 second) {
std::cout << "First: " << first << ", Second: " << second << std::endl;
}
```

6. 函数模板重载

```
template <typename T>
void show(T &a, T &b) {}

template <typename T1,typename T2>
void show(T1 &a, T2 &b) {}

template <typename T1,typename T2>
void show(T1 &a, T2 &b, int c) {}
```

7. 函数定义和函数声明可以分开写，但是**均不能省略模板身份**

特化函数模板

可以**提供一个具体化的函数定义**，当编译器找到与函数调用匹配的具体化定义时，将使用该定义， 不再寻找模板

```
class Person
{
public:
int a_, rank;
Person() : a_(0) {}
Person(int a) : a_(a) {}
};

template <typename T>
void Swap(T &a, T &b)
{
T tmp = a;
a = b;
b = tmp;
}

template <>
void Swap(Person &a, Person &b)
{
Swap(a.a_, b.a_); // 满足业务需求即可
a.rank = b.rank = 0;
}

int main()
{
Person a = 1, b = 2;
Swap(a, b);
cout << a.a_ << " " << b.a_;
}
```

函数调用规则

1. 普通函数 > 具体化函数模板 > 常规模板
2. 三者皆有情况下，可以通过加<>指定调用模板函数(仍是具体化 > 普通)

```
swap<>(a, b);
```

3. 而如果函数模板可以产生更好的匹配，将使用函数模板

```
void Swap(int&, int&);

template <>
void Swap(int &a, int &b);

template <typename T>
void Swap(T &a, T &b);
...
int a = 1, b = 1;
char c = 'c', d = 'd';
Swap(a, b); // 1
Swap(c, d); // 3
Swap<>(a, b); // 2
```

函数模板分文件编写

1. 函数模板只是函数的描述，**没有实体**，则将创建函数模板的代码放在头文件中
2. 函数模板的具体化有实体，**编译的原理和普通函数一样**，所以，声明放在头文件中，定义放在源文件中。

```
#pragma once

#include <iostream>
using namespace std;

// 函数模板放在头文件中
template <typename T>
void Swap(T &a, T &b)
{
T tmp = a;
a = b;
b = tmp;
}

// 具体化函数模板定义放在源文件中
template <>
void Swap(int &a, int &b);

// 普通函数定义放在源文件中
void Swap(int &a, int &b);
```

高级函数模板

decltype 关键字

C++11，用于**查询表达式的数据类型**，但**不会执行**表达式

```
decltype(expression) var;
```

1. expression**不加括号**，var的类型与其相同
2. expression**是函数调用**，var与返回值类型相同(不能是void)
3. expression**是左值**，且加括号(如(a))，var是其的引用
4. 都不满足则是表达式的类型，如**x+1**

与auto不同，auto需要初始化，decltype不需要

可结合using和typedef

函数后置返回类型

```
template <typename T1, typename T2>
auto func(T1 x, T2 y) -> decltype(x + y) // 后置在作用域内
{
auto tmp = x + y; // x为char， y为int这样处理
return tmp;
}
```

C++14 auto

上面的可以变成：

```
template <typename T1, typename T2>
auto func(T1 x, T2 y)
{
auto tmp = x + y;
return tmp;
}
```

类模板

```
#include "public.hpp"

template <class T1, class T2 = string> // 缺省类型；函数模板的缺省类型意义不大
class AA
{
public:
T1 m_a; // 模板可以用于类中的任何地方
T2 m_b;

AA() {}
AA(T1 a, T2 b) : m_a(a), m_b(b) {}

T1 geta()
{
T1 a = 2;
return m_a + a;
}
T2 getb();
};

// 成员函数类外实现
template <class T1, class T2> // 不能在类外部指定缺省模板参数
T2 AA<T1, T2>::getb()
{
return m_b;
}

int main()
{
AA<int, int> a; // 必须指明类型，不存在自动推导这一说

AA<int, string> *a = new AA<int, string>(3, "haha"); // new创建
// 卧槽，java
}
```

1. 另外，传入的模板参数需要**符合模板类逻辑**

2. 模板类的成员函数**只有类实例化了，才会创建**

模板类举例

先写普通类，调试好后写模板类

非通用类型模板参数

1. 通常整型，常量表达式，不能修改参数的值
2. 但会生成**不同的类**

```
template <class T, int len = 10>
class Array
{
private:
T items[len];
};

// 更好的办法
template <class T>
class Vector
{
private:
T *items;
int len;

public:
Vector(int size = 10) : len(size) { items = new T[len]; }

void resize(int size)
{
if (size <= len)
return;
// xxx
}
};
```

嵌套和递归使用模板类

容器中有容器；

```
Vector<Stack<string> > vs; // C++11前，>>要分隔开
// 均动态扩展，注意resize时的浅拷贝问题——需要重写赋值构造函数
Stack<Vector<string> > sv;
Vector<Vector<string> > vv; // 第二维大小不固定

Stack &operator=(const Stack &v) // 重载赋值运算符函数，实现深拷贝。
{
delete[] items; // 释放原内存。
stacksize = v.stacksize; // 栈实际的大小。
items = new DataType[stacksize]; // 重新分配数组。
for (int ii = 0; ii < stacksize; ii++)
items[ii] = v.items[ii]; // 复制数组中的元素。
top = v.top; // 栈顶指针。
return *this;
}
```

模板类具体化

完全具体化，部分具体化

1. 函数模板无部分具体化
2. 具体化程度高的 > 具体化程度低的

```
template <class T1, class T2>
class AA{
//xxx
};

template <>
class AA<int, string>{
//xxx
}

template <class T1>
class AA<T1, string>{
//xxx
}
```

模板类与继承

模板类 继承 普通类

平平无奇

```
class A
{
private:
int a_;

public:
A(int a) : a_(a) {}
};

template <class T1, class T2> // note
class B : public A
{
private:
int b_;

public:
B(int a, int b) : A(a), b_(b) {}
};
```

普通类 继承 模板类-实例化版本

继承的时候实例化一下

```
template <class T>
class A
{
private:
T a_;

public:
A(T a) : a_(a) {}
};

class B : public A<int> // note
{
private:
int b_;

public:
B(int a, int b) : A(a), b_(b) {}
};
```

普通类 继承 模板类

变成模板类

```
template <class T> // note
class A
{
private:
T a_;

public:
A(T a) : a_(a) {}
};

template <class T>
class B : public A<T> // note
{
private:
int b_;

public:
B(T a, int b) : A<T>(a), b_(b) {} // note
};
```

模板类 继承 模板类

加个参数

```
class A
{
private:
T a_;

public:
A(T a) : a_(a) {}
};

template <class T, class T1>
class B : public A<T>
{
private:
T1 b_;

public:
B(T a, T1 b) : A<T>(a), b_(b) {}
};
```

模板类 继承 模板参数给出的基类

该基类不能是模板类，但可以是具体化的模板类

```
class A
{
public:
A() { cout << "A" << endl; }
};

template <class T>
class B
{
public:
B() { cout << "B" << endl; }
};

template <class T>
class C : public T
{
public:
C() : T() { cout << "C" << endl; }
};

int main()
{
C<A> c;
C<B<int>> c2;
// C<B> c; // 错误
}
```

模板类与函数

优先级同理，越抽象，优先级越低

```
template <class T1, class T2>
class B
{
public:
B() { cout << "B" << endl; }
void show() {}
};

// 普通函数版本
B<int, string> func(B<int, string> v) { v.show(); }

// 函数模板版本，只支持B的函数模板版本
template <typename T1, typename T2>
B<T1, T2> func2(B<T1, T2> o) { o.show(); } // 不规范的写法

// 规范的写法，支持任意类型，体现模板精髓--相似逻辑提取
template <typename T>
T func3(T a) { a.show(); }
```

模板类与友元

有多种方案，为一个模板类生成相应的友元函数

模板具体化友元函数

不再赘述，麻烦且毫无意义

非模板友元函数

比上一种好一点，但是还是得多次定义

```
template <class T>
class A
{
private:
int a_;

public:
// 非模板友元函数，利用模板类参数自动生成的友元函数--非模板函数
friend void show(const A<T> &a);
// 同时定义如下函数会导致重定义问题
// friend void show(const A<int> &a);
};
void show(const A<int> &a) {} // 需指定特定的类型——因为传入的对象类型早已被确定
```

约束模板友元函数

最好的模板友元函数

1. 支持多个模板类
2. 可以具体化

```
template <typename T>
void show(T &a); // 友元函数模板的声明

template <class T1, class T2>
class AA
{
T1 m_x;
T2 m_y;

public:
friend void show<>(AA<T1, T2> &a); // 模板类中，再次声明友元函数模板。
};

template <typename T> // 友元函数模板的定义
void show(T &a) {}

template <> // 具体化版本
void show(AA<int, string> &a) {}
```

非约束模板友元

实例化多个友元函数——即便是跟自己没关系的友元函数，每个实例化的类都拥有n个友元函数

```
template <class T1>
class AA
{
private:
T1 m_x;

public:
template <typename T>
friend void show(const T &o);
};
template <typename T>
void show(const T &o) {}
```

成员模板

```
template <class T>
class A
{
public:
template <class T, class T1>
void show(T1 a) {} // 可以相同，可以不同，也可以重载

// 类外实现
template <class T2>
void show(T2 b);
};

template <class T>
template <class T2>
void A<T>::show(T2 b) {}
```

模板类用作参数--模板的模板

主要用于数据结构设计中，函数模板不支持模板的模板

```
template <template <class, int> class tableType, class dataType, int len>
class LinearList
{
private:
tableType<dataType, len> table_;
};

template <class T, int len>
class Array {};
int main()
{
LinearList<Array, int, 10> a;
}
```

编译、链接、命名空间

预处理

include

无情的复制粘贴机器，包含txt都行

- <>从编译器自带库查找，""自定义目录

define

预处理时做字符串替换

- 有参数的宏，无参数的宏
- 没有内容的宏

undef，# ifdef，# ifndef

```
#ifdef _WIN32
cout << "windows系统" << endl;
typedef long long int64;
#else
cout << "非winddows系统" << endl;
typedef long int64;
sakdjoasjdo // 不会报错，因为代码不被启用
#endif
```

防止重复包含

```
#pragma once //有些编译器不支持
```

```
#ifndef _ok_ // 不重名即可
#define _ok_
// xxx
#endif
```

一些工作链上的问题

.cpp--预处理→.i--编译器→.s--汇编器→.o--链接→.out或.exe

1. 分开编译，最后链接的**效率高**
2. **找不到标识符**：未声明
3. **声明**写在头文件中--并**不会编译**
4. **无法解析的外部命令**：未定义
5. **重定义问题**
6. .h中写源代码，被多次包含，最后链接在一起
7. # include "*.cpp"
8. 全局变量：在**头文件**中声明并加**extern**关键字，定义在相应文件中即可
9. const仅对单个文件有效，故须定义在头文件中
10. 模板分文件编写

命名空间

```
namespace mySpace
{
class A {};
int ok = 1;
}

int main()
{
int b = mySpace::ok // 无任何冲突

using mySpace::A;
A a;
using namespace mySpace;
// 作用域问题，局部>全局；重定义问题；多个空间同时存在的不明确调用问题

namespace a = mySpace; //别名
}
```

分文件编写；同一个命名空间可以分散在不同的文件中；命名空间可以嵌套

```
// .h
namespace mySpace
{
class A {};
extern int ok;
extern int c;
}
// 1.cpp
namespace mySpace
{
int ok = 1;
}
// 2.cpp
namespace mySpace
{
int c = 1;
}
```

在命名空间中声明全局变量，而不是使用外部全局变量和静态变量

- 只声明不定义和静态变量的效果是一样的

using 优先在局部使用，不要在头文件使用——与设计理念不符

匿名命名空间

类型转换

认为C风格的转换太松散：(int)a，int(a)；本质没什么不同，都只是语法上的解释

- 隐式转换：int → long 安全，long → int可能丢失数据，报警，需要强转int(var)

总之，进行更严格的语法检查

static_cast

普通需求

```
void func(void *ptr) { double *p = static_cast<double *>(ptr); }
int main()
{
int a = 1;
long b = static_cast<long>(a); // 与c没什么不同

// 转换指针?
void *ptr = &a;
double *p = static_cast<double *>(ptr);
// 同一个函数内转换指针类型的场景几乎不存在，主要用于函数传递参数:
func(ptr);
}
```

reinterpret_cast

专门为转换指针而设计，要求原类型和目标类型至少有一个为指针or引用

```
void func(void *ptr) { int a = reinterpret_cast<int>(ptr); }
int main()
{
// case 1：普通转换
int a = 1, b2 = 1;
double *o = reinterpret_cast<double *>(&a);
// case 2：传值给一个只有指针参数的函数
func(reinterpret_cast<void *>(a)); // 大小不一样的时候会出现警告，改为long吧
}
```

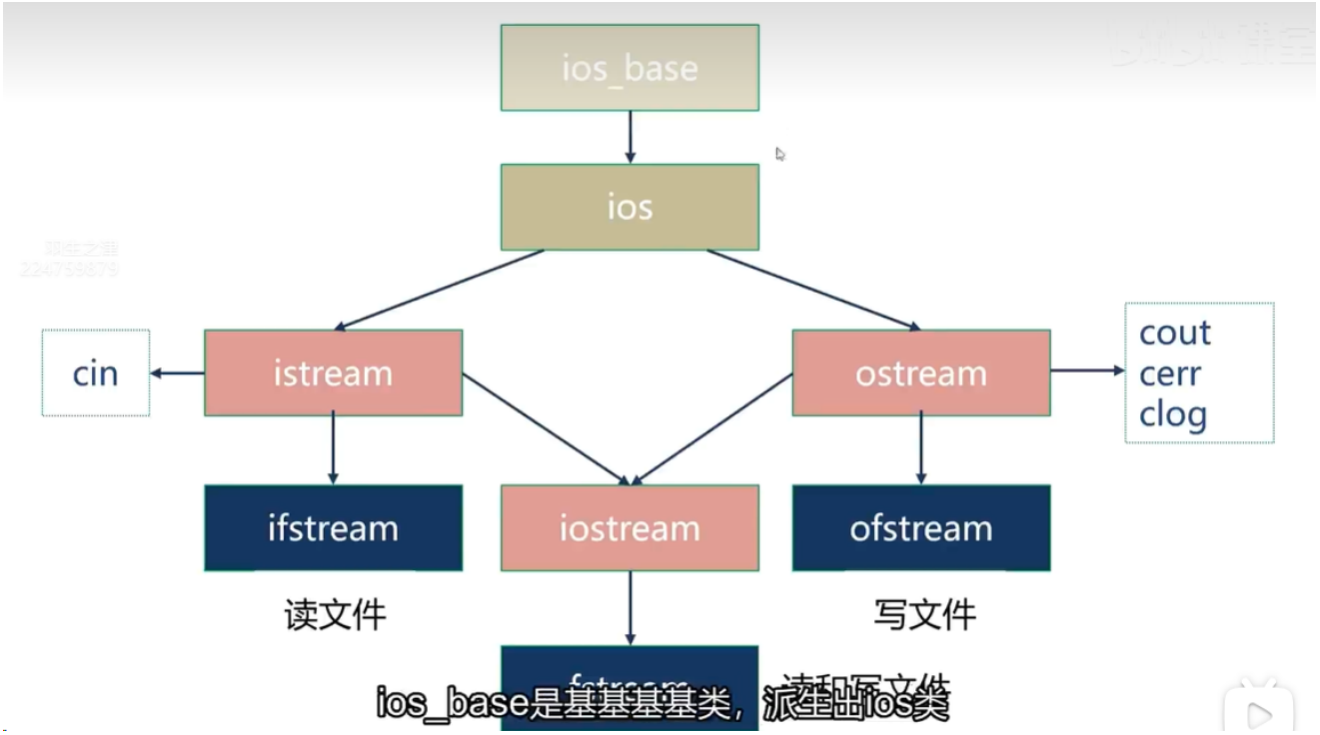
const_cast

丢掉不能隐式转换的指针const

```
int a = 1;
const int *ptr = &a;
int *ok = const_cast<int *>(ptr);
```

dynamic_cast

文件操作



写入文本文件 ofstream

```
#include <iostream>
#include <fstream> //ofstream所需
using namespace std;
int main()
{
ofstream fout;
string filename = "text.txt";

fout.open(filename); // C风格和string类都支持；创建在程序运行目录，也可以指定路径
if (!fout.is_open()) // 判断是否失败， 失败的原因主要有：1）目录不存在；2）文件不存在；3）没有权限，Linux 平台下很常见
cout << "打开文件" << filename << "失败\n";

fout << "哦\n";
fout << "哈哈，C++语法真复杂\n";
fout << "哈哈，C++语法真复杂\n";

fout.close(); // 需要关闭

return 0;
}
```

打开参数

ofstream打开的文件，不存在时进行创建；若存在，需指定打开方式

```
fout.open("text.txt"); // 缺省值，覆盖
fout.open("text.txt", ios::out); // 缺省值，覆盖
fout.open("text.txt", ios::trunc); // 覆盖
fout.open("text.txt", ios::app); // 在内容末尾追加
```

打开方式

```
string file = R"(E:\desktop\today\text.txt)";
ofstream fout(file);//ok，可加参数
fout.open(file); //ok
fout(file); //不行
```

文件路径写法

一般用全路径

1. "D:\data\txt\test.txt" // 错误。
2. R"(D:\data\txt\test.txt)" // 原始字面量，C++11 标准。
3. "D:\data\txt\test.txt" // 转义字符。
4. "D:/tata/txt/test.txt" // 把斜线反着写。95
5. "/data/txt/test.txt" // Linux 系统采用的方法。

读取文本文件 ifstream

```
ifstream fin;
fin.open(file, ios::in); // 缺省值

string buffer;
// <string>; 按行读取全部内容，每行覆盖到buffer中，读取完毕返回空
while (getline(fin, buffer))
{
cout << buffer;
}

//第二种方法，char数组，fin.getline(buffer, 15) -- 不能自动扩展，无法预估文件每行字节数
//第三种，ifstream会读取数据，直到遇到空白字符；操作失败时，会被隐式转化为false
string buffer;
while (fin >> buffer)
{
cout << buffer << endl;
}
```

写入二进制文件

```
#include <iostream>
#include <fstream> //ofstream所需
using namespace std;
int main()
{
string file = R"(E:\desktop\today\test.dat)";
ofstream fout(file, ios::app | ios::binary); // 同时按这两种方式打开

struct Person
{
int age;
char name[31];
} me;

fout.write((const char *)&me, sizeof(me)); // 应该是void*定义的函数，但是C++没这么做，需要强转
fout.close();

return 0;
}
```

一些细节

文件与换行

1. 在 windows 平台下，文本文件的换行标志是"\r\n"。在 linux 平台下，文本文件的换行标志是"\n"
2. windows下，以文本文件形式打开文件，系统会把\r\n转换为\n；写入时，会把\n转换为\r\n。以二进制方式打开文件，写和读都不会进行转换
3. linux下，以文本或二进制方式打开文件，系统不会做任何转换
4. 以文本方式读取文件的时候，遇到换行符停止，读入的内容中没有换行符；以二制方式读取文件 的时候，遇到换行符不会停止，读入的内容中会包含换行符（换行符被视为数据）----故读取二进制文件须指明size
5. 实际开发中，一般：
6. 以文本模式打开文本文件，用行的方法操作它
7. 以二进制模式打开二进制文件，用数据块的方法操作它

8. 以二进制模式打开文本文件和二进制文件，用数据块的方法操作它——这种情况表示不关心数据的内容。（例如复制文件和传输文件）
9. 不要以文本模式打开二进制文件，也不要用行的方法操作二进制文件，可能会破坏二进制数据文件的格式，也没有必要。

二进制文件

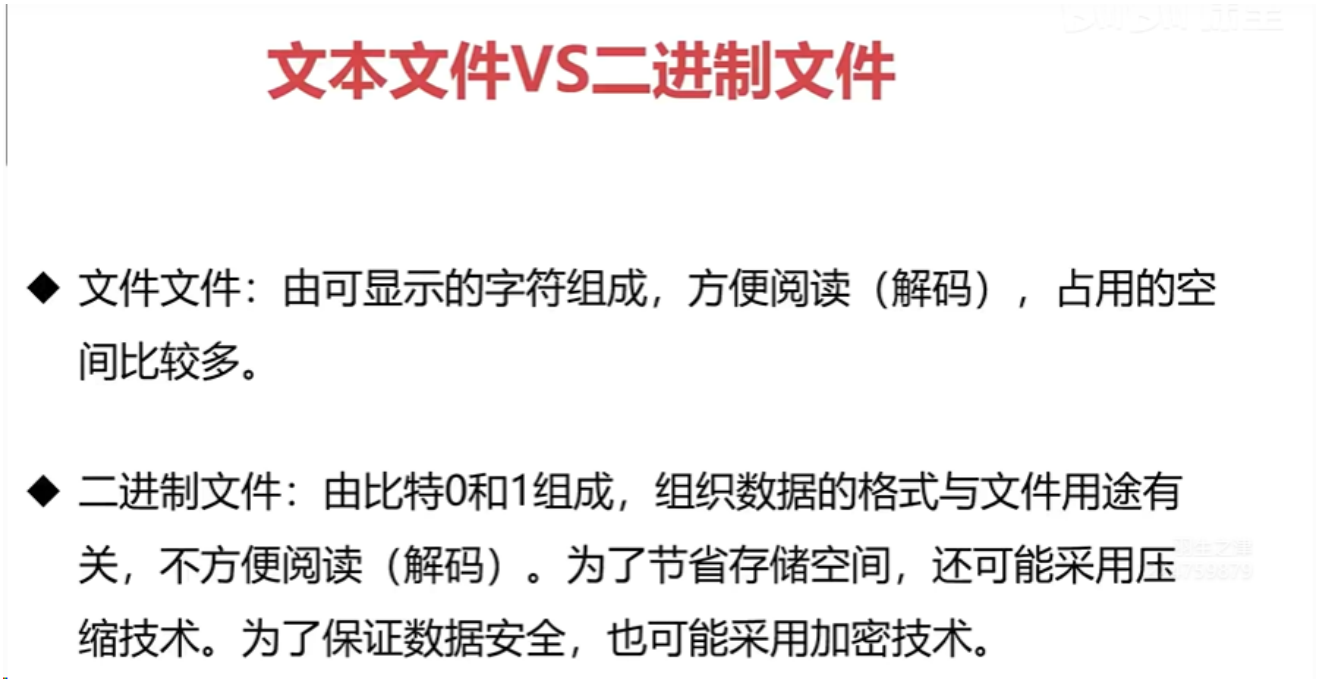
写：从内存写入磁盘

读：从磁盘读到内存

'1'	'3'	'1'	'4'	'2'	'5'
49	51	49	52	50	53
00110001	00110011	00110001	00110100	00110010	00110101

1. 上面时文本文件的表现形式，下面是计算机中实际存储的二进制文件，中间是ASCII编码
2. 二进制文件以数据类型的形式组织，应作为整体考虑，单个字节往往没有意义
3. 二进制文件后缀没有意义，太多了...内容全都是一串01，看你怎么解码

- 需要正确的解码方式才能显示正确的文件内容



读取二进制文件

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
string file = R"(E:\desktop\today\test.dat)";
ifstream fin(file, ios::in | ios::binary);
if (!fin.is_open())
return -1;

struct Person
{
int age;
char name[31];
} me;

while (fin.read((char *)&me, sizeof(me)))
{
cout << me.name;
}

return 0;
}
```

fstream类

可替代ofstream/ifstream进行操作，语法完全一致

- 注意fstream **类的缺省模式是 ios::in | ios::out**，如果文件不存在，则创建文件；但是，不会清空文件原有的内容。**最好写明**ios::in或ios::out
- **实际开发中**：如果只想写入数据，用 ofstream；如果只想读取数据，用 ifstream；如果想写和读数据，用 fstream，这种情况不多见。**不同的类体现不同的语义**
- 在 **Linux** 平台下，文件的写和读有严格的权限控制。（需要的权限越少越好）

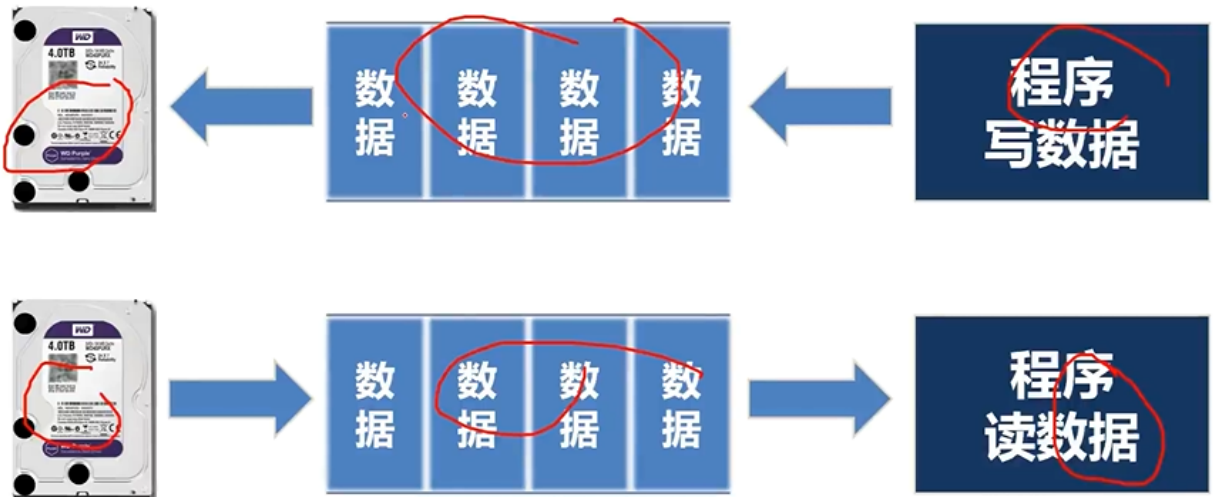
随机存取

- 有一个**文件位置指针**，指向读or写的位置，会**自动移动**，通过**tellp()**或**tellg()**成员函数获取文件指针，其位置**以字节为单位**
- 两个函数名字全称为tell put pointer和tell get pointer
- 注意，无论是tellp还是tellg，其实是**同一个**文件指针
- **移动位置指针 seekg() seekp()**

```
std::istream &seekg(std::streampos _Pos);
fin.seekg(128); // 把文件指针移到第 128 字节。
fin.seekp(128); // 把文件指针移到第 128 字节。
fin.seekg(ios::beg); // 把文件指针移动文件的开始。
fin.seekp(ios::end); // 把文件指针移动文件的结尾。

enum seek_dir { beg, cur, end};
fin.seekg(30, ios::beg); // 从文件开始的位置往后移 30 字节。
fin.seekg(-5, ios::cur); // 从当前位置往前移 5 字节。
fin.seekg(8, ios::cur); // 从当前位置往后移 8 字节。
fin.seekg(-10, ios::end); // 从文件结尾的位置往前移 10 字节。
```

文件缓冲区

文件缓冲区



1. 缓冲区的设计是为了**减少磁盘读写的次数**，提高速度
2. 每个流都有**独立的内存缓冲区**
3. 程序的输出流在默认状态下，并不会立即写入，而是在特定条件下后写入，有时需要**手动刷新缓冲区**，方法为：

```
fout.flush(); //手动写入

fout << "输出的内容" << endl; //换行并刷新

fout << unitbuff << "输出的内容" << nounitbuff; //关闭缓冲区机制，写入，再开启缓冲区机制
```

不立即刷新可能存在*风险*

流状态

流有三个状态：**eofbit、badbit 和 failbit**，**取值{0，1}**，全0一切正常，**good()**函数返回true

1. **eof()**: 到达末尾变1
2. **bad()**: 无法判断的错误变1，一般为：写入输入流、磁盘无空间等致命错误
3. **fail()**: 未能读取到预期字符时变1，一般为软件错误，可晚会，如：想读取一个整数，但内容是一个字符串；文件到了末尾；I/O失败，文件末尾等
4. clear()、setstate()？？

异常

C++11已弃用异常——但还是要学的

- 关键词：**异常使用方法，异常终止**

```
try
{
// 可能抛出异常的代码。

// throw 异常对象
// 可以throw：常量、变量、用户定义的类
// throw后直接跳转到catch，不会继续执行接下来的代码
// 接下来的代码
}
catch (...) // 可有多个catch匹配
{
// 不管什么异常，都在这里统一处理。
}

// catch后继续向下执行
// 但如果有异常没catch，程序就会异常中止
```

noexcept

C++98 标准提出了异常规范，目的是为了让使用者**知道函数可能会引发哪些异常**

```
void func1() throw(A, B, C); // 表示该函数可能会抛出 A、B、C 类型的异常。
void func2() throw(); // 表示该函数不会抛出异常。
void func3(); // 该函数不符合 C++98 的异常规范。
```

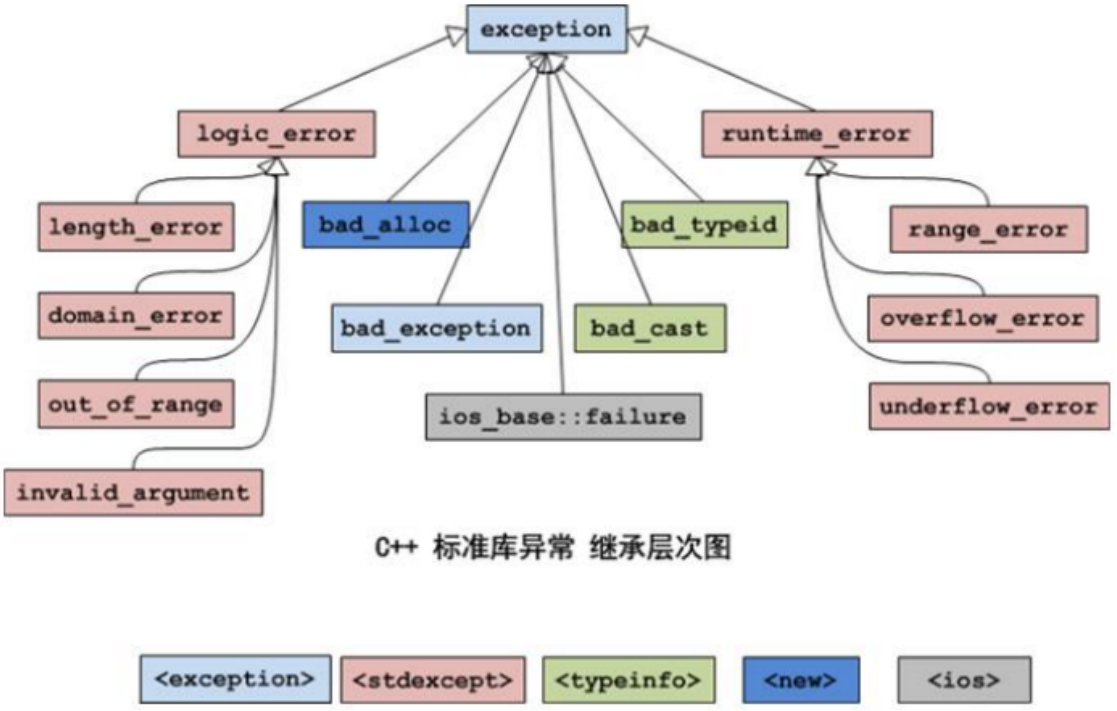
C++11弃用异常规范

```
void func4() noexcept; // 该函数不会抛出异常。
```

关键字 noexcept 也可以用作运算符，判断表达试（操作数）是否可能引发异常；如果表达式可能 引发异常，则返回 false，否则返回 true

C++ 标准库异常

标准库很可能会抛出异常



C++ 标准库异常 继承层次图

std::bad_alloc

1. 分配内存失败
2. 通过以下方法避免抛出异常

```
double* ptr = new (std::nothrow) double[100000000000];
if (ptr == nullptr) cout << "ptr is null.\n";
```

std::bad_cast

dynamic_cast 可以用于引用，但是，C++没有与空指针对应的引用值，如果转换请求不正确，会 出现 std::bad_cast 异常

std::bad_typeid

假设有表达式 typeid(*ptr)，当 ptr 是空指针时，如果 ptr 是多态的类型，将引发 std::bad_typeid 异常。

std::logic_error

1. std::out_of_range
2. std::length_error

呃，没有建设性的学习，等用到了再说吧

断言

或<assert.h>

1. 断言失败时，调用abort()终止程序
2. 可读性提升
3. 不建议在断言中执行表达式
4. 断言应该处理程序中不应出现的错误，而不是逻辑错误，一般放在函数第一行，用于检查形参错误

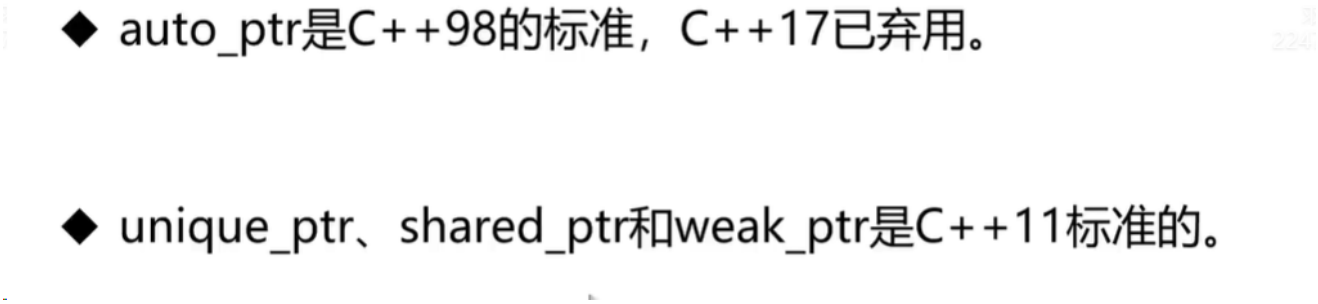
静态断言

无需头文件，编译时检查

```
static_assert(constant,提示信息);
```

智能指针

1. 堆区资源需要手动new和delete...容易内存泄漏
2. 先创建普通指针，然后把普通指针交给智能指针对象，当对象生命周期结束时，会自动销毁



unique_ptr

同时只有一个unique_ptr指向一个对象

```
template <typename T, typename D = default_delete<T>>
class unique_ptr
{
public:
explicit unique_ptr(pointer p) noexcept; // 不可用于转换函数
~unique_ptr() noexcept; // 有delete

// 重载* ->，可以像普通指针一样使用智能指针
T& operator*() const;
T* operator->() const noexcept;

// 禁用拷贝构造、默认赋值——防止程序员犯错；编译器做了特别的处理，只要是没多个智能指针管理一个内存对象就行
unique_ptr(const unique_ptr &) = delete;
unique_ptr& operator=(const unique_ptr &) = delete;

// 右值引用
unique_ptr(unique_ptr &&) noexcept;
unique_ptr& operator=(unique_ptr &&) noexcept;
// ...
private:
pointer ptr; // 内置的指针。
};
```

初始化

——本身创建在栈上

```
unique_ptr<AA> p0(new AA("ok"));

AA* p = new AA("ok"); // 不推荐
unique_ptr<AA> p0(p);

unique_ptr<AA> p0 = make_unique<AA>("ok"); // C++14 标准。
unique_ptr<int> pp1 = make_unique<int>(); // 数据类型为 int。
unique_ptr<AA> pp2 = make_unique<AA>(); // 数据类型为 AA，默认构造函数。
unique_ptr<AA> pp3 = make_unique<AA>("ok"); // 数据类型为 AA，一个参数的构造函数。
unique_ptr<AA> pp4 = make_unique<AA>("ok", 8); // 数据类型为 AA，两个参数的构造函数。
```

使用方法

1. 正常使用，**不支持拷贝、赋值，不要用同一个裸指针初始化多个智能指针**
2. **get()**返回裸指针
3. 智能指针**只能管理new**分配的内存
4. 作为参数**只能引用，不能传值**
5. 不支持指针运算

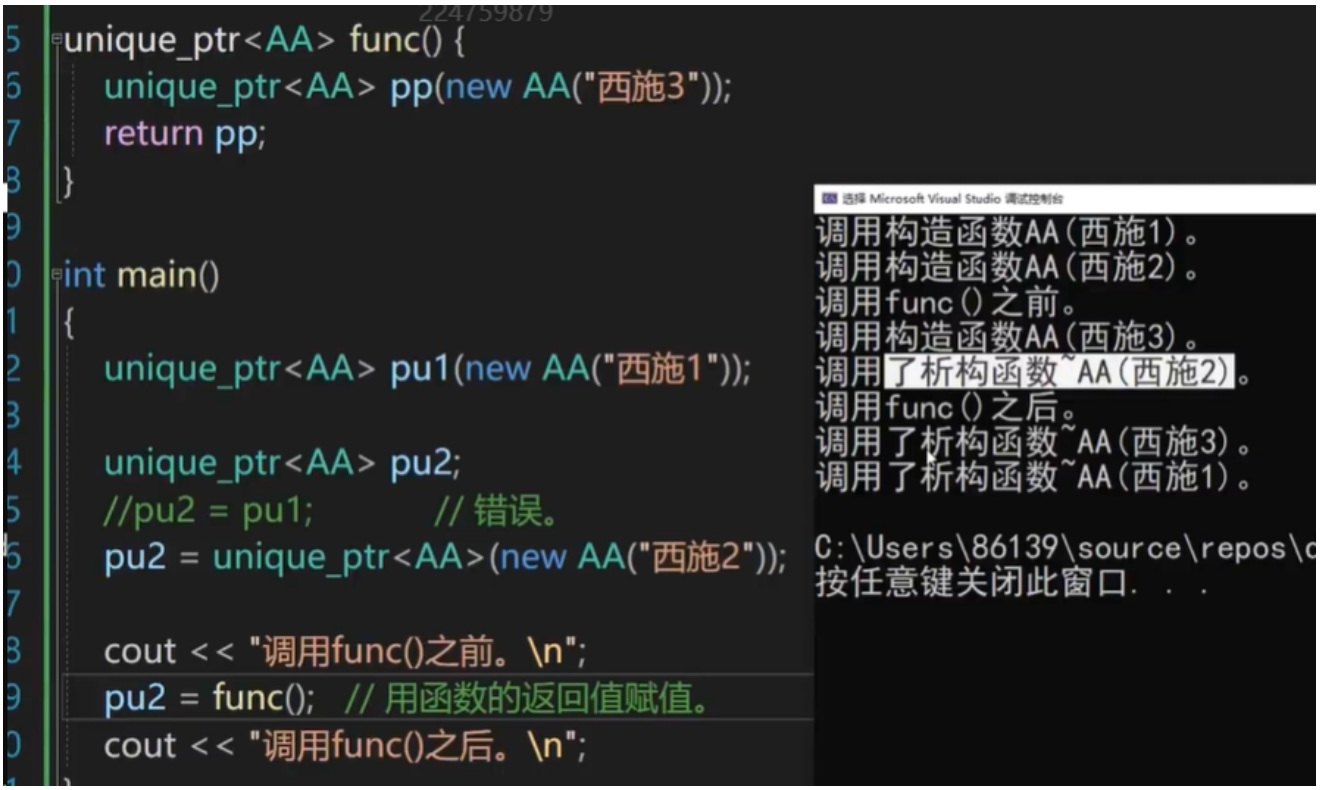
细节

1

——编译器的特别处理

1. 将一个 unique_ptr 赋给另一个时，如果源 unique_ptr 是一个**临时右值，编译器允许这样做**
2. 如果源 unique_ptr 将存在一段时间，**编译器禁止这样做**。
3. 一般用于函数的返回值。

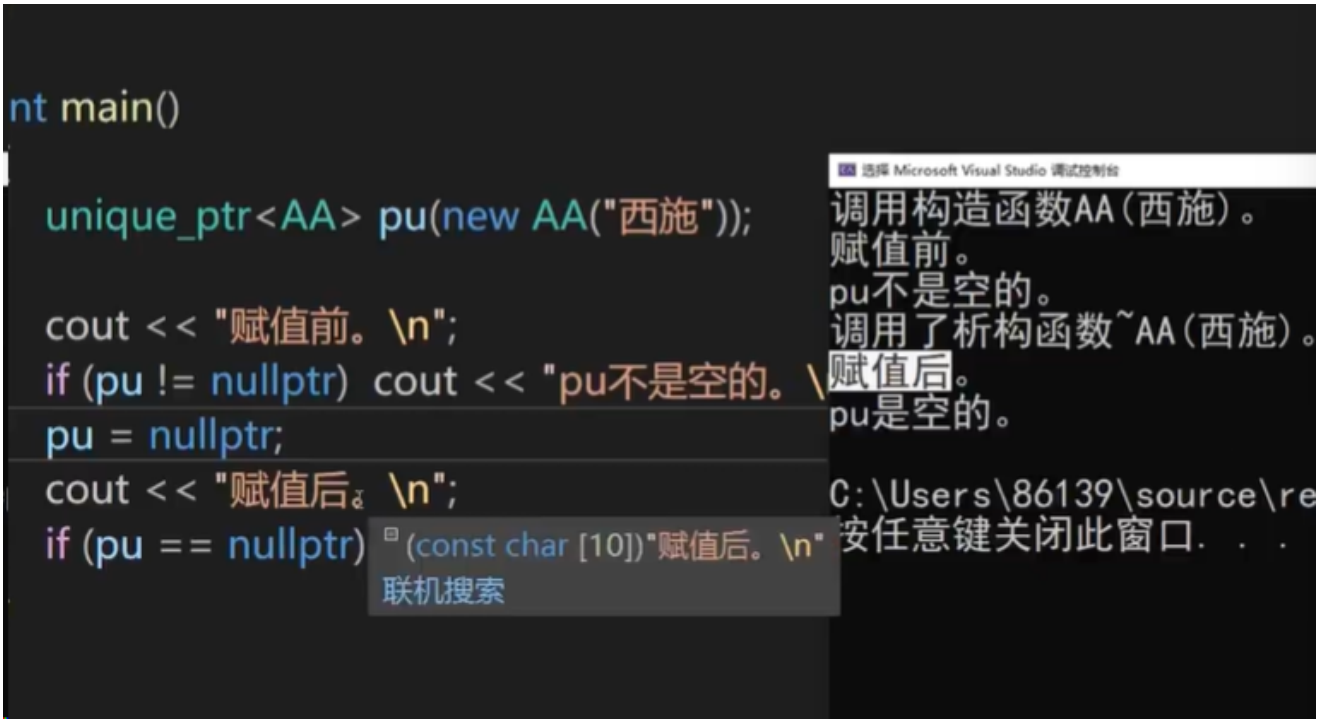
即允许：



1. 禁用=，居然可以=？--特别处理
2. pu2可以二次赋值？--先释放原来的，再接手新的
3. 如果将3创建为全局变量，则会报错

2

用空指针给智能指针赋值将释放原对象



3

1. release可以释放原指针的控制权，返回原指针，释放后原智能指针自动被置空

2. move可以转移控制权

```
func1(pu.get()); // 函数 func1()需要一个指针，但不对这个指针负责。
func2(pu.release()); // 函数 func2()需要一个指针，并且会对这个指针负责。
func3(pu); // 函数 func3()需要一个 unique_ptr，不会对这个 unique_ptr负责。
func4(move(pu)); // 函数 func4()需要一个 unique_ptr，并且会对这个 unique_ptr负责。
```

4 reset

```
void reset(T *_ptr = (T *)nullptr);

pp.reset(); // 释放 pp 对象指向的资源对象。
pp.reset(nullptr); // 释放 pp 对象指向的资源对象
pp.reset(new AA("bbb")); // 释放 pp 指向的资源对象，同时指向新的对象。
```

5 swap

```
void swap(unique_ptr<T> &_Right);
```

6 多态

7 不安全的智能指针

exit退出时，无法释放局部的智能指针，可以释放全局的

也可以造成空指针和野指针的问题——空智能指针

8 数组智能指针

```
unique_ptr<AA[]> parr2(new AA[3]{string("西施"), string("冰冰"), string("幂幂")});
```

shared_ptr

共享对象，内部采用计数机制

1. 新shared_ptr关联时，计数+1
2. 超出作用域-1，为0时，自动释放对象

初始化

构造函数为explicit，有拷贝构造和赋值函数

```
shared_ptr<AA> p0(new AA("西施"));

shared_ptr<AA> p0 = make_shared<AA>("西施"); // C++11 标准，效率更高。
shared_ptr<int> pp1 = make_shared<int>(); // 数据类型为 int。
shared_ptr<AA> pp2 = make_shared<AA>(); // 数据类型为 AA，默认构造函数。
shared_ptr<AA> pp3 = make_shared<AA>("西施"); // 数据类型为 AA，一个参数的构造函数。
shared_ptr<AA> pp4 = make_shared<AA>("西施", 8); // 数据类型为 AA，两个参数的构造函数。

AA *p = new AA("西施");
shared_ptr<AA> p0(p); // 用已存在的地址初始化。

shared_ptr<AA> p0(new AA("西施"));
shared_ptr<AA> p1(p0); // 用已存在的 shared_ptr 初始化，计数加 1。
shared_ptr<AA> p1 = p0; // 用已存在的 shared_ptr 初始化，计数加 1。
```

使用方法

重载了* 和→，可以正常使用

1. unique()，计数为1返回真，否则假
2. use_count()，返回计数
3. 一个shared_ptr作左值，原计数-1，作右值，原计数+1
4. get()，裸指针
5. 不要用同一个裸指针初始化多个shared_ptr，必须管理new分配的内存
6. 作为参数**只能引用，不能传值**
7. 不支持指针运算

细节

1. 没有release函数——释不释放资源不是一个人说了算
2. 可以以用move转移控制权，还可以将unique_ptr转移给shared_ptr，反过来不行
3. reset，计数-1；swap
4. exit导致的不安全；多态；
5. 指向数组——[]返回的是引用，可以作为左值
6. 线程安全性
7. 能有unique就用unique，效率高占用资源少

智能指针删除器

1. 默认使用delete，可以自定义行为
2. 删除器可以是全局函数、仿函数和 Lambda 表达式，形参为原始指针

```
void deletefunc(AA* a) { // 删除器，普通函数。
cout << "自定义删除器（全局函数）。\n";
delete a;
}
shared_ptr<AA> pa1(new AA("西施 a"), deletefunc);
// unique_ptr<AA, decltype(deleterlamb)> pu3(new AA("西施 3"), deleterlamb);
```

weak_ptr

1. shared_ptr如果出现循环引用，将无法释放资源——weak_ptr解决了这一问题
2. weak_ptr绑定到shared_ptr时，不会改变其计数，不控制生命周期，更像是shared_ptr的助手，仅仅为了防止循环引用的情况出现罢了
3. 它仅仅知道对象是否还活着，可以用lock()升级为shared_ptr，这是线程安全的

```
class BB;
class AA
{
public:
string m_name;
AA() { cout << m_name << "调用构造函数 AA()。\n"; }
AA(const string & name) : m_name(name) { cout << "调用构造函数 AA("<< m_name
<<")。\n"; }
~AA() { cout << "调用了析构函数~AA(" << m_name << ")。\n"; }
weak_ptr<BB> m_p;
};
class BB
{
public:
string m_name;
BB() { cout << m_name << "调用构造函数 BB()。\n"; }
BB(const string& name) : m_name(name) { cout << "调用构造函数 BB(" << m_name
<<")。\n"; }
~BB() { cout << "调用了析构函数~BB(" << m_name << ")。\n"; }
weak_ptr<AA> m_p;
};
```

使用方法

1. 没有重载→和*，不能直接访问资源
2. operator=(); expired(); lock(); reset()； swap();
3. weak_ptr提升的行为是线程安全的，可以知道对象是否过期

```
shared_ptr<BB> pp = pa->m_p.lock(); // 把 weak_ptr 提升为 shared_ptr。
if (pp == nullptr)
cout << "语句块外部：pa->m_p 已过期。\n";
else
cout << "语句块外部：pp->m_name=" << pp->m_name << endl;
```

在多线程中，if内进行pa是否过期的判断是线程不安全的，建议采用上述写法

C++11 标准

前文已覆盖：

1. long long
2. 原始字面量
3. auto
4. decltype
5. 可以作为常规函数的参数
6. nullptr
7. 智能指针
8. 弃用异常
9. explicit
10. 类内成员初始化
11. 基于范围的for循环
12. 嵌套模板的尖括号
13. 静态断言

char16_t 和 char32_t 类型

新增了类型 char16_t 和 char32_t，以支持 16 位和 32 位的字符

统一初始化列表

<initializer_list>

1. 丰富了 {} 的适用范围，可以用于所有内置类型和用户自定义类型
2. STL 容器也提供了相关的的构造函数
3. 可以作为函数的参数or初始化函数的参数

模板别名

对于冗长或复杂的标识符，如果能够创建其别名将很方便。以前，C++为此提供了 typedef：

```
typedef std::vector<std::string>::iterator itType;
```

C++11 提供了另一种创建别名的语法，这在第 14 章讨论过：

```
using itType = std::vector<std::string>::iterator;
```

差别在于，新语法也可用于模板部分具体化，但 typedef 不能：

```
template<typename T>
  using arr12 = std::array<T,12>;  // template for multiple aliases
```

上述语句具体化模板 array<T, int>（将参数 int 设置为 12）。例如，对于下述声明：

```
std::array<double, 12> a1;
std::array<std::string, 12> a2;
```

可将它们替换为如下声明：

```
arr12<double> a1;
arr12(std::string> a2;
```

强类型枚举

传统的 C++枚举提供了一种创建常量的方式，但类型检查比较低级。还有，如果在同一作用域内定义的两个枚举，它们的成员不能同名。

针对枚举的缺陷，C++11 标准引入了枚举类，又称强类型枚举。

声明强类型枚举非常简单，只需要在 enum 后加上关键字 class。

例如：

```
enum e1{ red, green };
enum class e2 { red, green, blue };
enum class e3 { red, green, blue, yellow };
```

使用强类型枚举时，要在枚举成员名前面加枚举名和::，以免发生名称冲突，如：e2::red，e3::blue

强类型枚举默认的类型为 int，也可以显式地指定类型，具体做法是在枚举名后面加上:type，type 可以是除 wchar_t 以外的任何整型。

例如:

```
enum class e2:char { red, green, blue };
```

新STL容器

1）array（静态数组）

array 的大小是固定的，不像其它的模板类，但 array 有 begin()和 end()成员函数，程序员可以 array 对象使用 STL 算法。

2）forward_list（单向链表）

3）unordered_map、unordered_multimap、unordered_set、unordered_multiset（哈希表）

新STL算法

1. 新增方法cbegin()、cend()、crbegin()、crend()，用于自动推导
2. emplace
3. 移动构造函数和移动赋值函数

弃用export

但作为关键字保留

final

放在类/函数后，表示不能被继承/虚函数重写

```
class BB : public AA
{
public:
void test() final // 如果有其它类继承 BB，test()方法将不允许重写。
{
cout << "BB class...";
}
};
```

override

放在函数名后用于提高可读性，不是重写会报错

```
class BB : public AA
{
public:
void test() override
{
cout << "BB class...";
}
};
```

数值类型和字符串之间的转换

传统方法

1. sprintf()和 snprintf()函数把数值转换为 char*字符串
2. 用 atoi()、atol()、atof()把 char *字符串转换为数值。

C++11

中的to_string()，有多个重载，把任意类型转换为string

其他to string

```
int stoi(const string &str, size_t *pos = nullptr, int base = 10);
long stol(const string &str, size_t *pos = nullptr, int base = 10);
long long stoll(const string &str, size_t *pos = nullptr, int base = 10);
unsigned long stoul(const string &str, size_t *pos = nullptr, int base = 10);
unsigned long long stoull(const string &str, size_t *pos = nullptr, int base = 10);
float stof(const string &str, size_t *pos = nullptr);
double stod(const string &str, size_t *pos = nullptr);
long double stold(const string &str, size_t *pos = nullptr);
```

constexpr

const有双重语义：只读变量，常量

```
void func(const int len1) // len1 是只读变量，不是常量。
{
int array1[len1]={0}; // VS 会报错，Linux 平台的数组长度支持变量，不会报错。
const int len2 = 8;
int array2[len2]={0}; // 正确，len2 是常量。
}
```

C++11中，为const保留“只读变量”语义，constexpr语义划分为常量

默认函数控制 =default，=delete

前者启用默认函数，后者禁用默认函数

委托构造

1. 注意，不要互相委托形成死递归
2. 委托构造不能初始化其他成员变量

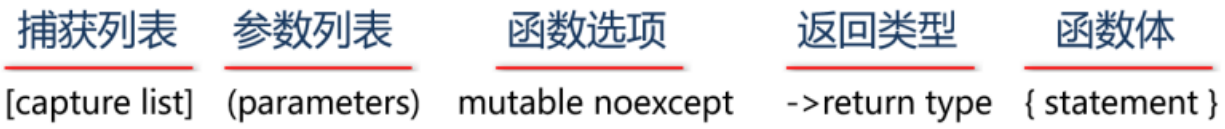
```
class AA
{
private:
int m_a;
int m_b;
double m_c;

public:
AA(double c) : m_c(c + 3){}

AA(int a, int b) : m_a(a + 1), m_b(b + 2){}

// 构造函数委托 AA(int a, int b)初始化 m_a 和 m_bAA(int a, int b, const string &str) :
AA(a, b)
{
cout << "m_a=" << m_a << ",m_b=" << m_b << ",str=" << str << endl;
}
};
```

继承构造

```
class BB : public AA // 派生类。
{
public:
double m_c;
using AA::AA; // 使用基类的构造函数。

BB(int a, int b, double c) : AA(a, b), m_c(c) // 不是委托构造
{
cout << " BB(int a, int b, double c)" << endl;
}
};
```

lambda 函数

——不需要提前准备函数or仿函数，临时写就行



1. 不支持默认参数、可变参数，因为没有任何意义，必须有参数名；参数列表没有可以不写
2. 后置返回类型，不写的话编译器会自动推断，建议手动写
3. 通过捕获列表，函数**可以直接使用父作用域的非静态局部变量——本质就是传递参数**

- 静态变量和全局变量可以直接访问，无须、也不能捕获
- =和&都是**自动判断**需要捕获的变量，一个是**值捕获**，一个是**引用捕获**，"隐式捕获"
- 二者可以混合捕获，此时必须以不同的方式捕获(一引用，一值)

lambda捕获方式	
[ ]	空捕获列表。lambda不能使用所在函数中的变量。
[names]	names是一个逗号分隔的名字列表，这些名字都是lambda所在函数的局部变量。默认情况下是值捕获，名字前加&指明是引用捕获。
[=]	隐式捕获列表，采用值捕获方式。lambda体将拷贝所使用的来自所在函数的实体都值。
[&]	隐式捕获列表，采用引用捕获方式。lambda体中所使用的来自所在函数的实体都采用引用方式使用。
[&, identifier_list]	identifier_list是一个逗号分隔的列表，包含0个或多个来自所在函数的变量。这些变量采用值捕获方式，而任何隐式捕获的变量都采用引用捕获。identifier_list中的名字前面不能使用&
[=, identifier_list]	identifier_list中的变量都采用引用方式捕获，而任何隐式捕获的变量都采用值捕获。identifier_list中的名字不能包括this，且这些名字之前必须使用&

本质

1. 编译器将lambda函数翻译为一个类，重载了()，也即一个**functor**
   - 直接创建一个函数对象
2. 值捕获时，编译器将**捕获到的值作为参数初始化自己的成员变量**
   - 也因此，值捕获是**创建lambda函数时拷贝**，而非调用时拷贝
   - 且**不影响原变量**的值
3. 重载()时，**默认重载为一个常函数**，可以加**mutable**去掉const
4. 引用捕获直接引用即可，但必须保证在执行lambda时，原对象存在

```
int main()
{
static int a = 3;
int ii = 1;
double dd = 8.3;
auto f = [ii, dd](const int &no) mutable noexcept -> double
{
a = 34;
cout << "hello" << no << endl;
cout << a << endl;
return 1.1;
};
cout << a << endl;
f(1);
cout << a << endl;

return 0;
}
```

右值引用

能取地址的都是左值，否则右值。右值分为纯右值和将亡值。

```
class AA{};
AA getTemp(){return AA();}

int ii = 3; // ii 是左值，3 是右值。
int jj = ii + 8; // jj 是左值，ii+8 是右值。
AA aa = getTemp(); // aa 是左值 ， getTemp()的返回值是右值（临时变量）。若返回引用则为左值
```

右值引用就是给右值取个名字，和普通变量没有区别，此时就变成了左值

```
int b = 8;
int&& c = b + 5; // 右值引用
```

本应结束生命周期的右值仍具有利用价值，通过右值引用获得了新生，生命周期同变量一样

引入右值引用的目的是实现移动语义

左值引用只能绑定左值，右值引用只能绑定右值

- 常量左值引用却是个奇葩，是一个万用的引用类型——可以绑定非常量左值、常量左值、右值，且可像右值引用一样延长生命周期，但是只读不能改
- 本质还是创建了临时对象

移动语义

如果要根据一个 **有堆区资源的 临时对象 进行深拷贝**，会进行不必要的资源**申请、复制、释放**操作操作

——移动语义可以**直接使用临时对象的资源**，节省时间

必须提供**移动构造函数和移动赋值函数**

```
class AA
{
public:
int *m_data = nullptr; // 指向堆区资源的指针。

AA() = default;

void alloc()
{
m_data = new int;
memset(m_data, 0, sizeof(int));
}

AA(const AA &a) // 拷贝构造函数。
{
if (m_data == nullptr)
alloc();
memcpy(m_data, a.m_data, sizeof(int));
}

AA(AA &&a) // 移动构造函数。
{
if (m_data != nullptr) // 如果已分配内存，先释放掉。
delete m_data;

m_data = a.m_data;
a.m_data = nullptr;
}

AA &operator=(const AA &a) // 赋值函数。
{
if (this == &a) // 避免自我赋值。
return *this;
if (m_data == nullptr) // 如果没有分配内存，就分配。
alloc();

memcpy(m_data, a.m_data, sizeof(int));
return *this;
}
```

```
AA &operator=(AA &&a) // 移动赋值函数。
{
if (this == &a)
return *this;
if (m_data != nullptr)
delete m_data;

m_data = a.m_data; // 把资源从源对象中转移过来。
a.m_data = nullptr; // 把源对象中的指针置空。
return *this;
}

~AA()
{
if (m_data != nullptr)
{
delete m_data;
m_data = nullptr;
}
}
};
int main()
{
AA a1;
a1.alloc();
*a1.m_data = 3;

AA a2 = a1; // 将调用拷贝构造函数。

AA a3;
a3 = a1; // 将调用赋值函数。

auto f = []
{ AA aa; aa.alloc(); *aa.m_data = 8; return aa; };
AA a4 = f(); // 调用移动构造函数

AA a6;
a6 = f(); // 返回右值，将调用移动赋值函数
}
```

1. 可以对一些作为局部变量的**左值使用移动构造函数**,它的生命周期也很短，构造完就没用了
- ——使用move把左值转义为右值，转义后原对象不会立刻析构，只有离开自己作用域的时候才会析构，继续使用原对象中的资源会发生意想不到的错误

```
auto a = move(a1); // std::move()，只能move一次
```

2. 没有提供移动构造/赋值函数，编译器便会使用拷贝构造/赋值函数
3. C++11所有容器都实现了移动语义，避免了不必要的拷贝
4. 移动语义只对拥有资源的对象有效，对于基本类型无效

完美转发

中转时如何维持原参数左右值的属性？完美转发：模板T&&，即可接受左值又可接收右值，且提供了模板函数std::forward(参数)用于转发参数

```
void func1(int& ii) {
cout << "参数是左值=" << ii << endl;
}
void func1(int&& ii) {
cout << "参数是右值=" << ii << endl;
}

template<typename TT>
void func(TT&& ii) // 存储了左右值信息；注意int&& jj只能接收右值
{
func1(forward<TT>(ii)); // 读取存储的信息并转发出去
}
int main()
{
int ii = 3;
func(ii); // 实参是左值。
func(8); // 实参是右值。
}
```

可变参数模板

C语言的可变参数很麻烦；即便统一初始化列表也是一种可变参数，但是数据类型必须相同

这是一种特别的语法，编译器做了特别的处理，不是常规语法

```
template <typename T>
void show(T girl) // 向超女表白的函数，参数可能是超女编号，也可能是姓名，所以用 T。
{
cout << "亲爱的" << girl << "，我是一只傻傻鸟。\n";
}

void print() // 递归终止调用的非模板函数，名字必须相同
{
cout << "递归终止。\n";
}
template <typename T, typename... Args>
void print(T arg, Args... args) // 展开参数包的递归函数模板。
{
show(arg); // 把参数用于表白。
// cout << "还有" << sizeof...(args) << "个参数未展开。" << endl;
print(args...); // 继续展开参数。
}

template <typename... Args>
void func(const string &str, Args... args) // 除了可变参数，还可以有其它常规参数。
{
cout << str << endl; // 喊句口号。
print(args...);
cout << "ok。\n";
}
int main(void)
{
// print("金莲", 4, "西施");
// print("冰冰", 8, "西施", 3);
func("我是绝世帅歌。", "冰冰", 8, "西施", 3); // 除第一个参数外均为可变参数
}
```

时间操作库

时间长度

duration模板类

```
#include <chrono>
template<class Rep, class Period = std::ratio<1, 1>>
class duration
{
.....
};
// std::chorono 命名空间包含：
using hours = duration<Rep, std::ratio<3600>> // 小时
using minutes = duration<Rep, std::ratio<60>> // 分钟
using seconds = duration<Rep> // 秒
using milliseconds = duration<Rep, std::milli> // 毫秒
using microseconds = duration<Rep, std::micro> // 微秒
using nanoseconds = duration<Rep, std::nano> // 纳秒
```

操作主要包含：

1. **创建时间**
2. **比较**
3. **获取值**

```
void test_duration()
{
chrono::hours t1(1); // 1 小时
chrono::minutes t2(60); // 60 分钟
chrono::seconds t3(60 * 60); // 60*60 秒
chrono::milliseconds t4(60 * 60 * 1000); // 60*60*1000 毫秒
// chrono::microseconds t5(60 * 60 * 1000 * 1000); // 警告：整数溢出。
// chrono::nanoseconds t6(60 * 60 * 1000 * 1000 * 1000); // 警告：整数溢出。

// 重载了==等运算符，以下表达式为真
if (t1 == t2)
cout << "t1==t2\n";
if (t1 == t3)
cout << "t1==t3\n";
if (t1 == t4)
cout << "t1==t4\n";

// 获取值
cout << "t1=" << t1.count() << endl;
cout << "t2=" << t2.count() << endl;
cout << "t3=" << t3.count() << endl;
cout << "t4=" << t4.count() << endl;
}
```

系统时间

三个静态函数：

```
// 返回当前时间的时间点。
static std::chrono::time_point<std::chrono::system_clock> now() noexcept;

// 将时间点 time_point 类型转换为 std::time_t 类型。
static std::time_t to_time_t( const time_point& t ) noexcept;

// 将 std::time_t 类型转换为时间点 time_point 类型。
static std::chrono::system_clock::time_point from_time_t( std::time_t t ) noexcept;
```

具体使用:

```
#define _CRT_SECURE_NO_WARNINGS // localtime()需要这个宏。
#include <iomanip> // put_time()函数需要包含的头文件。
#include <sstream>
#include <chrono>

void test_system_clock()
{
// 静态成员函数 chrono::system_clock::now() 获取系统时间（C++时间）
auto now = chrono::system_clock::now();

// 静态成员函数 chrono::system_clock::to_time_t()把系统时间转换为 time_t（UTC 时间）
auto t_now = chrono::system_clock::to_time_t(now);
// t_now += 24 * 60 * 60; // 把当前时间加 1 天。
// t_now += -1 * 60 * 60; // 把当前时间减 1 小时。
// t_now += 120; // 把当前时间加 120 秒。

// std::localtime()函数把 time_t 转换成本地时间。（北京时）
// localtime()不是线程安全的，VS 用 localtime_s()代替，Linux 用 localtime_r()代替。
auto tm_now = std::localtime(&t_now);

// 格式化输出时间tm 结构体中的成员。
std::cout << std::put_time(tm_now, "%Y-%m-%d %H:%M:%S") << std::endl; // 2024-03-22 13:02:12
std::cout << std::put_time(tm_now, "%Y-%m-%d") << std::endl; // 2024-03-22
std::cout << std::put_time(tm_now, "%H:%M:%S") << std::endl; // 13:02:12
std::cout << std::put_time(tm_now, "%Y%m%d%H%M%S") << std::endl; // 20240322130212

// 存储到字符串中
stringstream ss;
ss << std::put_time(tm_now, "%Y-%m-%d %H:%M:%S");
string timestr = ss.str();
cout << timestr << endl;

// 反过来?工作中遇上了再说
}
```

计时器

```
void test_chorono::test_steady_clock()
{
// 静态成员函数 chrono::steady_clock::now()获取开始的时间点。
auto start = chrono::steady_clock::now();

// 执行一些代码，让它消耗一些时间。
cout << "计时开始 ..... \n";for (int ii = 0; ii < 1000000; ii++)
{
// cout << "我是一只傻傻鸟。 \n";
}
cout << "计时完成 ..... \n";
// 静态成员函数 chrono::steady_clock::now()获取结束的时间点。
auto end = chrono::steady_clock::now();
// 计算消耗的时间，单位是纳秒。
auto dt = end - start;
cout << "耗时: " << dt.count() << "纳秒 （" << (double)dt.count() / (1000 * 1000 *
1000) << "秒） ";}
```

C++11 线程

在 C++11 之前，C++没有对线程提供语言级别的支持，各种操作系统和编译器实现线程的方法不 一样。

C++11 增加了线程以及线程相关的类，统一编程风格、简单易用、跨平台。

创建线程

构造函数：

```
thread() noexcept; // 只创建对象，不创建任何子线程

template< class Function, class... Args >
explicit thread(Function&& fx, Args&&... args ); // 创建对象，创建线程，执行函数

thread(const thread& ) = delete; // 禁止线程对象间拷贝构造
```

1. 传递的函数可以是普通函数、类的非静态成员函数、类的静态成员函数、lambda函数、仿函数
- **普通成员函数需要保证生命周期比线程长**，防止内存泄漏
2. 先创建的子线程不一定跑的最快，程序有很大的偶然性

```
void test_create_thread()
{
// 无参时会被解释为函数，用myFunctor{}或:
thread t2((myFunctor()));

thread t1([]
{
for (int i = 1; i <= 10; ++i)
{
cout << "子线程1 no." << i << endl;
Sleep(100);
} });
;

cout << "开始" << endl;
for (int i = 1; i <= 10; ++i)
{
cout << "主线程 no." << i << endl;Sleep(100);
}
cout << "结束" << endl;

t1.join(); // 回收资源
t2.join();
}
```

其他

```
thread t1(func, 3, "我是一只傻傻鸟。"); // 普通函数
thread t5(mythread2::func, 3, "我是一只傻傻鸟。"); // 类的静态成员函数

// 类的普通成员函数，必须先创建类的对象，必须保证对象的生命周期比子线程要长
mythread3 myth;
thread t6(&mythread3::func, &myth, 3, "我是一只傻傻鸟。"); // 第二个参数必须填对象的this 指针，否则会拷贝对象
```

线程退出与回收

虽然同一个进程的多个线程共享进程的栈空间，但是，每个子线程在这个栈中拥有自己**私有的栈**空间。所以，线程结束时需要回收资源。

1. 任务完成后子线程将退出，如果主线程退出(正常/意外)，所有子线程被强行终止
2. 回收资源--join
   - 子线程已退出则立即回收资源并返回
   - 未退出，阻塞等待直到子线程退出，回收资源
3. 自动回收--分离子线程detach()，系统在子线程退出后自动回收，禁止join()，可以用joinable()判断分离状态
   - 主程序必须等所有子线程退出后退出

以上均为成员函数

this_thread

表示当前线程的命名空间，该命名空间中有四个函数：get_id()、sleep_for()、sleep_until()、yield()。

1. thread 类也有同名的get_id成员函数
2. sleep_for() VS Sleep(1000) Linux sleep(1)

```
template <class Rep, class Period>
void sleep_for (const chrono::duration<Rep,Period>& rel_time);
this_thread::sleep_for(chrono::seconds(1)); // 休眠 1 秒。
```

方便移植，不同平台的sleep不一样

3. 执行定时任务，注意是休眠至时间点

```
template <class Clock, class Duration>
void sleep_until (const chrono::time_point<Clock,Duration>& abs_time);
```

4. 让出自己的CPU时间片

```
void yield() noexcept
```

其他函数

不能拷贝、赋值，但是可以转移、交换

```
void swap(std::thread& other);

// 可以移动构造，转移资源所有权；转移后原对象不再代表线程；左值会被ban掉
thread(thread&& other ) noexcept;
thread& operator= (thread&& other) noexcept;
```

```
static unsigned hardware_concurrency() noexcept; // 返回硬件线程上下文的数量。
// The interpretation of this value is system- andimplementation- specific, and may not
// be exact, but just an approximation.
// Note that this does not need to match the actualnumber of processors or cores avail
// able in the system: A system can supportmultiple threads per processing unit, or restrict
// the access to its resourcesto the program.
// If this value is not computable or well defined,the function returns 0
```

call_once

保证某个函数在多线程过程中只能被调用一次

```
template< class callable, class... Args >
void call_once( std::once_flag& flag, Function&& fx, Args&&... args );
```

```
#include <mutex> // std::once_flag 和 std::call_once()函数需要包含这个头文件。
once_flag onceflag; // once_flag 全局变量。本质是取值为 0 和 1 的锁。
void func(int bh, const string &str) // 普通函数
{
// 传递一个只调用一次的函数及其参数
call_once(onceflag, once_func, 0, "各位观众");
// xxx
}
```

不能用if判断来执行函数与否--

native_handle

1. C++11 定义了线程标准，不同的平台和编译器在实现的时候，本质上都是对操作系统的线程库进行 封装，会损失一部分功能。
2. thread 类提供了 native_handle()成员函数，用于获得与操作系统相关的原生线程句柄，而后操作系统原生的线程库--注意，可能会导致不可移植的程序

```
int main()
{
thread tt(func); // 创建线程。
this_thread::sleep_for(chrono::seconds(5)); // 休眠 5 秒。

pthread_t thid= tt.native_handle(); // 获取 Linux 操作系统原生的线程句柄。
pthread_cancel(thid); // 取消线程。

tt.join(); // 等待线程退出。
}
```

线程安全

多个线程同时访问统一资源时，无需额外的同步机制，也能保证不会出现数据竞争、死锁等并发问题

- 单核机器不存在这些问题

——比如对cout的竞争引发的输出错乱，比如两个线程同时对同一个全局int a = 0;+分别一百万次1，结果也不会是二百万

一些概念

顺序性

程序按照代码的先后顺序执行，但是编译器不保证这一点，可能会对其做优化，只保证结果与按顺序执行时一致

可见性

CPU操纵变量时，会将其从内存加载到cache中，操纵后不一定会立即写回内存——此时其他线程访问的是原值

可见性即希望这种修改被立刻看到

原子性

CPU执行指令时，按照：读取指令--读取内存--执行指令--写入内存的顺序进行，在此过程中如果时间片用完则会被打断

原子性指，一个操作要么完整生效，要么不执行；上述情况不能保证原子性，aa++也不行

volatile

直接在内存中读写，且不允许编译器优化--只解决可见性问题，不能解决线程安全问题

存在性能损失：直接对内存操作、编译器限制优化、内存屏障引入、缓存一致性开销

线程同步

互斥锁

共享资源有人占用，就会等待；不被占用时可以访问，访问前加锁，解决问题后解锁

普通mutex

```
#include <mutex> // 互斥锁类的头文件。
mutex mtx;
int aa = 0;
void func(int bh, const string &str)
{
for (int ii = 1; ii <= 10; ii++)
{
mtx.lock(); // 申请加锁，锁上了就不能再被申请加锁，申请者们会形成等待队列
++aa; // 不是锁定资源，是锁定锁
mtx.unlock(); // 只有本线程能解锁；且只有解锁后才能加锁——同线程也不例外
}
}
```

另外还有成员函数try_lock()，如果获取锁成功便会返回true，失败则返回false，继续执行其他代码

超时互斥锁

timed_mutex

增加两个成员函数：

bool try_lock_for(时间长度); bool try_lock_until(时间点);

```
template< class Rep, class Period >
bool try_lock_for( const std::chrono::duration<Rep,Period>& timeout );

template< class Clock, class Duration >
bool try_lock_until( const std::chrono::time_point<Clock,Duration>& timeout_time );
```

递归互斥锁

recursive_mutex

解决死锁问题：本线程上锁后，调用另一个尝试上锁的函数，将永远等待，形成死锁

```
#include <iostream>
#include <mutex> // 互斥锁类的头文件。
using namespace std;
class AA
{
recursive_mutex m_mutex;

public:
void func1()
{
m_mutex.lock();
cout << "调用了 func1()\n";m_mutex.unlock();
}
void func2()
{
m_mutex.lock();
cout << "调用了 func2()\n";func1();
m_mutex.unlock();
}
};
int main()
{
AA aa;
// aa.func1();
aa.func2(); //
}
```

递归超时锁

同理

lock_guard类

可以简化互斥锁的使用，也更安全(防止意外没有执行unlock)

```
template<class Mutex>
class lock_guard
{
explicit lock_guard(Mutex& mtx);
}
```

在构造函数中加锁，在析构函数中解锁——自动加锁解锁，采用了RAII思想(在类构造函数中分配资源，在析构函数中释放资源，保证资源在离 开作用域时自动释放)——资源与生命周期绑定

条件变量和生产消费者模型

条件变量是一种线程同步机制。当条件不满足时，相关线程被一直阻塞，直到某种条件出现，这些线程才会被唤醒，C++提供了两个类：

- condition_variable：只支持与普通 mutex 搭配，效率更高
- condition_variable_any：通用的条件变量，可与任意 mutex 搭配(包括自定义的锁)

condition_variable 类

1. 有默认构造函数，禁止拷贝，禁止赋值
2. notify_one(), notify_all()
3. wait(unique_lock lock)，wait(unique_lock lock,Pred pred)
4. wait_for(unique_lock lock,时间长度)，wait_for(unique_lock lock,时间长度,Pred pred)
5. wait_until(unique_lock lock,时间点)，wait_until(unique_lock lock,时间点,Pred pred)

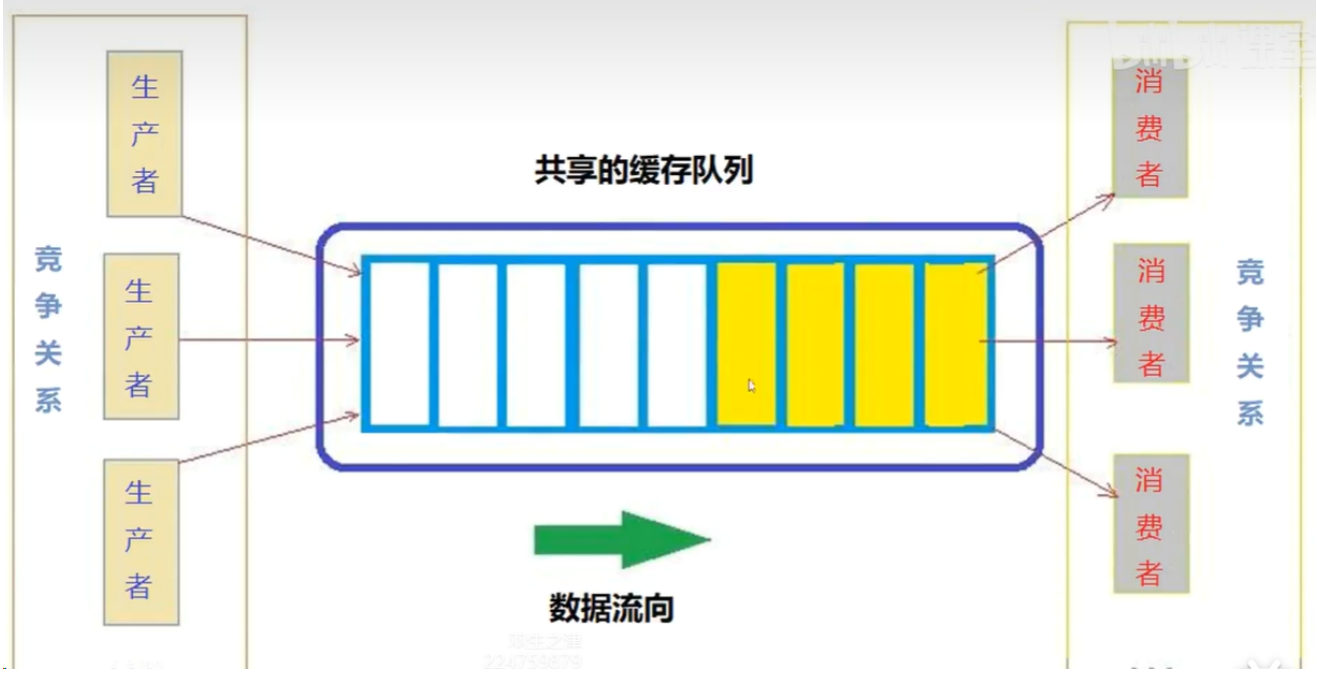
条件变量

- 当条件不满足时，相关线程被一直阻塞，直到某种条件出现，这些线程才会被唤醒。
- 为了保护共享资源，条件变量需要和互斥锁结合一起使用。
- 生产/消费者模型（高速缓存队列）。

unique_lock 类

RAII风格，为了配合 condition_variable，unique_lock 还有 lock()和 unlock()成员函数。而lock_guard没有

生产消费者模型



生产数据：加锁后生产消息，入队缓存队列，然后唤醒一个被阻塞的线程

```
void incache(int num) // 生产数据，num 指定数据的个数。
{
lock_guard<mutex> lock(m_mutex);
string message;
for (int ii = 0; ii < num; ii++)
{
static int bh = 1;
message = to_string(bh++) + "号";
m_q.push(message); // 进入缓存队列
}

m_cond.notify_one(); // 唤醒一个被当前条件变量阻塞的线程。
}
```

消费者任务函数：不退出

```
void outcache() // 消费者线程任务函数，不退出
{
while (true)
{
string message;

{
// 使用unique_lock管理原锁，并申请加锁(构造函数中)
unique_lock<mutex> lock(m_mutex); // 按理来说，多个线程只有一个线程不会被阻塞，但是wait改变这一情况

// 如果队列空，进入循环，否则直接处理数据。必须用循环，不能用if——防止拿不到数据的线程虚假唤醒notify_all
while (m_q.empty())
// 等待生产者的唤醒信号。解锁--阻塞等待被唤醒--唤醒后加锁
// 保证了所有当前线程被阻塞在这里，并且维护锁的机制
m_cond.wait(lock);
// 等价于如下函数，里面也有一个while循环
// m_cond.wait(lock, [this]
// { return !m_q.empty(); });

// 数据元素出队。
message = m_q.front();
m_q.pop();
cout << "线程： " << this_thread::get_id() << "， " << message << endl;
} // 自动解锁原锁，也可以不要{}进行手动解锁

// 处理出队的数据（把数据消费掉）。
this_thread::sleep_for(chrono::milliseconds(1)); // 假设处理数据需要 1 毫秒。
}
}
```

完整代码

```
#include <iostream>
#include <string>
#include <thread> // 线程类头文件。
#include <mutex> // 互斥锁类的头文件。
#include <deque> // deque 容器的头文件。
#include <queue> // queue 容器的头文件。
#include <condition_variable> // 条件变量的头文件。
using namespace std;
class AA
{
mutex m_mutex; // 互斥锁。
condition_variable m_cond; // 条件变量。
queue<string, deque<string>> m_q; // 缓存队列，底层容器用 deque。
public:
void incache(int num) // 生产数据，num 指定数据的个数。
{
lock_guard<mutex> lock(m_mutex);

for (int ii = 0; ii < num; ii++)
{
static int bh = 1;
string message = to_string(bh++) + "号";
m_q.push(message); // 进入缓存队列
}

// m_cond.notify_one(); // 唤醒一个被当前条件变量阻塞的线程。
m_cond.notify_all(); // 竞争抢缓存中的数据
}

void outcache() // 消费者线程任务函数，不退出
{
while (true)
{
string message;

{
// 使用unique_lock管理原锁，并申请加锁(构造函数中)
unique_lock<mutex> lock(m_mutex); // 按理来说，多个线程只有一个线程不会被阻塞，但是wait改变这一情况
```

```
// 如果队列空，进入循环，否则直接处理数据。必须用循环，不能用if——防止拿不到数据的线程虚假唤醒notify_all
while (m_q.empty())
// 等待生产者的唤醒信号。解锁--阻塞等待被唤醒--唤醒后加锁
// 保证了所有当前线程被阻塞在这里，并且维护锁的机制
m_cond.wait(lock);
// 等价于如下函数，里面也有一个while循环
// m_cond.wait(lock, [this]
// { return !m_q.empty(); });

// 数据元素出队。
message = m_q.front();
m_q.pop();
cout << "线程： " << this_thread::get_id() << "， " << message << endl;
} // 自动解锁原锁，也可以不要{}进行手动解锁

// 处理出队的数据（把数据消费掉）。
this_thread::sleep_for(chrono::milliseconds(1)); // 假设处理数据需要 1 毫秒。
}
}
};
int main()
{
AA aa;

thread t1(&AA::outcache, &aa); // 创建消费者线程 t1。
thread t2(&AA::outcache, &aa); // 创建消费者线程 t2。
thread t3(&AA::outcache, &aa); // 创建消费者线程 t3。
this_thread::sleep_for(chrono::seconds(2)); // 休眠 2 秒。
aa.incache(3); // 生产 3 个数据。
this_thread::sleep_for(chrono::seconds(3)); // 休眠 3 秒。
aa.incache(5); // 生产 5 个数据。
t1.join(); // 回收子线程的资源。
t2.join();
t3.join();
}
```

原子类型 atomic

只支持整型，如 bool、char、 int、long、long long、指针，从硬件级别保证了线程安全，效率较基本类型低，但是比锁和消息传递高

#include

1. 有默认构造函数，转换函数，禁用了拷贝构造函数与赋值函数
2. 常用：

```
void store(const T val) noexcept; // 存值
T load() noexcept; // 返回原值
T fetch_add(const T val) noexcept; // 原值+val，返回原值
T fetch_sub(const T val) noexcept; // 原值-val，返回原值
T exchange(const T val) noexcept; // 存值val，返回原值

T compare_exchange_strong(T & expect, const T val) noexcept; // CAS指令()

bool is_lock_free(); // 是否为原子性的，有的硬件可能不支持
```

3. 别名：

原子类型	相关特化类
atomic_bool	std::atomic<bool>
atomic_char	std::atomic<char>
atomic_schar	std::atomic<signed char>
atomic_uchar	std::atomic<unsigned char>
atomic_int	std::atomic<int>
atomic_uint	std::atomic<unsigned>
atomic_short	std::atomic<short>
atomic_ushort	std::atomic<unsigned short>
atomic_long	std::atomic<long>
atomic_ulong	std::atomic<unsigned long>
atomic_llong	std::atomic<long long>
atomic_ullong	std::atomic<unsigned long long>

4. 可以进行正常的运算；指针是原子类型不代表指向的对象也是原子类型
5. 原子整型可以用作计数器，布尔型可以用作开关
6. CAS 指令是实现无锁队列基础。

可调用对象与其包装器、绑定器、应用

可以调用的对象统称为可调用对象，或函数对象，可以用指针存储地址，可以被引用(类的成员函数除外)

可调用对象

类的普通成员函数

普通函数有函数类型，类的普通成员函数没有函数类型，只有指针类型，&和*不能省略——模板没法写

```
CC c;
c.show(1, "hi");
// 函数指针
void (CC::*ptr)(int, const string &) = &CC::show;
(c.*ptr)(1, "ok");

using ptr_Fun = void (CC::*)(int, const string &);
ptr_Fun pf = &CC::show;
(c.*pf)(1, "o");
```

可被转换为函数指针的类对象

重载类型转换运算符，意义不大

```
#include <iostream>
using namespace std;
// 定义函数
void show(int bh, const string& message) {
cout << "亲爱的" << bh << "，" << message << endl;
}
struct DD // 可以被转换为函数指针的类。
{
using Fun = void (*)(int, const string&);
operator Fun() {
return show; // 返回普通函数。
}
};
int main()
{
DD dd;
dd(17, "我是一只傻傻鸟。"); // 可以被转换为函数指针的类对象。
}
```

包装器 Function

用简单的、统一的方式处理可调用对象

```
#include <functional>

template<class _Fty> // 可调用参数的类型：返回类型(参数列表)
class function..... // 重载了bool运算符，判断是否包装了可调用对象，未包装就调用会抛出异常
```

传入可调用参数的类型即可

```
// 普通函数
function<void(int, const string &)> fn1 = show;
fn1(1, "我是一只傻傻鸟。 ");

// 包装类的静态成员函数
function<void(int, const string &)> fn3 = AA::show;
fn3(2, "我是一只傻傻鸟。 ");

// 包装仿函数
function<void(int, const string &)> fn4 = BB(); // 对象or匿名对象
fn4(3, "我是一只傻傻鸟。 ");

// lambda
function<void(int, const string &)> fn5 = lb; // 包装 lamba 函数。
fn5(4, "我是一只傻傻鸟。 ");

// 类的普通成员函数，第一个参数必是类的对象名，通过类的对象名才能调用
function<void(CC &, int, const string &)> fn11 = &CC::show;
fn11(cc, 5, "我是一只傻傻鸟。 "); // 第一个参数是对象

// 可以被转换为函数指针的类对象
function<void(int, const string &)> fn12 = dd;
fn12(6, "我是一只傻傻鸟。 ");
```

适配器/绑定器

用一个可调用对象及其参数，**生成一个新的可调用对象**，应用场景有：提前绑定缺省参数，调换参数位置，增加参数个数等等

```
#include <functional>
template< class Fx, class... Args >
function<> bind (Fx&& fx, Args&...args);
```

```
// 原来的样子，可变参数可以是左值、右值、参数占位符
function<void(int, const string &)> f1 = bind(show, placeholders::_1, placeholders::_2);

// 调换参数位置；此处int为占位符2，实际传递给show的参数为int,const string&
function<void(const string &, int)> f2 = bind(show, placeholders::_2, placeholders::_1);

// 提前绑定参数; 可变参数如果不是占位符，默认是值传递，需要引用传递可以用std::ref处理
// 第一个参数已绑定，故第二个参数对应的占位符是_1
int no_ = 1;
function<void(const string &)> f3 = bind(show, std::ref(no_), placeholders::_1);

// 增加参数的个数
function<void(int, const string &, int)> f4 = bind(show, placeholders::_1,
placeholders::_2);
f4(1, "toki", 111); // 第三个参数啥都行

// 类的非静态成员函数——最终统一六种函数的调用
CC c;
function<void(int, const string &)> f5 = bind(&CC::show, &c, placeholders::_1,
placeholders::_2);
f5(1, "ok");
```

可变参数函数的实现

```
class CC
{
public:
void show1(int a, const string &s) { cout << a << s << endl; }
};

// 和thread第二个构造函数类似，传递给其可调用对象及其参数，执行该函数
template <typename Function, typename... Args>
auto show(Function &&fn, Args &&...args) -> decltype(bind(forward<Function>(fn),
forward<Args>(args)...))
{
auto f = bind(forward<Function>(fn), forward<Args>(args)...); // 提前绑定所有参数
f();

return f; // 支持移动语义，并且可以把结果返回
}
int main()
{
CC c;
auto f = show(&CC::show1, &c, 1, "ok");
// f()
}
```

回调函数

```
struct CC
{
function<void(const string &)> callback_;

template <typename Function, typename... Args>
void callback(Function &&f, Args &&...args)
{ // 注册回调函数
callback_ = bind(forward<Function>(f), forward<Args>(args)..., placeholders::_1);
}

void show(const string &messages)
{
cout << messages << endl;
}
};
int main()
{
CC c;
c.callback(&CC::show, &c);
c.callback_("ok");
}
```

1.占位符是为了保证在调用m_callback时，传递的string在正确的位置上。

2.绑定m_callback时，如果原函数是类的普通成员函数，会传递一个this指针并提前绑定，并且占位原函数所需的参数

3.调用m_callback时，我们只会传递一个string，而不会传递this指针

4.于是我们实际给原类的成员函数传递的第一个参数是提前绑定的this指针，第二个参数是给m_callback函数传递的第一个参数，也即string

取代虚函数

1. C++虚函数在执行过程中会跳转两次（先查找对象的函数表，再次通过该函数表中的地址找到真正的执行地址），这样的话，CPU 会跳转两次，而普通函数只跳转一次。
2. CPU 每跳转一次，**预取指令要作废很多**，所以效率会很低。

这里我们**使用包装器和绑定器取代虚函数**，本质就是**每次注册不同的函数**

```
#include <iostream>
#include <functional>
using namespace std;
struct Hero
{
// 可调用对象show_
function<void()> show_;

template <typename Fn, typename... Args>
void callback_show(Fn &&fn, Args &&...args)
{
show_ = bind(forward<Fn>(fn), forward<Args>(args)...);
}
};

struct XS : public Hero
{
void show()
{
cout << "im xs" << endl;
}
};

int main()
{
Hero *ptr = nullptr;
if (true)
{
ptr = new XS;
ptr->callback_show(&XS::show, static_cast<XS *>(ptr)); // 注册"虚函数"
}
ptr->show_();
}
```

参考资料

《系统化学习C++》B站 课件

01大学C++、C语言课程

C Primer Plue

Chatgpt