# 内存模型

# 数据类型

## 内建类型

### 算术类型

算术类型包括基本算术类型和void特殊类型。



### 有符号/无符号类型

有三种不同的char类型：没有修饰的（plain）char、signed char和unsigned char。

注：plain char具体时间unsigned还是signed是由编译器决定的，C++标准没有规定。



确定要使用的整数类型和浮点类型：

1. 确定数值不能为负时，使用无符号类型
2. 整数计算时使用int
3. 不要再算术表达式中使用char或者bool，只可以用它们保留字符和真值
4. 在浮点计算中使用double

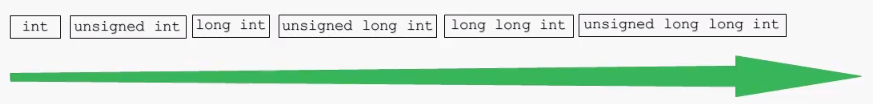
### 类型转换

### 字面量

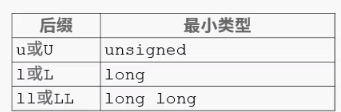
十进制整数字面量类型：



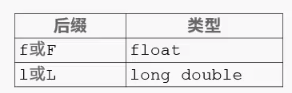
八进制和十六进制整数字面量的类型：



整数字面量后缀：



浮点数字面量后缀：



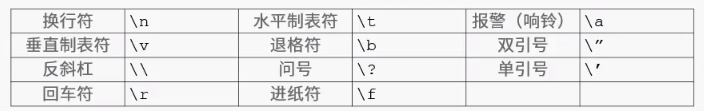
注：double是浮点数默认的类型。

字符和字符串字面量前缀：



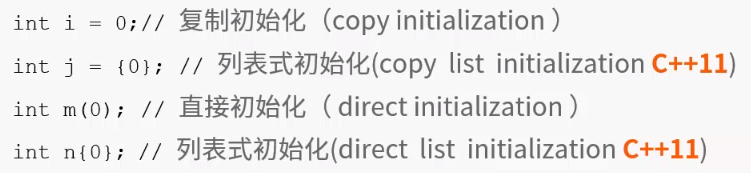
注：目前STL只提供了wchar\_t的输入和输出。

转义字符：



## 变量

变量的初始化：



## 复合类型

### 指针

### 引用

# 类型别名

## typedef

参考：<https://blog.csdn.net/superhoy/article/details/53504472>

## using

别名声明符：using

# 类型推断

## auto

## decltype

# 类型转换

为什么不使用C的强制转换？

C的强制转换表面上看起来功能强大什么都能转，但是**转化不够明确，不能进行错误检查，容易出错**。

C++中四种类型转换是：static\_cast，dynamic\_cast，const\_cast，reinterpret\_cast。

## static\_cast

用于各种**隐式转换**，比如**非const转const，void\*转指针**等, static\_cast能用于**多态向上转化，如果向下转能成功但是不安全，结果未知**。

### 实现

### 使用

template<typename T, typename U>

shared\_ptr<T> static\_pointer\_cast(constshared\_ptr<U> &other) noexcept {

T \*ptr = static\_cast<T \*>(other.get());

returnshared\_ptr<T>(other, ptr);

}

调用：

// shape\* -> circle\*使用static\_cast转换后，指针为空与dynamic\_cast相比，不安全

shared\_ptr<shape> sptr2(new shape);

shared\_ptr<circle> sptr1 = static\_pointer\_cast<circle>(sptr2); //基类转子类

cout << "use count of sptr1 is now " << dptr1.use\_count() << endl; // 0

cout << "use count of sptr2 is now " << dptr2.use\_count() << endl; // 1

// circle\* -> circle\*使用dynamic\_cast转换后，指针不为空，此时资源被两者共同使用，引用计数为2

shared\_ptr<shape> sptr3(new circle);

//shared\_ptr<circle> sptr3(new circle); // 上面或者当前行，后面输出一样！

shared\_ptr<circle> sptr1\_1 = static\_pointer\_cast<circle>(sptr3); // 基类转子类

cout << "use count of sptr1\_1 is now " << sptr1\_1.use\_count() << endl; // 2

cout << "use count of sptr3 is now " << sptr3.use\_count() << endl; // 2

// circle\* -> circle\* 使用static\_cast转换后,指针不为空,此时资源被两者共同使用,引用计数为2 等价于dynamic\_cast

shared\_ptr<circle> sptr3\_1(new circle);

shared\_ptr<shape> sptr2\_1 = static\_pointer\_cast<shape>(sptr3\_1); //子类转基类，上行转换，安全！

cout << "use count of sptr2\_1 is now " << sptr2\_1.use\_count() << endl; // 2

cout << "use count of sptr3\_1 is now " << sptr3\_1.use\_count() << endl; // 2

输出结果同上dynamic\_cast，不同之处，在下行转换的时候(基类转子类)，是不安全的！

**适用场景：**

1. 内置数据类型转换；
2. 父子类转换：

**向下类型转换（父类->子类）不安全，向上类型转换（子类->父类）安全**

没有父子关系的两个类之间是无法转换成功的

## dynamic\_cast

用于动态类型转换。**只能用于含有虚函数的类，用于类层次间的向上和向下转化**。**只能转指针或引用**。**向下转化时，如果是非法的对于指针返回NULL，对于引用抛异常**。要深入了解内部转换的原理。

向上转换：指的是子类向基类的转换

向下转换：指的是基类向子类的转换

它通过判断在执行到该语句的时候变量的运行时类型和要转换的类型是否相同来判断是否能够进行向下转换。

注：一般向上转换是安全的，向下转换因为有非法操作检查，因此动态类型转换在这类涉及到父类转子类的操作时很有用。

### 实现

### 使用

在unique\_ptr实现了子类向基类的转换，但是却没有实现基类向子类的转换，例如：:unique\_ptr<circle>转unique\_ptr<shape>。

实现这种，需要使用dynamic\_cast，实现如下：

首先为了实现这些转换，我们需要添加构造函数，允许在对智能指针内部的指针对象赋值时，使用一个现有的智能指针的共享计数。

// 实现强制类型转换需要的构造函数

template<typename U>

shared\_ptr(constshared\_ptr<U> &other, T \*ptr) noexcept {

ptr\_ = ptr;

if (ptr\_) {

other.shared\_count\_->add\_count();

shared\_count\_ = other.shared\_count\_;

}

}

其次，就是实现转换函数：

template<typename T, typename U>

shared\_ptr<T> dynamic\_pointer\_cast(constshared\_ptr<U> &other) noexcept {

T \*ptr = dynamic\_cast<T \*>(other.get());

returnshared\_ptr<T>(other, ptr);

}

调用：

// shape\* -> circle\* 使用dynamic\_cast转换后,指针为空.此时资源还是被dptr2拥有,dptr1为0

shared\_ptr<shape> dptr2(new shape);

shared\_ptr<circle> dptr1 = dynamic\_pointer\_cast<circle>(dptr2); // 基类转子类

cout << "use count of dptr1 is now " << dptr1.use\_count() << endl; // 0

cout << "use count of dptr2 is now " << dptr2.use\_count() << endl; // 1

// circle\* -> circle\* 使用dynamic\_cast转换后,指针不为空,此时资源被两者共同使用,引用计数为2

shared\_ptr<shape> dptr3(new circle);

// shared\_ptr<circle> dptr3(new circle); // 上面或者当前行,后面输出一样！

shared\_ptr<circle> dptr1\_1 = dynamic\_pointer\_cast<circle>(dptr3); // 基类转子类

cout << "use count of dptr1\_1 is now " << dptr1\_1.use\_count() << endl; // 2

cout << "use count of dptr3 is now " << dptr3.use\_count() << endl; // 2

// circle\* -> circle\* 使用dynamic\_cast转换后,指针不为空,此时资源被两者共同使用,引用计数为2

shared\_ptr<circle> dptr3\_1(new circle);

shared\_ptr<shape> dptr2\_1 = dynamic\_pointer\_cast<shape>(dptr3\_1); // 子类转基类 上行转换,安全！

cout << "use count of dptr2\_1 is now " << dptr2\_1.use\_count() << endl; // 2

cout << "use count of dptr3\_1 is now " << dptr3\_1.use\_count() << endl; // 2

dynamic\_cast主要用于类层次间的上行转换和下行转换，还可以用于类之间的交叉转换。在类层次间进行上行转换时，dynamic\_cast和static\_cast的效果是一样的；在进行下行转换时，dynamic\_cast具有类型检查的功能，比static\_cast更安全。在多态类型之间的转换主要使用dynamic\_cast，因为类型提供了运行时信息。

1、下行转换，基类转换为子类，例如：智能指针转换类似于shape\* 转换为circle\* 使用dynamic\_cast转换后,指针为空.此时资源还是被dptr2拥有,dptr1为0。比static\_cast安全。

2、平行转换，指向一致的相互转换，例如：智能指针转换类似于circle\*转换为circle\*。此时引用计数为两者共享。

3、上行转换，子类转基类，例如：智能指针转换类似于circle\*转换为shape\*，此时引用技术为两者共享。等价于static\_cast。

在数据库Item类型转换中，如果传入的是Item \*item，在该函数内判断该Item类型为Item\_func，则可以使用动态类型转换为子类：

Item\_func \*item\_func = dynamic\_cast<Item\_func\*> item。

适用场景：

1. 内置数据类型不适用（有精度损失或不安全转换不允许）；
2. 如果发生多态，父子之间转换总是安全的。

## const\_cast

参考：<https://blog.csdn.net/lwbeyond/article/details/6213382>

用于将const变量转为非const。

const\_cast < type-id > ( expression )

主要是用来去掉const属性，当然也可以加上const属性。主要是用前者，后者很少用。

去掉const属性：const\_case<int\*> (&num)，常用，因为不能把一个const变量直接赋给一个非const变量，必须要转换。

加上const属性：const int\* k = const\_case<const int\*>(j)，一般很少用，因为可以把一个非const变量直接赋给一个const变量，比如：const int\* k = j;

### 实现

### 使用

去掉const属性：

template<typename T, typename U>

shared\_ptr<T> const\_pointer\_cast(

constshared\_ptr<U> &other) noexcept {

T \*ptr = const\_cast<T \*>(other.get());

returnshared\_ptr<T>(other, ptr);

}

调用：

shared\_ptr<circle> s = const\_pointer\_cast<circle>(shared\_ptr<const circle>(new circle));

适用场景：

1. 指针或引用转换；
2. 不可以对非指针或非引用做const\_cast转换。

举例：

int a = 10;

const int& b = a;

int &c = const\_cast<int&>(b);

c = 20;

或：

const int\* p = NULL;

int\* p2 = const\_cast<int\*>(p);

int\* p3 = NULL;

const int\* p4 = const\_cast<const int\*>(p3);

比如在数据库中，如果需要获取Item\_field的name并进行修改，这个时候就需要用到const\_cast，因为Item\_field的name定义为：

const char \*field\_name;

因此，在使用的时候不能直接定义char \*变量接收，应该强转后，否则会报错：

Item\_field\* field = (Item\_field\*)in;

char \*filed\_name = const\_cast<char \*>(field->name);

比如数据库String使用的时候获取ptr指针也需要注意，String中的\*ptr()函数定义：

inline const char \*ptr() const { return Ptr; }

使用的时候：

String current;

current.reset\_buffer(buffer, sizeof(buffer));

item->print(&current, QT\_ORDINARY);

char \*c\_s = thd->strmake(const\_cast<char \*>(current.ptr()), current.length());

## reinterpret\_cast

几乎什么都可以转，比如将int转指针，可能会出问题，尽量少用。

### 实现

### 使用

例如：想把一个指针转为整数，就可以用reinterpret\_cast。

template<typename T, typename U>

shared\_ptr<T> reinterpret\_pointer\_cast(

constshared\_ptr<U> &other) noexcept {

T \*ptr = reinterpret\_cast<T \*>(other.get());

returnshared\_ptr<T>(other, ptr);

}

调用：

int a = reinterpret\_pointer\_cast<int>(s);

重新解释转换，最不安全，不建议使用（没有关系的类之间使用）。

# 标准输入输出

## cin

### cin.get

### cin.getline

### cin.ignore

### cin.peek

### cin.put\_back

## cout

### cout.put

### cout.write

### cout.flush

### cout.width

### cout.fill

### cout.self

### cout.unself

## cerr

## clog

## iostream

### istream

### ostream

## fstream

参考：<https://blog.csdn.net/weixin_42587961/article/details/86677869>

<https://blog.csdn.net/fengbingchun/article/details/51570728>

<https://blog.csdn.net/fengbingchun/article/details/51287210>

ofstream是从内存到硬盘，ifstream是从硬盘到内存，其实所谓的流缓冲就是内存空间。

### ifstream

### ofstream

### fstream

### filebuf

## sstream

参考：<http://www.cplusplus.com/reference/sstream/>

### istringstream

### ostringstream

### stringstream

### stringbuf

# 内存分配

## new/delete/malloc/free

delete会调用对象的析构函数，和new对应free只会释放内存，new调用构造函数。

malloc与free是C++/C语言的标准库函数，new/delete是C++的运算符。它们都可用于申请动态内存和释放内存。

对于非内部数据类型的对象而言，**光用malloc/free无法满足动态对象的要求**。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于malloc/free是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于malloc/free。因此C++语言需要一个能完成动态内存分配和初始化工作的运算符new，以及一个能完成清理与释放内存工作的运算符delete。

注意：**new/delete不是库函数**。

## delete与delete []

delete只会调用一次析构函数，而delete[]会调用每一个成员的析构函数。

在More Effective  C++中有更为详细的解释：“当delete操作符用于数组时，它为每个数组元素调用析构函数，然后调用operatordelete来释放内存。”delete与New配套，delete []与new []配套：

  MemTest \*mTest1 = new MemTest[10];

  MemTest \*mTest2 = new MemTest;

  int \*pInt1 = new int[10];

  int \*pInt2 = new int;

  delete []pInt1;  //-1-

  delete []pInt2;  //-2-

  delete []mTest1;//-3-

  delete []mTest2;//-4-

  在-4-处报错。

这就说明：对于内建简单数据类型，delete和delete[]功能是相同的。对于自定义的复杂数据类型，delete和delete[]不能互用。delete[]删除一个数组，delete删除一个指针。

简单来说，用new分配的内存用delete删除用new[]分配的内存用delete[]删除delete[]会调用数组元素的析构函数。内部数据类型没有析构函数，所以问题不大。如果你在用delete时没用括号，delete就会认为指向的是单个对象，否则，它就会认为指向的是一个数组。

# 指针与引用

## 引用

### 普通引用

在分析之前我们先介绍一下引用。引用为对象起了一个别名。例如：

int a = 1024；

int &b = a;

这里的b就是指向a，相当于a的另外一个名字。b是一个引用，但并非一个对象，同时引用的类型都要和对象的类型严格匹配（在具有继承关系的类中是例外，这里不展开），因此有下面的操作：

int a = 1024；

int &b = a;

int &c = b; //非法，b在此之前已经是引用，不能定义引用的引用

int &d = 1024; //非法，引用类型的初始值必须是对象，而不能是字面值

const int &e = 1024 //合法，e为常量引用，只要能够转换成引用的类型，可以使用任何表达式作为初始值

double f = 1.024；

int &g = f; //非法，类型不匹配

而对引用赋值，实际上是将值赋给了与引用绑定的对象，例如：

int h = 1024；

int &i = h;

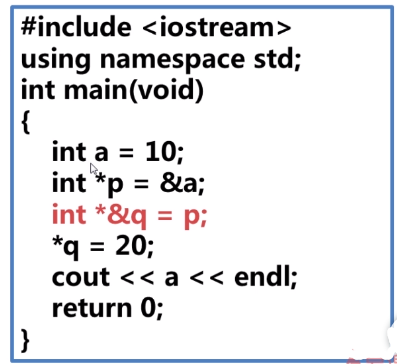
i = 256;//将256赋给了h

同时，引用必须被初始化，下面的操作是非法的：

int &j;//非法，没有初始化

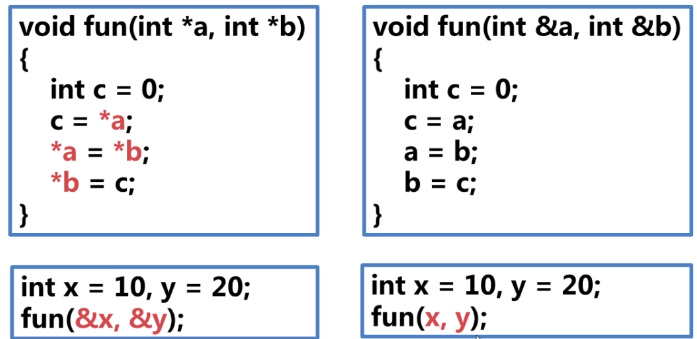
### 指针类型的引用

格式：类型 \*&指针引用名 = 指针;



注：**指针引用就是表示指向的是指针类型的引用，其实与普通数据的引用没有差别**。比如int\* &q表示的是int\*类型的引用q。此时输出的是a=20。

### 引用作为函数参数



### const与引用

int x = 3;

const int &y = x;

//x=10;正确

//y=20;错误

## 指针

学习过C的朋友可能对指针已经比较熟悉了，这里不占用较大篇幅。它存储的是一个地址，地址指向的是一个对象。例如：

int a = 0; //定义变量a

int \*ptr = &a;定义int类型指针ptr，它存放变量a的地址

与引用类似，指针也实现了对其他对象的间接访问。同样的，它也要求指针的类型和指向的类型严格匹配。

## 比较

### 引用不可为空，而指针可以

例如：

int &a; //非法，必须被初始化

int \*p;//合法，但如果是非静态的指针变量，将拥有一个不确定的值

声明一个指针变量而不初始化是合法的，但是不建议这么做，原因可参见C语言入坑指南-被遗忘的初始化。正因如此，使用引用而非指针作为函数的参数是一个不错的选择，因为引用永远不为空，函数入口也就不需要做过多的检查，引用也就更富效率。

### 引用不是对象，而指针本身是对象

引用不是对象（只是对象的别名），它不能被再次赋值，而指针是可以的。例如：

int a = 1024;

int b = 2048;

int &c = a;

int \*d = &a;

c = b; //其引用不会被再次赋值

d = &b;//合法，指针可以指向另外一个对象

也就是说，**一旦引用了引用，就没法让它绑定到其他的对象**。

另外，可以有指向指针的指针，而不存在引用的引用。因为引用不是对象。

int a = 1024;

int &b = a;

int &&c = b;//非法

int \*d = &a;

int \*\*e = &d;//合法

### 引用指向对象本身，而指针指向的是对象的地址

定义一个指针的时候，编译器为他分配内存，而引用不会单独分配空间。

引用所代表的就是最初绑定的那个对象，因此使用sizeof分别作用于引用和指针时，前者得到的是引用所绑定对象大小，而后者得到的是指针占用空间大小（4或8字节），例如在64位的程序中：

/\*\*假设有以下结构\*/

typedef struct INFO

{

int a;

int b;

char c;

}INFO;

INFO info = {0};//定义结构info

INFO &ref = info;//定义引用ref

INFO \*ptr = &info;//定义指针ptr

sizeof(ref); //大小为INFO结构体占用空间大小，即12字节

sizeof(ptr); //占用大小为指针占用空间大小，即8字节

### 引用使用时无需解引用，而指针需要

例如：

int a = 1024;

int &ref = a;

int \*p = &a;

ref = 10; //使用引用可直接使用，将所绑定对象的值修改

\*p = 11;//使用指针需要解引用

### 作为参数时的差别

指针作为参数时，看起来是地址传参，实际上仍然传值，即将指针的一个拷贝作为实参，而由于指针指向的是一个对象，因此在函数内可以实现对指针所指向对象的内容进行改变。而引用作为参数时，实际上传递的对象本身，但又不需要拷贝，因为引用绑定的就是对象。

## 区别

1、指针有自己的一块空间，而引用只是一个别名；

2、指针在使用中可以指向其它对象，但是引用只能是一个对象的引用，不能被改变；

3、指针可以被初始化为NULL，而引用必须被初始化且必须是一个已有对象的引用；

4、作为参数传递时，指针需要被解引用才可以对对象进行操作，而直接对引 用的修改都会改变引用所指向的对象；

5、可以有const指针，但是没有const引用；

6、使用sizeof看一个指针的大小是4，而引用则是被引用对象的大小；

7、指针可以有多级指针（\*\*p），而引用只有一级；

8、指针和引用使用++运算符的意义不一样；

9、如果返回动态内存分配的对象或者内存，必须使用指针，引用可能引起内存泄露。

## 使用

1、如果使用一个变量并让它指向一个对象，但是该变量在某些时候也可能不指向任何对象，这时应该把变量声明为指针，因为这样可以赋空值给该变量；

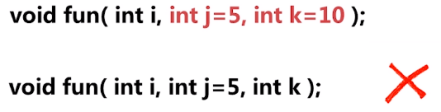
2、如果变量肯定指向一个对象，例如设计中不允许变量为空，这时就可以把变量声明为引用；

3、重载操作符时应当返回引用（主要是为了减少不必要开销，引用效率高）。

# 函数

## 参数默认值

有默认参数值的参数必须在参数表的最右端：



无实参则用默认值，否则实参覆盖默认值。

## 内联函数



既然内联函数有这样的特点，为什么不是所有函数都用内联函数？

1. 内联编译时建议性的，由编译器决定；
2. 逻辑简单，调用频繁的函数建议使用内联；
3. 递归函数无法使用内联方式

# 异常

C语言处理异常的缺陷在于，返回的值没有统一（比如返回值-1一般表示错误，但是也可以表示其他情况），返回的值可以是异常的结果，也可以是正常的结果，C++中使用try-catch-throw。

栈解旋：从try代码块开始起，到throw抛出异常前，所有栈上的对象都被释放掉，释放的顺序和构造的顺序是相反的，这个过程称为栈解旋。

# 封装性

## 类和对象

C++将对象抽象成类进行处理，由于目的的不同抽象出的信息也可能不同。

根据目的不同，决定哪些信息需要暴露出去，哪些信息需要隐藏起来，这些通过访问限定符（public、protected、private）。如果需要暴露信息使用public访问控制符，如果需要隐藏信息则使用private访问控制符。

**class与struct区别：**

1、字面上的区别

在字面上struct是structure的缩写，通常叫做“结构体”，在C语言里用于将多种数据、多个变量组织在一起，便于表达比较复杂的数据类型，在C++中为了兼容C语言保留了该关键字，并且保留了C语言中的所有功能。

而class，则称作“类”，是C++新增来支持面向对象思想概念中“类”的概念的一个关键词，并且比struct具有了更强大的功能，不仅可以像C语言中的struct一样把数据组织在一起，还可以将与数据相关的方法组织在一起，并增加了如虚函数、继承等特性来支持面向对象编程。

虽然在字面上struct与class的含义不一样，但在C++中其功能基本是相同的，C++中的struct不仅可以包含数据成员，而且与class一样支持新增的面向对象特性，仅在以下细节上有略微差别。

既然两者在字面上不一样，为了更好地利用这一点，建议在C++中使用struct时仍然只使用C中的特性，来表示一些复杂的数据而不进行方法的封装，这样还可以提高软件的可读性。

2、默认成员权限区别

struct的成员默认权限是public，而class的成员默认权限是private。例如：

struct Sb{ void f( void ) { } };

struct Sc : Sb{ };

class Cb{ void f( void ) { } };

class Cc1 : public Sb{ };

class Cc2 : Sb{ };

sb.f(); // 合法

cb.f(); // 不合法，因为在Cb类中f( )函数默认为private，此处不可访问

建议在工程代码中显示声明成员的权限，而有使用默认权限，因为并不是所有人都知道这一点。

3.、默认继承方式

struct的默认继承方式为public，而class的默认继承为private，例如：

Sc sc; Cc1 cc1; Cc2 cc2;

sc.f(); // 合法

cc1.f(); // 合法，因为Cc1显示地使用public继承

cc2.f(); // 不合法，Cc2默认private继承自Sb，在Cc2中f( )为private

在C语言中struct不可以继承，虽然我们知道在C++中struct可以继承，但在实际使用中，在不需要继承的场合我们使用struct，而在需要继承的场合使用class，这样更贴近其字面意思，使程序有更好的可读性。

虽然知道class的默认继承为private，但并不是项目组的每个人都清楚这一点，建议在工程代码中不省略private，使代码可读性更强。

4、用于定义模板参数

模板为C++语言新增特性，C语言没有，只有class可用于定义参数，而struct不可以，例如：

template

class TValue {

private: T \_v;

public: TValue(T v) : \_v(v){}

T Get( void ) { return \_v; }

};

此处只能使用class，不能使用struct。当然，此处还可以使用typename代替class，class与typename也仅在定义模板参数时可以互换，而且建议此时使用typename，因为这样读起来更接近人类语言，更具有可读性。

## 对象实例化

类就是一个模板，所谓的对象实例化就是根据一个类的设定，制造多个对象的过程。

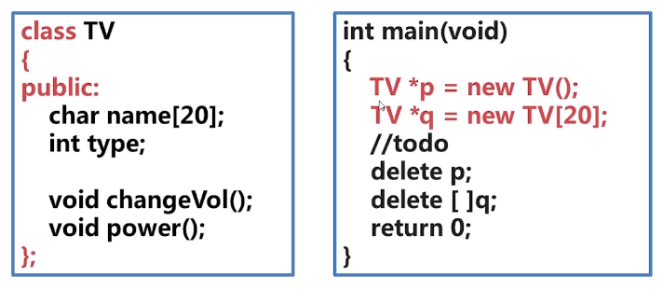
有两种实例化对象的方式：

1. 从栈中实例化：



从栈中实例化对象不需要手动释放内存，系统完成这部分工作，但是在堆中实例化则需要手动释放内存。

1. 从堆中实例化：



在堆中实例化对象需要手动释放内存，如果实例化的是一个对象，则采用delete p的形式，如果是实例化对象数组，那么采用delete []p的形式。

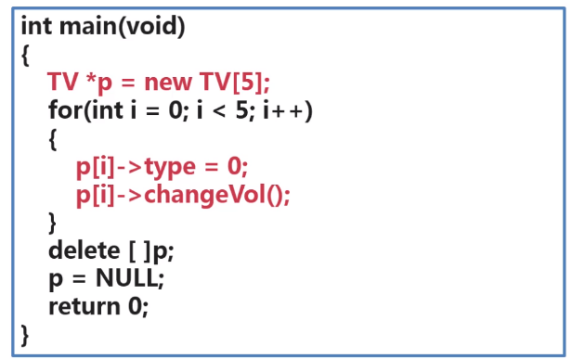
## 对象成员访问

实例化对象后需要访问对象的成员，根据栈和堆两种实例化方式的不同，访问成员的方式也有所不同。

针对单一对象而言访问方式：



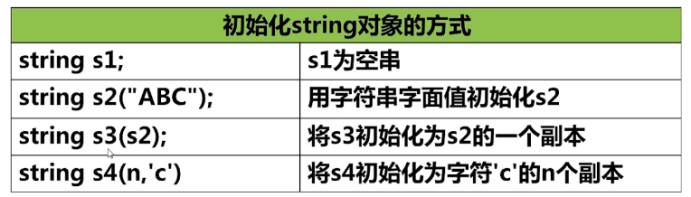
针对对象数组而言访问方式：



## String

C语言中字符处理使用char，C++中可以使用字符串string，这是封装的对象。

### 初始化



### 常用操作



有一种连接方式需要格外注意：



只有string变量与双引号“”的字符串结合的时候才是合法的，两个“”字符串结合是非法的，比如这里的s6。

**拓展：为什么Java的String设置为不可变的呢？**

Java中将String设计成不可变的是综合考虑到各种因素的结果，想要理解这个问题，需要综合内存，同步，数据结构以及安全等方面的考虑：

1、字符串常量池的需要

字符串常量池(String pool, String intern pool, String保留池) 是Java堆内存中一个特殊的存储区域，当创建一个String对象时，假如此字符串值已经存在于常量池中，则不会创建一个新的对象，而是引用已经存在的对象。

假若字符串对象允许改变，那么将会导致各种逻辑错误，比如改变一个对象会影响到另一个独立对象。严格来说，这种常量池的思想，是一种优化手段。

2、允许String对象缓存HashCode

Java中String对象的哈希码被频繁地使用，比如在hashMap 等容器中。

字符串不变性保证了hash码的唯一性，因此可以放心地进行缓存。这也是一种性能优化手段，意味着不必每次都去计算新的哈希码。

3、安全性

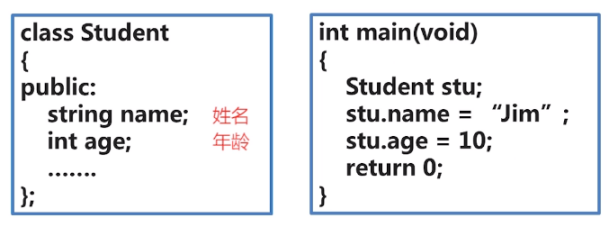
String被许多的Java类(库)用来当做参数。例如：网络连接地址URL，文件路径path，还有反射机制所需要的String参数等，假若String不是固定不变的，将会引起各种安全隐患。

## 数据的封装

所谓数据的封装就是将类中成员变量的操作转换为成员函数的操作，这样才是封装性的体现，即不直接暴露成员变量本身，而是暴露出成员变量操作的函数。

面向对象的思想就是以谁做什么为基础，反映到代码层面上就是类的成员函数。

如下例子就是违背了这种思想：



应该采用如下的方式：



这样具备两种好处：

1、更加符合面向对象的思想；

2、可以根据需要对参数进行校验：



3、可以设置只读属性：

有些时候我们可能需要设置某些成员变量只能被读或者写操作，这样就单独设置仅有getter或者setter中的某一个操作函数即可。



## inline/内联函数

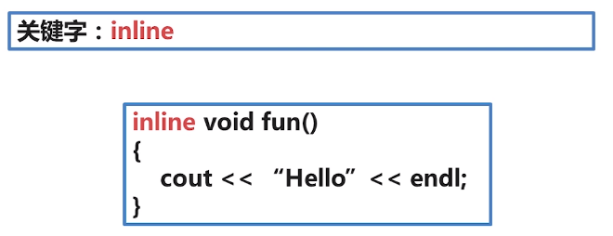
**C++中推荐使用内联函数替代宏代码片段；**

宏代码块看上去像函数但实际不是函数，因此其使用常带有副作用；

消除这样的副作用需要函数，但是函数的调用有入栈返回等开销，这一点宏代码块没有；

综合两者有点，避免两者缺点，内联函数被提出来。

### 语法



### 普通函数

内联函数与普通函数的区别：



内联函数相当于省去了2、4调用和返回的过程，只有一个3函数本身执行的过程（即**内联函数没有普通函数调用时的额外开销（压栈，跳转，返回）**）。这样**效率更高，速度更快**，但是仅仅适用于结构比较简单的函数，如果函数逻辑复杂，即使设置内联函数，编译器也会拒绝按照内联函数的方式处理。

### 虚函数

虚函数可以被声明为inline，这是毫无疑问的，因为inline同register一样，只是对编译器的建议。

**那么inline virtual函数是否会被内联展开呢？答案是可以！**

有的人说不能，理由是virtual的意思是"等到运行时再决定调用哪个函数"，inline的意思是"在编译期间将调用之处用被调函数来代替"，如果编译器甚至还不知道哪个函数将被调用，当然就不能责怪它拒绝生成内联调用了。

其实上面的观点是不正确的，如果virtual可以在编译期决定调用什么函数，那么就可以被inline。

**那么什么时候可以决定调用何种函数呢？**

用一个类对象通过成员选择符.调用虚函数，如obj.vf()。这时虚函数vf()就可以被vc优化内联展开。这样调用等于告诉编译器你要调用的具体函数，在函数有inline修饰或是体内定义的情况下就会被内联展开。

当然还有其他情况可以被inline，总的意思是虚函数可以申明为inline，具体到编译器解释时候，如果编译器在编译的时候就可以确定该虚函数的决议，则编译器以inline方式静态决议该虚函数。如果编译器在编译的时候不能决定，则必须在运行时决议虚函数，此时虚函数不能以inline函数的方式调用。

### 特征

1、内联函数具有普通函数的特征（参数检查，返回类型等）；

类型更加安全；

C++ 提供各种方式让大家摒弃 C 中不好的一些特性，内联函数就是其中之一；

2、函数的内联请求可能被编译器拒绝；

可以通过配置的方式让编译器支持内联函数的请求；

3、函数被内联编译后，函数体直接扩展到调用的地方；

如果内联成功，其效率上可以和宏代码块媲美，且其在类型上比宏代码块安全很多，因此在 C++ 编程中，首选内联函数；

宏代码片段由预处理器处理，进行简单的文本替换，没有任何编译过程，因此可能出现副作用；

### 限制

C++中inline内联编译的限制：

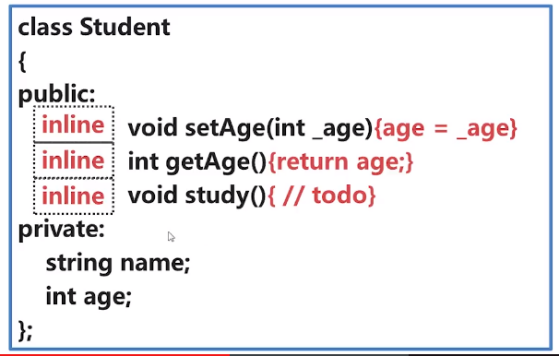
1. 不能存在任何形式的循环语句；
2. 不能存在过多的条件判断语句；
3. 函数体不能过于庞大；
4. 不能对函数进行取址操作；
5. 函数内联声明必须在调用语句之前；

总的来说就是函数体不能过于复杂，这个复杂没有界定，对于现在C++编译器，只要函数不是太夸张，几乎都可以满足请求。

## 类内/类外定义

### 类内定义

如果将成员函数和函数体写在类的内部，这就是类内定义。类内定义的成员函数，编译器会优先将其编译为inline函数，它不会显式地标识inline，而是优先按照inline编译。对于复杂的函数，无法编译为内联函数的，则编译为普通函数。

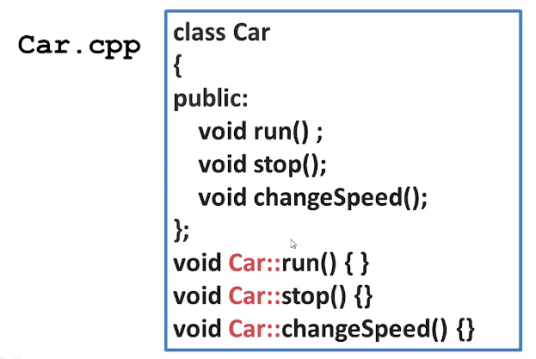


既然将这个函数实现定义在类内部，系统就默认为这个函数比较短小，可以采用inline方式编译，如果是分开的那么就不会默认为该函数非常短小了。

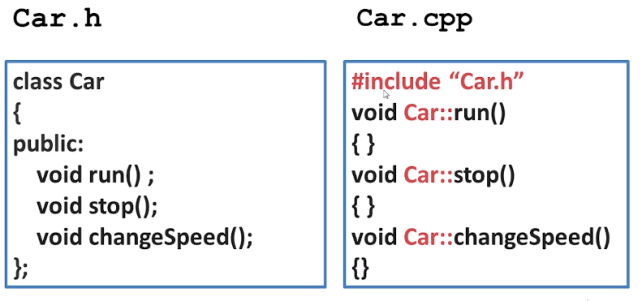
### 类外定义

类外定义是成员函数和函数体写在类的外面。包括两种：同文件类外定义和分文件类外定义。

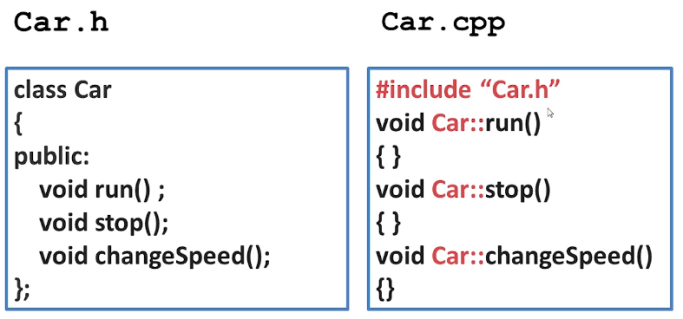
同文件类外定义：



分文件类外定义（常用）：

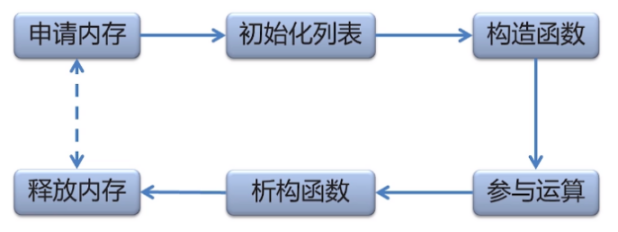


专业的C++项目都是按照分文件定义的方式来组织代码的。首先，要定义一个头文件.h，这里声明成员变量和成员函数；然后，在.cpp文件中导入.h文件（即声明），实现具体函数。



## 构造函数

在理想的世界中，构造函数和析构函数是不存在开销的。它们将只执行必要的初始化和清除工作，普通的编译器都会内联它们。理论上如此，但是实际情况与此稍有不同。



**析构函数可以为virtual型，构造函数则不能。为什么构造函数不能为虚的？**

虚函数采用一种虚调用的办法。**虚调用是一种可以在之有部分信息的情况下工作的机制，特别允许我们调用一个只知道接口而不知道其准确对象类型的函数**。但是如果要创建一个对象，你势必要知道对象的准确类型，因此构造函数不能为虚。

**不是所有的函数都能自动地从基类继承到派生类中的。**

构造函数和析构函数是用来处理对象的创建和析构的，它们只知道对在它们的特殊层次的对象做什么。

所以，在整个层次中的所有的构造函数和析构函数都必须被调用，也就是说，**构造函数和析构函数不能被继承**。

子类的构造函数会显示的调用父类的构造函数或隐式的调用父类的默认的构造函数进行父类部分的初始化。

析构函数也一样。它们都是每个类都有的东西，如果能被继承，那就没有办法初始化了。

### 构造原则

在创建子类对象时，为了初始化从父类继承来的数据成员，系统需要调用其父类的构造方法。

如果没有显式的构造函数，编译器会给一个默认的构造函数，并且该默认的构造函数仅仅在没有显式地声明构造函数情况下创建。

构造原则如下：

1. 如果子类没有定义构造方法，则调用父类的无参数的构造方法。
2. 如果子类定义了构造方法，不论是无参数还是带参数，在创建子类的对象的时候,首先执行父类无参数的构造方法，然后执行自己的构造方法。
3. 在创建子类对象时候，如果子类的构造函数没有显示调用父类的构造函数，则会调用父类的默认无参构造函数。
4. 在创建子类对象时候，如果子类的构造函数没有显示调用父类的构造函数且父类自己提供了无参构造函数，则会调用父类自己的无参构造函数。
5. 在创建子类对象时候，如果子类的构造函数没有显示调用父类的构造函数且父类只定义了自己的有参构造函数，则会出错（如果父类只有有参数的构造方法，则子类必须显示调用此带参构造方法）。
6. 如果子类调用父类带参数的构造方法，需要用初始化父类成员对象的方式。

### 内存分布/内存模型

要想知道对象是怎么存储的，需要知道内存按照用途是怎么划分的：



注：在构造函数的时候，其实就是构造内存模型。

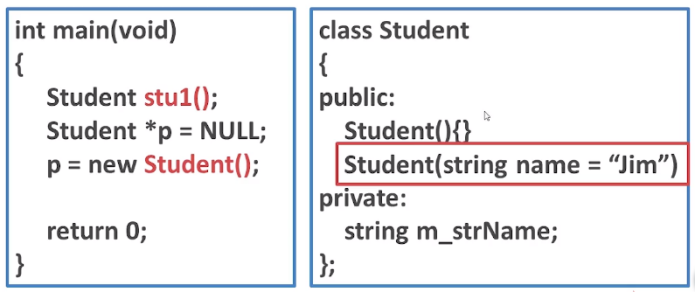
### 对象初始化

一个对象如果new之后没有进行设置，那么就相当于我们知道了地址但是没有内容，所以需要先进行对象的初始化操作。

对象的初始化包括：有且仅有一次，根据条件初始化。

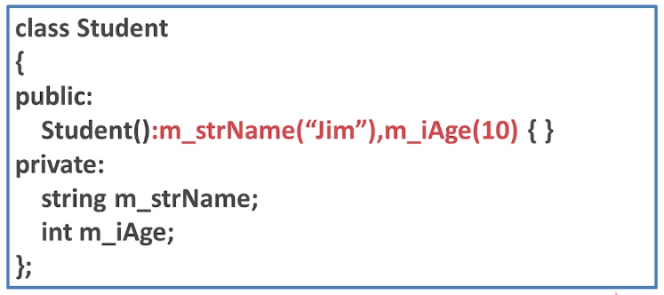
有且仅有一次的初始化就是构造函数，构造函数没有返回值，可以重载，但是实例化对象时只用到其中一个构造函数。

在实例化对象的时候，如果不传入参数，则成为默认的构造函数，如下：



### 初始化列表

参考：<http://events.jianshu.io/p/58c690713cb3>



注：使用初始化列表进行初始化的时候，需要在构造函数后加“：”，赋值操作不使用“=”，而是使用“()”，多个类成员变量之间使用“,”。

初始化列表的一些基本特性：

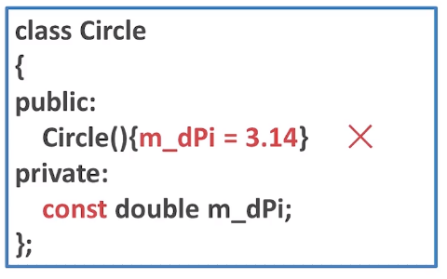
1、初始化列表先于构造函数执行；

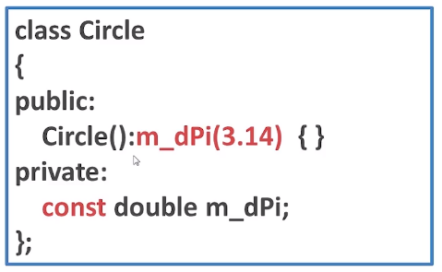
2、初始化列表只能用于构造函数；

3、初始化列表可以同时初始化多个数据成员。

注：初始化列表赋值方式和构造函数在顺序上有区别。

既然初始化列表的工作可以由构造函数完成，那么为什么还需要初始化列表呢？





注：对于const类型的成员变量，如果在构造函数中进行初始化，则会报错，只能在构造函数之前（即真正创造对象前）进行初始化赋值。

在C++11中使用初始化列表进行初始化的数据编译器是会检查其是否发生类型收窄的。

**在C++11中，列表初始化是唯一一种可以防止类型收窄的初始化方式**。

**实例：**

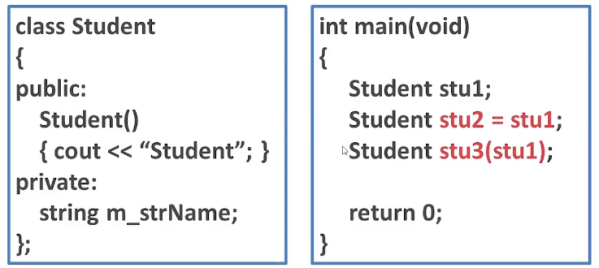




### Explicit

1. explicit用于阻止编译器的隐式转换，一般用于修饰构造函数。
2. C++98/03由于不能使用{}列表初始化，即隐式转换只发生在调用带一个参数的构造函数中。但C++11允许用{}列表初始化对象，这可能会隐式调用带多个参数的构造函数。
3. C++11中，explicit可用于修饰带多个参数的构造函数以防止隐式转换。（注意C++98/03中explicit被用于修饰只带一个参数的构造函数，如果修饰带多个参数的构造函数则无效）

### 拷贝构造函数



注：这里只输出一个Student（即实际上只调用了1次构造函数，stu1调用的），并不是我们预期的3个（按照预期应该stu1,stu2,stu3调用3次构造函数），这是因为这里没有调用我们预期的Student()构造函数，而是调用了默认的拷贝构造函数。

拷贝构造函数定义：



注：拷贝构造函数参数是const+对象的引用

说明：

1、如果没有自定义的拷贝构造函数则系统默认自动生成一个默认的拷贝构造函数；

2、当采用直接初始化或者复制初始化实例化对象时系统自动调用拷贝构造函数。

## 析构函数

析构函数与构造函数对应，当对象结束其生命周期，如对象所在的函数已调用完毕时，系统会自动执行析构函数。

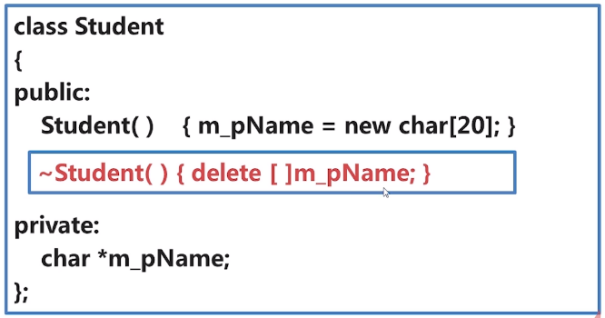
析构函数名也应与类名相同，只是在函数名前面加一个位取反符~，例如~stud( )，以区别于构造函数。**它不能带任何参数，也没有返回值（包括void类型）。只能有一个析构函数，不能重载**。

如果用户没有编写析构函数，编译系统会自动生成一个缺省的析构函数（即使自定义了析构函数，编译器也总是会为我们合成一个析构函数，并且如果自定义了析构函数，编译器在执行时会先调用自定义的析构函数再调用合成的析构函数），它也不进行任何操作。所以许多简单的类中没有用显式的析构函数。

如果一个类中有指针，且在使用的过程中动态的申请了内存，那么最好显式构造析构函数在销毁类之前，释放掉申请的内存空间，避免内存泄漏。

类析构顺序：

1. 派生类本身的析构函数；
2. 对象成员析构函数；
3. 基类析构函数。



说明：

**析构函数没有参数不允许重载！**

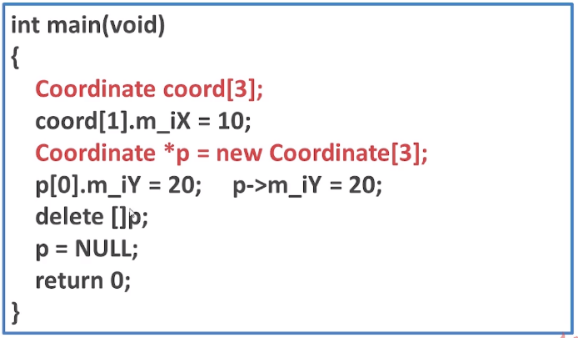
析构，就是一个对象自杀。把自己占有的内存空间释放，把自己的名字从对象列表删除，对于任何对象，这个过程是一样的，没有必要重载。重载构造函数的目的，是用于通过多种方式，构造一个新对象。

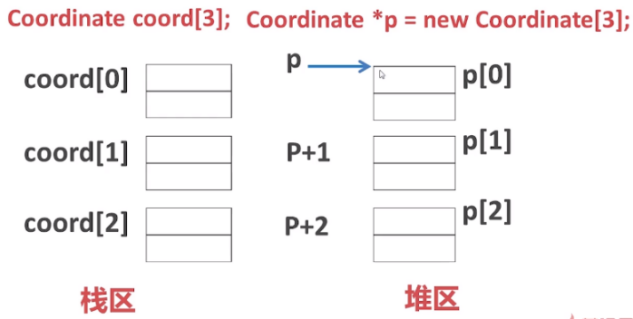
将可能会被继承的父类的析构函数设置为虚函数，可以保证当我们new一个子类，然后使用基类指针指向该子类对象，释放基类指针时可以释放掉子类的空间，防止内存泄漏。

**C++默认的析构函数不是虚函数是因为虚函数需要额外的虚函数表和虚表指针，占用额外的内存**。**而对于不会被继承的类来说，其析构函数如果是虚函数，就会浪费内存**。

因此**C++默认的析构函数不是虚函数，而是只有当需要当作父类时，设置为虚函数**。

## 对象数组





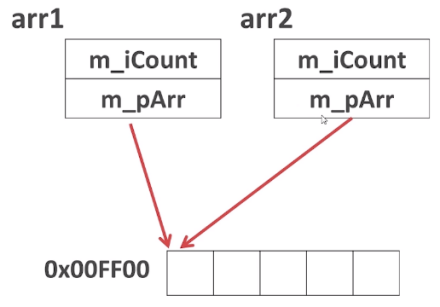
## 深浅拷贝

参考：

<https://www.cnblogs.com/mikeCao/p/8710837.html>

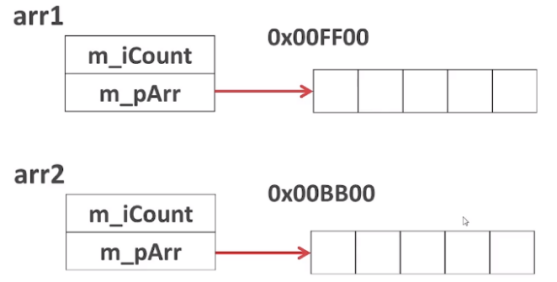
### 浅拷贝





浅拷贝只是将数据成员的值进行简单的拷贝（不管是不是指针），上述arr1和arr2会同时指向一个内存地址，如果往arr1写入一个值，然后再往arr2写入另一个值，那么这个地址对应的值就会被重写覆盖原来的值，这显然不是我们期望的。还有更严重的，在销毁arr1的时候，为了避免内存泄露，会释放m\_pArr的内存，同理arr2在销毁的时候也会释放这个内存，这样同一个内存地址释放了两次，计算机会core掉。

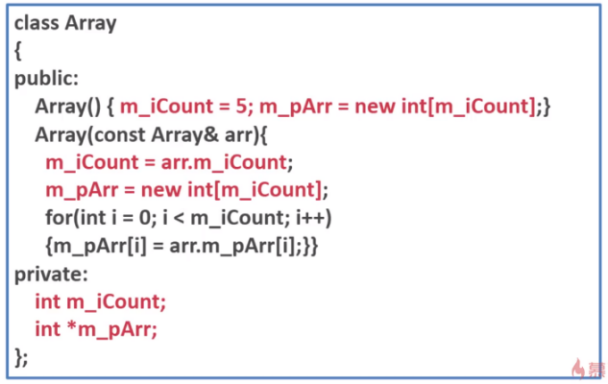
我们期望的是内存地址指向的数据依次执行拷贝，即如下：



即在拷贝构造函数中不是简单的赋值操作，而是重新分配内存。

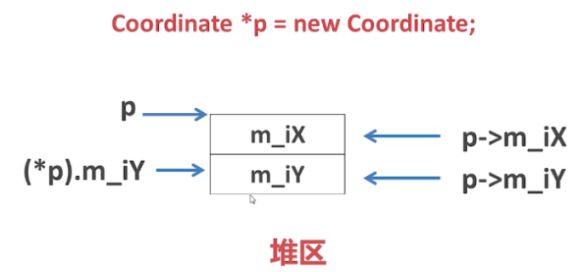
### 深拷贝

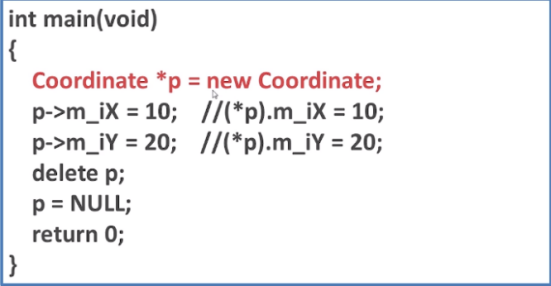
深拷贝不是简单的执行拷贝，而是将堆中的内存数据执行拷贝。



注：与浅拷贝不同的是，在自定义拷贝构造函数中使用new生成一个新的对象。

## 对象指针

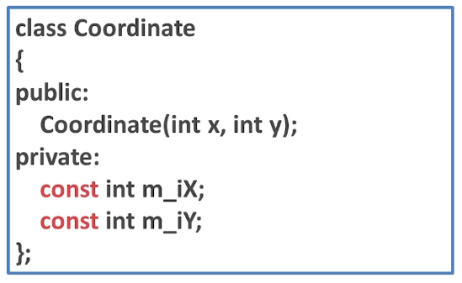


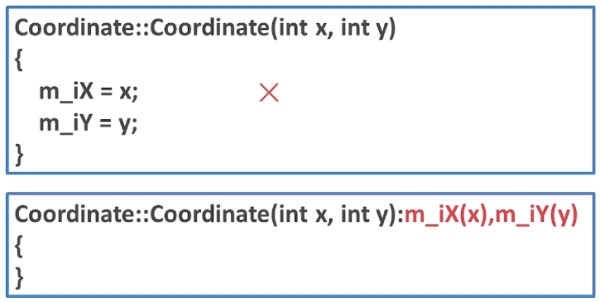


注：(\*p).x与p->x一样

## const

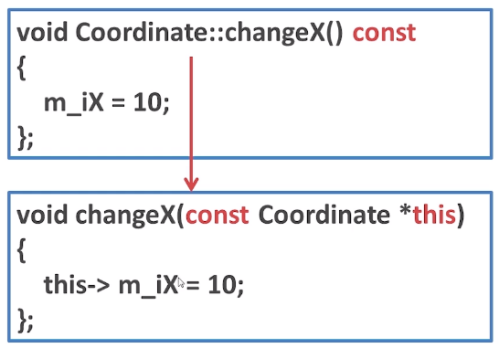
### 常对象成员



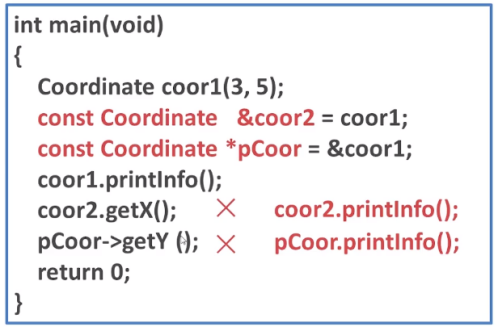


注：常对象成员即一旦被初始化就不允许改变，此时使用初始化列表进行赋值，在有些应用场景是期望这种对象的。

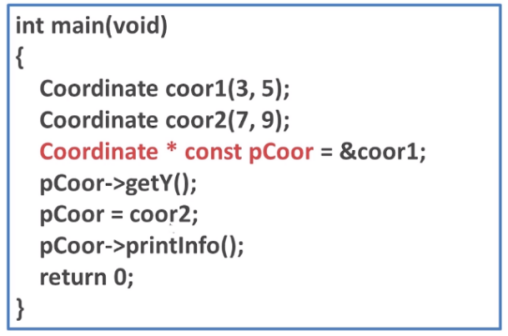
### 常成员函数



### 常指针与常引用



可以采用如下：



# 继承性

C++通过public、protected、private三个关键字来控制成员变量和成员函数的访问权限，它们分别表示公有的、受保护的、私有的，被称为成员访问限定符。

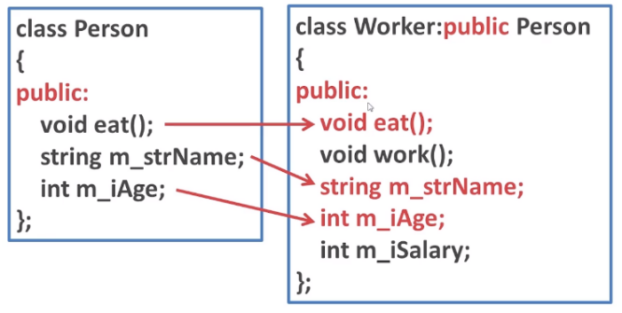
在类的内部（定义类的代码内部），无论成员被声明为public、protected还是private，都是可以互相访问的，没有访问权限的限制。在类的外部（定义类的代码之外），只能通过对象访问成员，并且通过对象只能访问public属性的成员，不能访问private、protected属性的成员。

## 公有继承

class A : public B

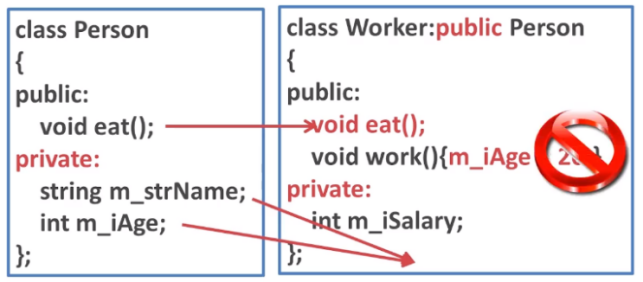


1、public成员变量公有继承



2、private成员变量公有继承

在继承中，private成员变量被继承在子类不可见的位置，而不是private的位置，因此不能在子类中访问父类的private变量。

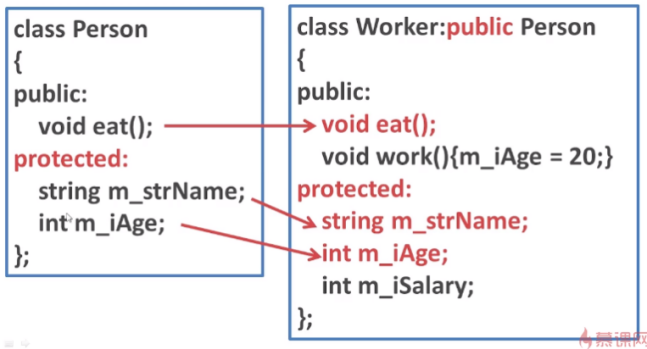


3、protected成员变量继承



说明：当在子类中访问父类protected成员变量时会报错。

在公有继承情况下protected成员变量的访问：



可以通过公有继承的子类的成员函数访问继承来的protected类型的成员变量。

## 私有继承

参考：

<https://blog.csdn.net/yu12345678900/article/details/41383523>

class A : private B



私有继承是has a的关系，即包含关系，子类对象包含一个父类对象，比如线段和坐标。

## 保护继承

class A : protected B

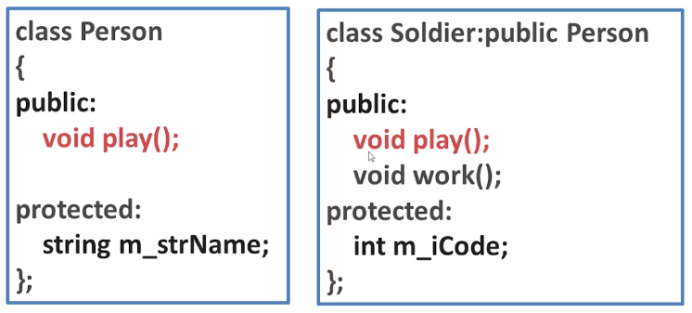


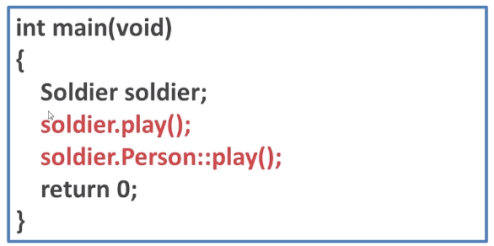
与公有继承不同的是，父类的public成员变量被继承在子类的protected控制符下面。

## 隐藏

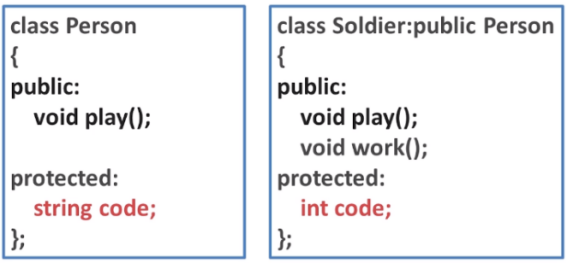
### 同名

如果子类和父类同时定义同一个方法，则其访问方式为：





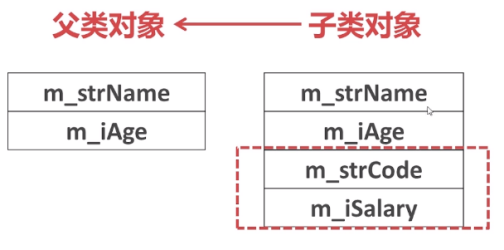
如果子类和父类同时定义了同一个成员变量，则其访问方式为：



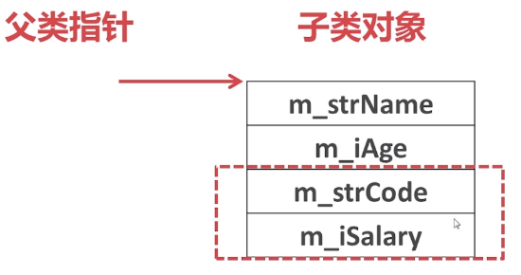


其实针对上述这样的成员变量“重名”的情况，尽量避免使用。也可以采用数据类型不同确定不同名称，比如父类中string类型成员变量命名为m\_scode，子类中int类型成员变量命名为m\_icode。

### is a



子类对象赋值给父类（或者子类初始化父类），其含义就是用子类继承自父类的成员变量给父类中的对应成员变量赋值，父类中不具备子类的成员变量丢失。



如果父类指针访问子类对象，那么它也只能访问到子类中继承自父类的成员变量，而不能访问子类特有的成员变量。

## 多继承/多重继承

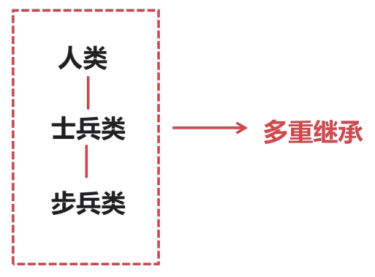
**多重继承：**

多重继承特点总结如下：

1、多重继承与多继承不同，当B类从A类派生，C类从B类派生，此时称为多重继承；

2、当实例化子类时，会首先依次调用所有基类的构造函数，最后调用该子类的构造函数；销毁该子类时，则相反，先调用该子类的析构函数，再依次调用所有基类的析构函数；

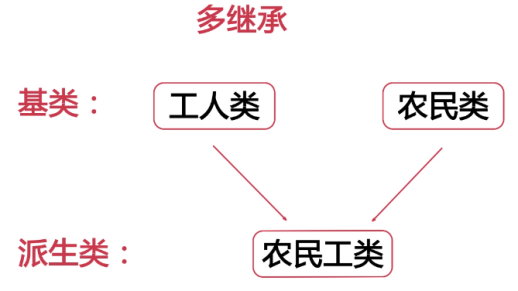
3、无论继承的层级有多少层，只要它们保持着直接或间接的继承关系，那么子类都可以与其直接父类或间接父类构成 is a的关系，并且能够通过父类的指针对直接子类或间接子类进行相应的操作，子类对象可以给直接父类或间接父类的对象或引用赋值或初始化。





**多继承：**

多继承是指一个子类继承多个父类。多继承对父类的个数没有限制，继承方式可以是公共继承、保护继承和私有继承，不写继承方式，默认是private继承。





注：**多重继承会使得继承显得混乱**，所以Java中没有引入。

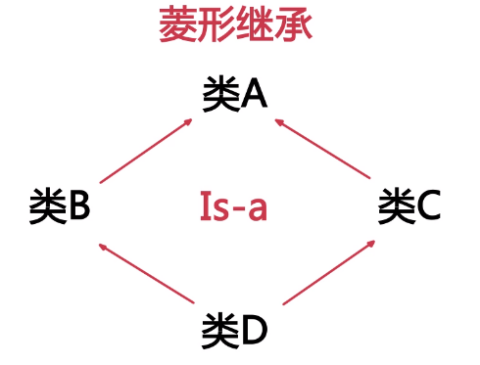
## 虚继承

参考：

<https://blog.csdn.net/bxw1992/article/details/77726390>

### 背景

**虚继承是为了解决多继承和多重继承带来的问题**，比如菱形继承中存在的多个父类的问题：



在类D中存在两个完全一样的类A的副本，这种冗余数据我们不能承担带来的额外的开销，显然不是我们希望的。

类 A 派生出类 B 和类 C，类 D 继承自类 B 和类 C，这个时候类 A 中的成员变量和成员函数继承到类 D 中变成了两份，一份来自 A-->B-->D 这条路径，另一份来自 A-->C-->D 这条路径。

在一个派生类中保留间接基类的多份同名成员，虽然可以在不同的成员变量中分别存放不同的数据，但大多数情况下这是多余的：因为保留多份成员变量不仅占用较多的存储空间，还容易产生命名冲突。假如类 A 有一个成员变量 a，那么在类 D 中直接访问 a 就会产生歧义，编译器不知道它究竟来自 A -->B-->D 这条路径，还是来自 A-->C-->D 这条路径。下面是菱形继承的具体实现：

//间接基类A

class A{

protected:

int m\_a;

};

//直接基类B

class B: public A{

protected:

int m\_b;

};

//直接基类C

class C: public A{

protected:

int m\_c;

};

//派生类D

class D: public B, public C{

public:

void seta(int a){ m\_a = a; } //命名冲突

void setb(int b){ m\_b = b; } //正确

void setc(int c){ m\_c = c; } //正确

void setd(int d){ m\_d = d; } //正确

private:

int m\_d;

};

int main(){

D d;

return 0;

}

这段代码实现了上图所示的菱形继承，第 25 行代码试图直接访问成员变量 m\_a，结果发生了错误，因为类 B 和类 C 中都有成员变量 m\_a（从 A 类继承而来），编译器不知道选用哪一个，所以产生了歧义。

为了消除歧义，我们可以在 m\_a 的前面指明它具体来自哪个类：

void seta(int a){ B::m\_a = a; }

这样表示使用 B 类的 m\_a。当然也可以使用 C 类的：

void seta(int a){ C::m\_a = a; }

### 原理

为了解决多继承时的命名冲突和冗余数据问题，C++提出了虚继承，使得在派生类中只保留一份间接基类的成员。

在继承方式前面加上virtual关键字就是虚继承，请看下面的例子：

//间接基类A

class A{

protected:

int m\_a;

};

//直接基类B

class B: virtual public A{ //虚继承

protected:

int m\_b;

};

//直接基类C

class C: virtual public A{ //虚继承

protected:

int m\_c;

};

//派生类D

class D: public B, public C{

public:

void seta(int a){ m\_a = a; } //正确

void setb(int b){ m\_b = b; } //正确

void setc(int c){ m\_c = c; } //正确

void setd(int d){ m\_d = d; } //正确

private:

int m\_d;

};

int main(){

D d;

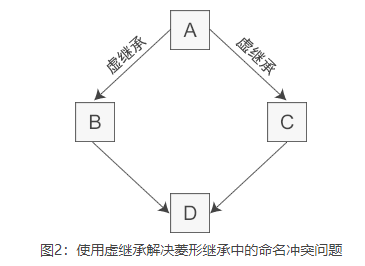
return 0;

}

这段代码使用虚继承重新实现了上图所示的菱形继承，这样在派生类D中就只保留了一份成员变量m\_a，直接访问就不会再有歧义了。

虚继承的目的是让某个类做出声明，承诺愿意共享它的基类。其中，这个被共享的基类就称为虚基类（Virtual Base Class），本例中的A就是一个虚基类。在这种机制下，不论虚基类在继承体系中出现了多少次，在派生类中都只包含一份虚基类的成员。

现在让我们重新梳理一下本例的继承关系，如下图所示：

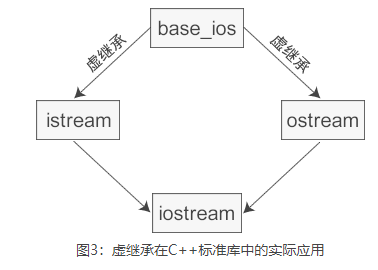


观察这个新的继承体系，我们会发现虚继承的一个不太直观的特征：**必须在虚派生的真实需求出现前就已经完成虚派生的操作。**在上图中，当定义D类时才出现了对虚派生的需求，但是如果B类和C类不是从A类虚派生得到的，那么D类还是会保留A类的两份成员。

换个角度讲，**虚派生只影响从指定了虚基类的派生类中进一步派生出来的类，它不会影响派生类本身。**

在实际开发中，位于中间层次的基类将其继承声明为虚继承一般不会带来什么问题。通常情况下，使用虚继承的类层次是由一个人或者一个项目组一次性设计完成的。对于一个独立开发的类来说，很少需要基类中的某一个类是虚基类，况且新类的开发者也无法改变已经存在的类体系。

C++标准库中的iostream类就是一个虚继承的实际应用案例。iostream从istream和ostream直接继承而来，而istream和ostream又都继承自一个共同的名为base\_ios的类，是典型的菱形继承。此时istream和ostream必须采用虚继承，否则将导致iostream类中保留两份base\_ios类的成员。



虚继承是解决C++多重继承问题的一种手段，从不同途径继承来的同一基类，会在子类中存在多份拷贝。这将存在两个问题：其一，浪费存储空间；第二，存在二义性问题，通常可以将派生类对象的地址赋值给基类对象，实现的具体方式是，将基类指针指向继承类（继承类有基类的拷贝）中的基类对象的地址，但是多重继承可能存在一个基类的多份拷贝，这就出现了二义性。

虚继承可以解决多种继承前面提到的两个问题：

虚继承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。

在这里我们可以对比虚函数的实现原理：他们有相似之处，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）。

虚基类依旧存在继承类中，只占用存储空间；虚函数不占用存储空间。

虚基类表存储的是虚基类相对直接继承类的偏移；而虚函数表存储的是虚函数地址。

### 虚基类成员的可见性

因为在虚继承的最终派生类中只保留了一份虚基类的成员，所以该成员可以被直接访问，不会产生二义性。此外，如果虚基类的成员只被一条派生路径覆盖，那么仍然可以直接访问这个被覆盖的成员。但是如果该成员被两条或多条路径覆盖了，那就不能直接访问了，此时必须指明该成员属于哪个类。

以图2中的菱形继承为例，假设 A 定义了一个名为 x 的成员变量，当我们在 D 中直接访问 x 时，会有三种可能性：

如果 B 和 C 中都没有 x 的定义，那么 x 将被解析为 A 的成员，此时不存在二义性。

如果 B 或 C 其中的一个类定义了 x，也不会有二义性，派生类的 x 比虚基类的 x 优先级更高。

如果 B 和 C 中都定义了 x，那么直接访问 x 将产生二义性问题。

### 方式

虚继承方式：class A : virtual public B

# 多态性

多态的作用：

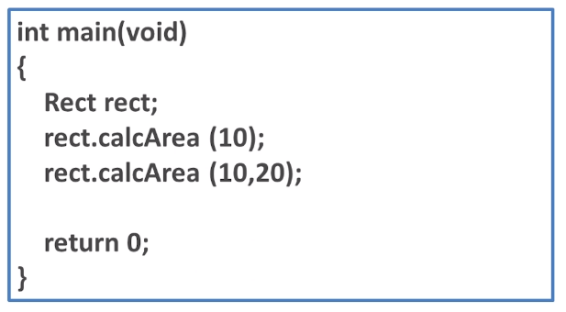
1. 隐藏细节：使得代码能够模块化，拓展代码模块，实现代码重用
2. 接口重用：为了类在继承和派生的时候，保证使用家族中任一类的实例的某一属性时的正确调用。

## 静态多态

**静态多态**又叫**早绑定**，在编译阶段就确定使用哪个函数了，比如重载函数，只要在定义的时候设置了参数就可以知道执行哪一个函数了。

静态函数在编译的时候就已经确定运行时机，虚函数在运行的时候动态绑定。虚函数因为用了虚函数表机制，调用的时候会增加一次内存开销。





### 为什么要重载

函数重载在一定程序上可以减轻程序员**起名字的负担**。最常见的一个例子就是构造函数的重载。

class Test

{

public:

Test(void); // 无参构造函数

Test(int a);//构造函数

Test(int a,int b);//两个整型参数的构造函数

};

可以看到，类Test的三个构造函数名都为Test。如果没有重载，要实现三个构造函数就可能需要三个不同的构造函数名区分，这也就增加了类的使用者的负担，使用者需要传入不同参数构造对象时，就需要使用不同的构造函数名称。而有函数重载之后，类的使用者可以使用同一个函数名传入不同的参数即可。

当然了，如果单纯地为了减轻起名字的负担而去使用函数重载，而使得函数失去了本来的信息，则是一个不明智的选择。我们可以为那些操作确实极其相似的函数进行重载。

### 不能重载的情况

以下几种情况下，是不能重载或者说是非法的。

**main函数不能重载**

这是在C++ 11标准中说明的：

A program shall contain a global function called main, which is the designated start of the program....

This function shall not be overloaded.

试想如果作为用户程序入口函数的main函数被重载了，那么加载的时候该以哪个为入口呢？

**只有返回值不同**

例如下面两个声明只有返回值不同，函数名和形参都相同：

double calcArea(const Square&);

int calcArea(const Square&); //非法，仅有返回值不同，不可重载

/\*以上声明同时出现会报错\*/

试想一下，当你传入Square类型参数，而不去使用返回值时，应该调用上面的哪个函数呢？

**形参列表看似不同，实则相同**

例如使用typedef给Triangle起了一个“别名”：

typedef Triangle MyTri;

double calcArea(const Triangle&);

double calcArea(const MyTri&);

/\*以上声明同时出现会报错\*/

上面这种情况的形参看似不一样，本质上来说它们并没有什么不同。

**形参名不同**

例如：

double calcArea(const Circle &circle );//形参名为circle

double calcArea(const Circle& cir);//形参名为cir

double calcArea(const Circle& );//省略形参名

/\*以上声明同时出现会报错\*/

这里形参的名字仅仅是起到说明或者记忆的作用，因此对于上面三个声明，它们的形参名可以随意起，但不会影响形参列表的内容。

**仅有顶层const的差异**

例如：

double calcArea(const Circle);//函数1

double calcArea(Circle);//重复声明了函数1

/\*以上声明同时出现会报错\*/

double calcArea(Circle\* const);//函数2

double calcArea(Circle\*);//重复声明了函数2

/\*以上声明同时出现会报错\*/

但需要特别注意的是，**如果形参是指针或引用，是可以通过区分指向到底是常量对象还是非常量对象来实现函数重载**。例如下面的情况是可以实现函数重载的：

double calcArea(const Circle&);//作用于常量引用

double calcArea(Circle&);

/\*以上声明同时出现不会报错\*/

double calcArea(const Circle\*);//作用于常量指针

double calcArea(Circle\*);

/\*以上声明同时出现不会报错\*/

### 重载与重写

参考：

<https://www.cnblogs.com/upcwanghaibo/p/6527354.html>

重载：两个函数名相同，但是参数列表不同（个数，类型），返回值类型没有要求，在同一作用域中。

重写：子类继承了父类，父类中的函数是虚函数，在子类中重新定义了这个虚函数，这种情况是重写。

### CRTP

简单来说，CRTP有两大特性：

1、继承自模板类。

2、派生类将自身作为参数传给模板类。

翻译成代码，这样：

// 我们先定义一个模板类作为基类

template <typename T>class Base

{

...

};

// 定义一个派生类，这个类继承以自身作为参数的基类

class Derived : public Base<Derived>

{

...

};

问题来了，为什么要这样做呢？

这样做的目的其实很明确，从基类对象的角度来看，派生类对象其实就是本身，这样的话只需要用一个static\_cast就可以把基类转化成派生类，从而实现基类对象对派生对象的访问。

我们还是来看看代码：

template <typename T>class Base{

public:

void doSomething()

{

T& derived = static\_cast<T&>(\*this);

}

};

class Derived : public Base<Derived>{

public:

void doSomething()

{

std::cout << " Derived class " << std::endl;

}

};

很多小伙伴可能发现了，这里将基类转换成派生类用的是static\_cast静态绑定，而普通基类转派生类用的是dynamic\_cast动态绑定。动态绑定的目的是为了确保你所转化的派生类是正确的，而对于CRTP来说，基类是继承于模板类的参数，也就是派生类本身。这也正是CRTP这种设计的目的。

**CRTP的优点是什么呢？**

多态是个很好的特性，但是动态绑定比较慢，因为要查虚函数表。而使用 CRTP，完全消除了动态绑定，降低了继承带来的虚函数表查询开销。

### 总结

在定义了重载函数后，我们需要以合理的实参进行调用。大多数情况下，我们很容易判断传入的对应实参需要调用哪个函数，但是有些时候却并不那么容易。我们将会在后面的文章中看到如何进行函数匹配。

函数重载能够减轻程序员命名的负担，但这不应该以丢失可读性为代价。

main函数不能重载。

重载函数的形参在数量或者类型上要有不同。

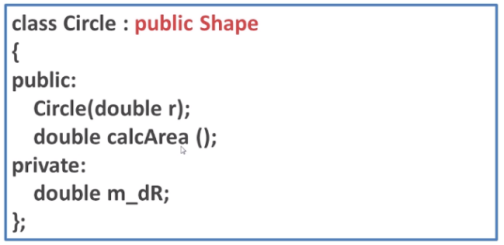
不能以返回值作为函数重载要素。

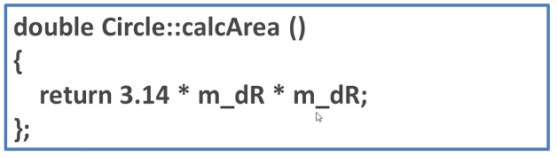
C语言没有函数重载。

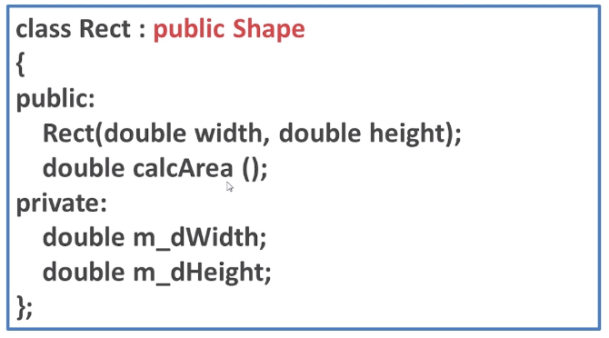
## 动态多态

**动态多态**又叫**晚绑定**，可以理解为对不同的对象下发相同的指令，但是却执行不同的操作，即在真正运行的时候确定执行的函数。动态多态的前提是必须以封装和继承为前提才可以。









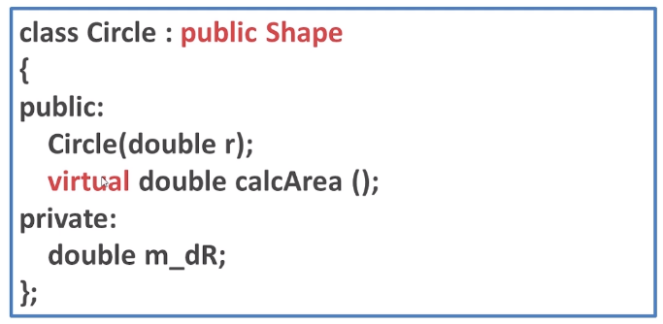


如果采用如下的调用方式，此时调用的都是父类的计算面积，即打印两行“cal area”，而不是调用子类具体的计算函数，这个结果不是我们想要的：

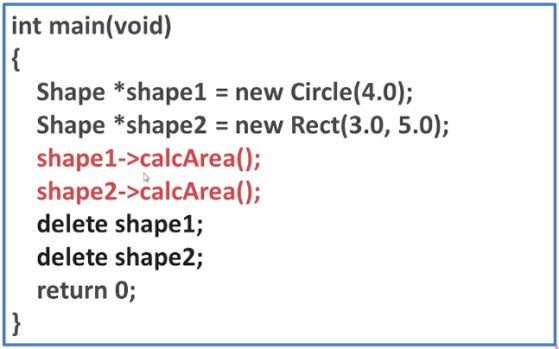


如果想要实现动态多态，需要使用virtual虚函数，对函数做如下修改：





在定义子类的时候继承自父类的virtual函数加一个virtual，如果不加默认会添加virtual，但是还是推荐显式增加virtual关键字。



## 虚函数

参考：

<https://www.cnblogs.com/sunada2005/archive/2013/10/30/3397423.html>

<https://www.cnblogs.com/vipchenwei/p/7466018.html?utm_source=debugrun&utm_medium=referral>

多态的实现主要分为静态多态和动态多态，静态多态主要是重载，在编译的时候就已经确定；动态多态是用虚函数机制实现的，在运行期间动态绑定。

举个例子：一个父类类型的指针指向一个子类对象时候，使用父类的指针去调用子类中重写了的父类中的虚函数的时候，会调用子类重写过后的函数，在父类中声明为加了virtual关键字的函数，在子类中重写时候不需要加virtual也是虚函数。

虚函数的实现：在有虚函数的类中，类的最开始部分是一个虚函数表的指针，这个指针指向一个虚函数表，表中放了虚函数的地址，实际的虚函数在代码段(.text)中。当子类继承了父类的时候也会继承其虚函数表，当子类重写父类中虚函数时候，会将其继承到的虚函数表中的地址替换为重新写的函数地址。使用了虚函数，会增加访问内存开销，降低效率。

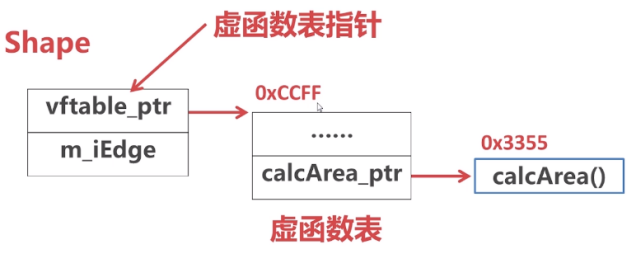
**静态函数与虚函数**

静态函数在编译的时候就已经确定运行时机，虚函数在运行的时候动态绑定。虚函数因为用了虚函数表机制，调用的时候会增加一次内存开销。

### 原理

对象指针：指向对象的指针；

函数指针：指向函数的指针。



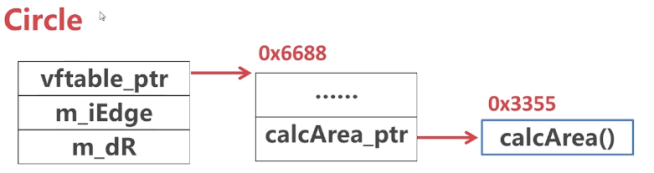
**虚函数表是编译时确定的，运行时会查找该表**。各个虚拟函数这时被组织成了一个虚拟函数的入口地址的数组，而**对象的隐藏成员—虚拟函数表指针是在运行期，也就是构造函数被调用时进行初始化的，这就是实现多态的关键**。

**子类若重写父类虚函数，虚函数表中，该函数的地址会被替换，对于存在虚函数的类的对象，在VS中，对象的对象模型的头部存放指向虚函数表的指针，通过该机制实现多态**。

注：当实例化对象的时候，除了必要的成员变量，还会有一个虚函数表指针的成员变量vftable\_ptr（指针，4字节），它指向虚函数表，**父类的虚函数表只有一个**，**通过父类实例化的子类虚函数表指针vftable\_ptr都是一样的**，以确保每一个子类都指向自己的虚函数表（通过偏移量），在虚函数表中找到对应函数的入口地址。

**注：如何找到虚函数表？即通过虚函数表指针的成员变量vftable\_ptr。**

当实例化Circle的时候，它本身没有虚函数，但是它继承自Shape，故子类Circle也会产生一个虚函数表，但是这个虚函数表中继承自父类的函数的函数入口地址也被继承来了，所以指向的是父类的那个函数：



如果子类覆盖父类的虚函数：



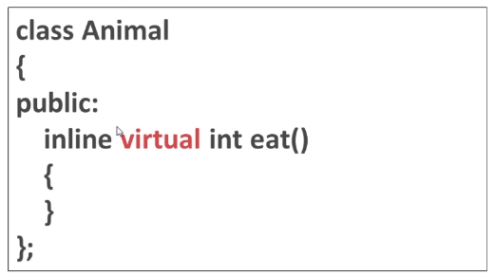
### 限制

1. 不能修饰普通函数（必须是某一类的成员函数，不能是全局函数）；
2. 不能修饰静态成员函数



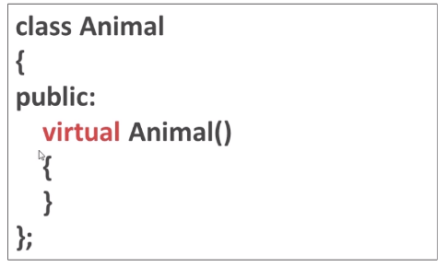
因为static的成员函数不属于任何一个对象，它与类共生存的，所以使用virtual修饰会造成编译报错。

1. 不能修饰内联函数：



如果修饰内联函数，计算机会忽略inline关键字，而使其变为虚函数。

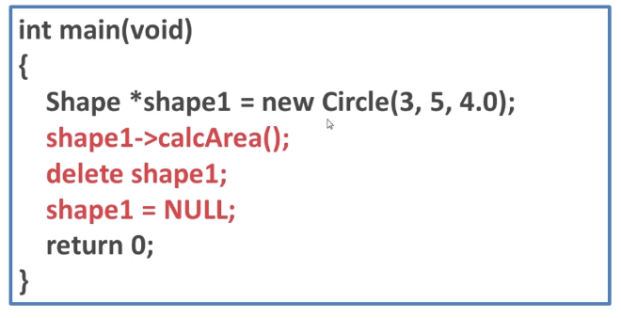
1. 不能修饰构造函数：



因为构造函数专属于本类，负责本类对象的构造，所以一个类不可能用另一个类的构造函数来构造本类的对象，也就不可能去重定义一个他类的构造函数，也就没有必要使之成为virtual函数。

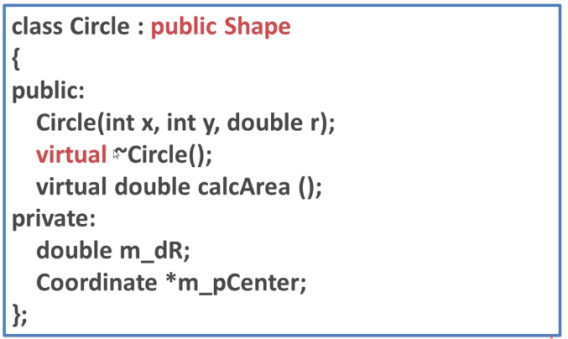
并且子类对象中父类成分的构造并不与继承机制相同。无论是动态绑定还是静态绑定父类的构造函数总是先被调用，因而可以说构造函数的调用次序是与绑定方式无关的，不存在次序的变更。也可以说它是自动的。

### 虚析构函数

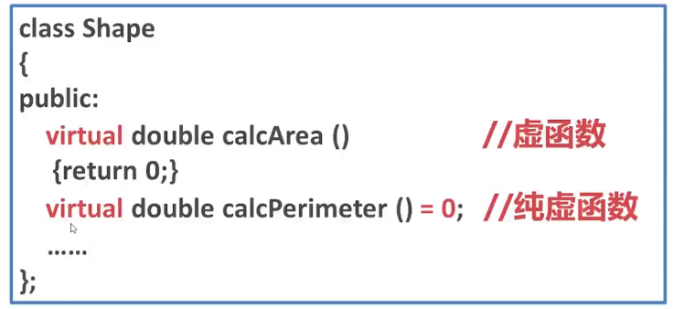


在上面的例子中，调用虚函数是正确的，执行结果与预期一致，但是在delete内存释放的时候就会有问题，这里释放的是父类的指针，则调用的是父类的析构函数，并不会调用子类Circle的析构函数，那么就造成内存泄露了。因此引入了虚析构函数，即用virtual修饰析构函数，此时就不会造成内存泄露。





## 纯虚函数





注：如果是虚函数，则在父类初始化后的虚函数表中存储的是一个有意义的地址，这个可以供子类使用，但是如果是纯虚函数，则这个地址是0，无意义，只能等到子类实例化之后赋值才有意义。

## 抽象类

包含纯虚函数的类叫做抽象类，抽象类的子类也可能是抽象类。

抽象类无法实例化对象。

抽象类的子类也可能是抽象类。

## 接口类

仅含有纯虚函数的类叫做接口类。

## RTTI

RTTI（Run-Time Type Identification），即运行时类型识别。

运行时类型检查，在C++层面主要体现在dynamic\_cast和typeid，VS中虚函数表的-1位置存放了指向type\_info的指针。对于存在虚函数的类型，typeid和dynamic\_cast都会去查询type\_info。

Typeid注意事项：

1. type\_id返回一个type\_info对象的引用；
2. 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；
3. 只能获取对象的实际类型。

dynamic\_cast注意事项：

1. 只能应用于指针和引用的转换；
2. 要转换的类型中必须包含虚函数；
3. 转换成功返回子类的地址，失败返回NULL。

## 异常处理

# 模板编程

C++的模板编程体现在函数上就是模板函数，体现在类上就是模板类，这些都可以用标准模板类表示。

## 友元函数

### 友元全局函数

函数定义在全局的，且是友元函数。

函数声明如下：



友元函数的参数传入一个对象的引用或者指针，目的得能够通过这个参数访问到类中的私有或者保护的成员变量/函数，这样才能体现出友元函数的优势。

使用方法如下：



传入引用或指针执行效率更高，因此推荐这种入参，如果直接传入对象（传值）则效率较低。这里是使用类的对象直接访问私有数据，因为这个方法已经声明为friend类型的了。

### 友元成员函数

函数定义在类中，并且声明为另外一个类的友元函数，叫做友元成员函数。

函数声明如下：



这里的友元函数printXY不再是类Coordinate的成员函数，而是Circle类的成员函数。

使用方法如下：



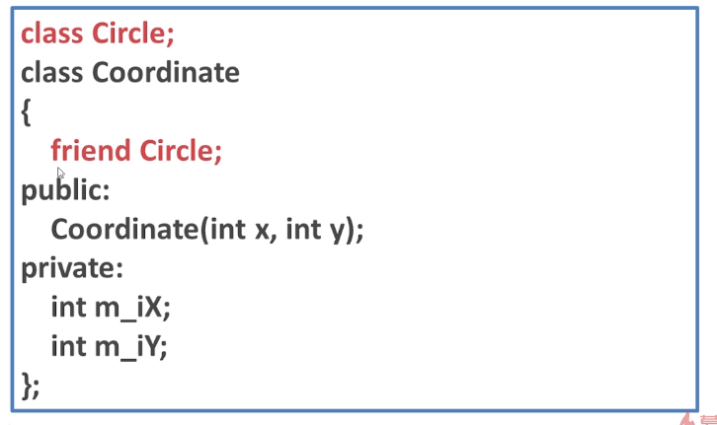
友元函数可以方便地访问另外一个类的私有数据，这是友元函数的优势；但是，友元函数破坏了封装性，出问题时比较难排查。一般情况下，少使用友元函数。

注：友元函数位置没有特殊要求，可以放在public、private、protected访问控制符下面，但是一般情况下我们放置在最前面，这样更加明晰。

## 友元类

友元只是封装的补充，并不是一种很好地语法，是不得已而为之。如果前期设计的好，完全可以避免使用友元。

友元类的声明如下：



在声明友元类之前，先使用class声明一下类，这里是告诉计算机存在Circle类，这里只是使用一下，后面会实现的，放置编译报错。

使用方法如下：



注意：

1、友元关系不可传递；

2、单向性；

3、友元声明的形式和数量不受限制。

## static

### 全局静态变量

在全局变量前加上关键字static，全局变量就定义成一个全局静态变量。

生命周期：**静态存储区，在整个程序运行期间一直存在**。

初始化：**未经初始化的全局静态变量会被自动初始化为0**（自动对象的值是任意的，除非他被显式初始化）。

作用域：**全局静态变量在声明他的文件之外是不可见的**，准确地说是从定义之处开始，到文件结尾。

### 局部静态变量

在局部变量之前加上关键字static，局部变量就成为一个局部静态变量。

内存中的位置：静态存储区

初始化：未经初始化的全局静态变量会被自动初始化为0（自动对象的值是任意的，除非他被显式初始化）。

作用域：作用域仍为局部作用域，当定义它的函数或者语句块结束的时候，作用域结束。但是当局部静态变量离开作用域后，并没有销毁，而是仍然驻留在内存当中，只不过我们不能再对它进行访问，直到该函数再次被调用，并且值不变。

### 静态函数

在函数返回类型前加static，函数就定义为静态函数。函数的定义和声明在默认情况下都是extern的，但静态函数只是在声明他的文件当中可见，不能被其他文件所用。

函数的实现使用static修饰，那么这个函数只可在本cpp内使用，不会同其他cpp中的同名函数引起冲突；

warning：不要再头文件中声明static的全局函数，不要在cpp内声明非static的全局函数，如果你要在多个cpp中复用该函数，就把它的声明提到头文件里去，否则cpp内部声明需加上static修饰；

使用关键字static声明静态成员和函数：



### 静态数据成员

**静态数据成员不依赖于对象存在，它不会在对象的构造函数中初始化，而是在构造函数前就存在，其声明周期比对象的久，存在于程序运行中**。

在类中，静态成员可以实现多个对象之间的数据共享，并且使用静态数据成员还不会破坏隐藏的原则，即保证了安全性。因此，静态成员是类的所有对象中共享的成员，而不是某个对象的成员。对多个对象来说，静态数据成员只存储一处，供所有对象共用。

静态成员函数依赖于类存在，不依赖于对象，而普通成员函数依赖于对象存在。

初始化方法：

类型 类名::变量名 = 变量值

访问方法：

一种是直接通过类名::变量名，一种是通过类对象访问（普通数据成员也采用该方法）。



注意：

1、静态数据成员必须单独的初始化，因为它不随着对象的产生而产生（因为构造函数析构函数不适用于静态数据成员）；

2、静态成员函数不能调用非静态成员函数和非静态数据成员；

3、静态数据成员只有一份，且不依赖对象而存在（如果用sizeof求对象大小，一定是不包含静态数据成员的）。

### 静态成员函数

静态成员函数和静态数据成员一样，它们都属于类的静态成员，它们都不是对象成员。因此，对静态成员的引用不需要用对象名。

在静态成员函数的实现中不能直接引用类中说明的非静态成员，可以引用类中说明的静态成员（这点非常重要）。如果静态成员函数中要引用非静态成员时，可通过对象来引用。从中可看出，调用静态成员函数使用如下格式：<类名>::<静态成员函数名>(<参数表>)。

## 运算符重载

C++中字符拼接“+”以及cout都是运算符重载：



运算符重载的本质就是函数重载。

关键字：operator

### 一元运算符

比如：-（负号）的重载（友元函数重载/成员函数重载），++运算符重载

成员函数重载：





使用：



友元函数重载：





### 二元运算符

## 函数模板

模板技术就是类型参数化，编写代码可以忽略类型。为了让编译器区分是普通函数还是模板函数，采用关键字template区分。

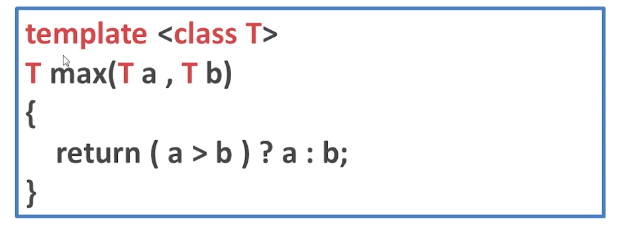
### template typemane

参考：<https://blog.csdn.net/lyn631579741/article/details/110730145>

<https://www.cnblogs.com/weiyouqing/p/10538892.html>

函数模板实现原理：函数模板-->模板函数-->被调用

关键字：template typemane class



"typename"是一个C++程序设计语言中的关键字。当用于泛型编程时是另一术语"class"的同义词。这个关键字用于指出模板声明（或定义）中的非独立名称（dependent names）是类型名，而非变量名。

注：typename<class T>和template <typename T>可以混用。

函数模板与普通函数的区别：

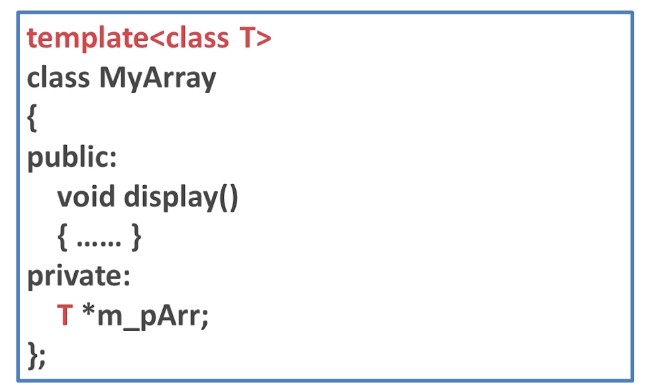
1. 函数模板不允许自动类型转换
2. 普通函数能够自动进行类型转化

函数模板和普通函数在一起调用规则：

1. 函数模板可以像普通函数那样被重载
2. C++编译器优先考虑普通函数
3. 如果函数模板可以产生一个更好的匹配，那么选择模板
4. 可以通过空模板实参列表的语法限定编译器只能通过模板匹配

## 类模板

### template class



注：模板代码不能分离编译。

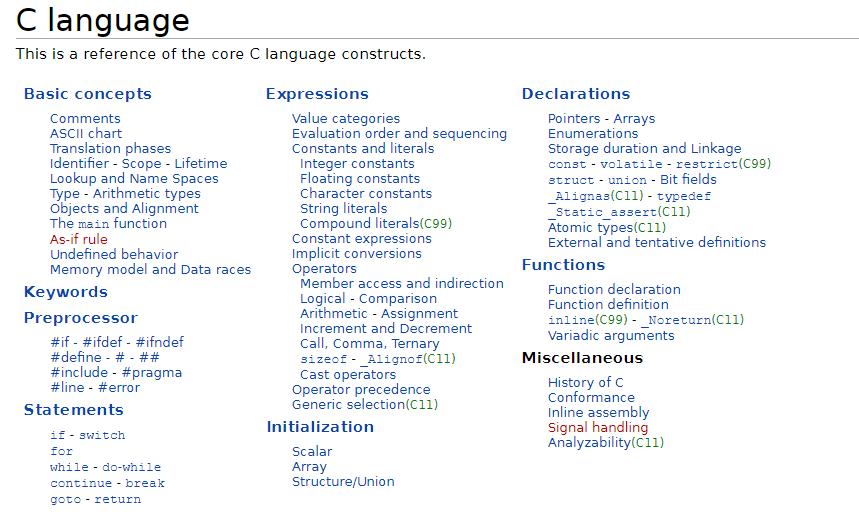
## 标准模板类

# C++与C

可以简单理解为C++是C的超集（当然，也有C的部分特性是C++不支持的），C语言较为底层，在嵌入式、操作系统等领域使用，C++则是面向对象的编程语言，引入了classes、templates、exceptions 等很多新特性，二者应用领域不同，不存在孰优孰劣。

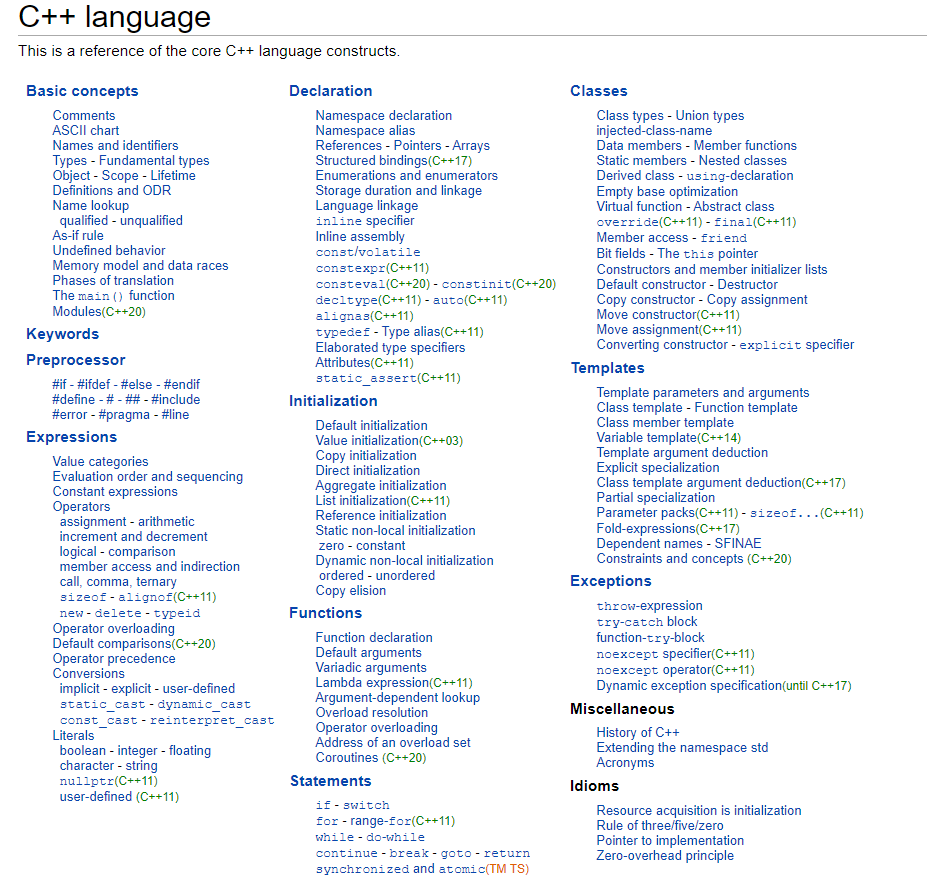
C语言知识图谱：

<https://en.cppreference.com/w/c/language>



C++语言知识图谱：

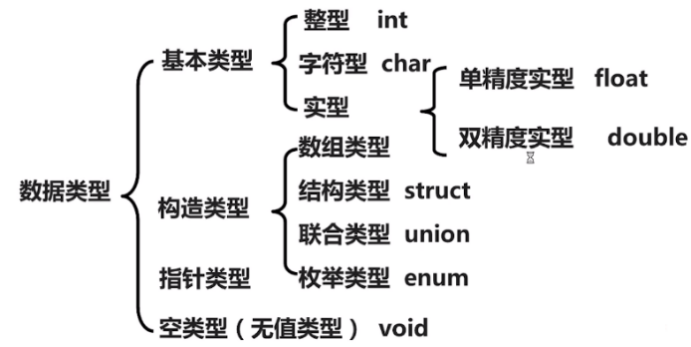
<https://en.cppreference.com/w/cpp/language>



除了上述图谱显示的诸多不同之处，下面介绍平时常用但是容易忽略的一些小区别：

## 数据类型

C语言中基本数据类型：



C/C++中的结构体struct：

C语言struct仅仅是对数据类型的封装，不能封装函数指针，C++可以。

封装性：C语言struct默认是public访问，不具备保护权限，C++中默认是pricate。

C++新增bool类型变量：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 逻辑类型 | 真 | 假 |
| C | 没提供 | 非0 | 0 |
| C++ | bool | true | false |

C语言的缺点：

1. C风格字符串（C语言没有字符串类型，使用\0作为结束符的字符数组表示字符串）：容易导致程序崩溃或执行恶意代码（很容易导致越界读写非法内存，如果写内存无效，则导致崩溃，如果非法读是有效的，则会出现不可预料的后果，如果非法写有效，则会植入恶意代码）
2. 动态内存管理：容易导致内存溢出，程序和系统僵死
3. C语言语法限制不严格，各种类型可以强制转换

## 初始化方法

C语言提供的初始化方法：int x = 1024;

C++提供两种初始化方法：

1. 复制初始化

int x = 1024;

1. 直接初始化

int x(1024);

## 随用随定义

C 语言：所有变量定义必须位于函数体的最前面

int main()

{

int v1 = 1;

int v2 = 2;

v1 = v1 + 2;

v2 = v2 + v1;

return 0;​

}

C++：所有变量随用随定义

int main()

{

int v1 = 1;

v1 = v1 + 2;

int v2 = 2;

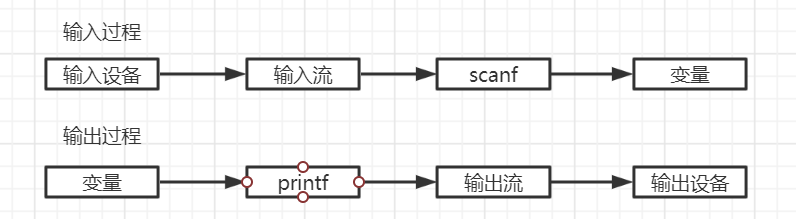
v2 = v2 + v1;

return 0;

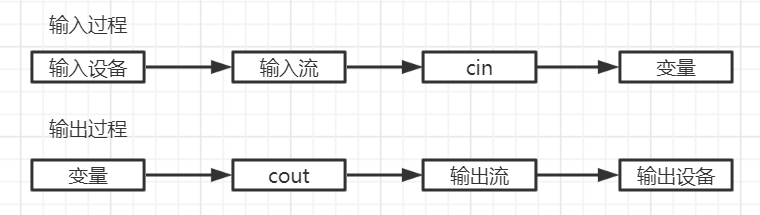
}

## 输入输出

C语言I/O方式：



C++语言I/O方式：



注：C++这种输入输出方式的好处：

1. 不用关注占位符
2. 不用关注数据类型
3. 不易出现问题

## namespace

namespace A

{

int x = 0;

void func1();

void func2();

}

namespace B

{

int x = 1;

void func1();

void func3();

}

cout << A::x << endl;

​B::func1();​



# C++11

## 演进

**C++演进：**

C++98（1.0）

C++03（TR1，TTechnical Report 1）

C++11（2.0）

C++14

**新特性：**

C++2.0新特性包括语言和标准库两个层面，标准库以header files头文件形式呈现。

语言：

Variadic Templates

Move Semantics

Auto

Range-base for loop

Initializer list

Lambda

标准库：

Type\_traits

Unordered容器

Forward\_list

Array

Tuple

Con-currency

RegEx

在测试C++11新特性前，需要确认支持C++11：macro \_\_cplusplus

如果是支持C++11，则\_\_cplusplus定义为：

#define \_\_cplusplus 2011031

否则如果是C++98或C++03：

#define \_\_cplusplus 1997111

注：可以直接cout<<\_\_cplusplus根据输出结果，查看是否支持新特性。

## 兼容性

## 内存模型

### 内存乱序

说到内存模型，首先需要明确一个普遍存在，但却未必人人都注意到的事实：程序通常并不是总按着照源码中的顺序一一执行，此谓之乱序，乱序产生的原因可能有好几种：

1、编译器出于优化的目的，在编译阶段将源码的顺序进行交换。

2、程序执行期间，指令流水被cpu乱序执行。

3、inherent cache的分层及刷新策略使得有时候某些写读操作的从效果上看，顺序被重排。

以上乱序现象虽然来源不同，但从源码的角度，对上层应用程序来说，他们的效果其实相同：写出来的代码与最后被执行的代码是不一致的。这个事实可能会让人很惊讶：有这样严重的问题，还怎么写得出正确的代码？这担忧是多余的了，乱序的现象虽然普遍存在，但它们都有很重要的一个共同点：在单线程执行的情况下，乱序执行与不乱序执行，最后都会得出相同的结果 (both end up with the same observable result), 这是乱序被允许出现所需要遵循的首要原则，也是为什么乱序虽然一直存在但却多数程序员大部分时间都感觉不到的根本原因。

乱序的出现说到底是编译器，CPU等为了让你程序跑得更快而作出无限努力的结果。

从乱序的种类来看，乱序主要可以分为如下4种：

1、写写乱序(store store), 前面的写操作被放到了后面的操作之后，比如：

a = 3;

b = 4;

被乱序为：

b = 4;

a = 3;

2、写读乱序(store load)，前面的写操作被放到了后面的读操作之后，比如：

a = 3;

load(b);

被乱序为

load(b);

a = 3;

3、读读乱序(load load)， 前面的读操作被放到了后一个读操作之后，比如：

load(a);

load(b);

被乱序为：

load(b);

load(a);

4、读写乱序(load store), 前面的读操作被放到了后一个写操作之后，比如：

load(a);

b = 4;

被乱序为：

b = 4;

load(a);

程序的乱序在单线程的世界里多数时候并没有引起太多引人注意的问题，但在多线程的世界里，这些乱序就制造了特别的麻烦，究其原因，最主要的有2个：

1、并发不能保证修改和访问共享变量的操作原子性，使得一些中间状态暴露了出去，因此像mutex，各种lock之类的东西在写多线程时被频繁地使用。

2、变量被修改后，该修改未必能被另一个线程及时观察到，因此需要“同步”。

解决同步问题就需要确定内存模型，也就是需要确定线程间应该怎么通过共享内存来进行交互。

### 概述

### 分类

一般来说，内存模型可分为静态内存模型和动态内存模型。

#### 静态内存模型

1. 静态内存模型主要是类（或结构）对象在内存中的布局。也就是类（或结构）成员在内存中是如何存放的。

类（或结构）对象的内存布局请参考：

Stanley B.Lippman的《深度探索C++对象模型》。

#### 动态内存模型

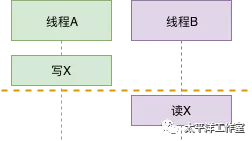
2、动态内存模型是从行为方面来看，多个线程对同一个对象同时读写时所做的约束，该模型理解起来要复杂一些，涉及了内存、Cache、CPU各个层次的交互，尤其是在多核系统中为了保证多线程环境下执行的正确性，需要对读写事件加以严格限制。std::memory\_order就是这用来做这事的，它实际上是程序员、编译器以及CPU之间的契约，遵守契约后大家各自优化，从而可能提高程序性能。

### C++11内存模型

在正式介绍memory\_order之前，我们先来看两个概念：synchronized-with和happends-before。

#### synchronized-with

行为：synchronized-with。  
 这是std::atomic生效的前提之一。假设X是一个atomic变量。如果线程A写了变量X, 线程B读了变量X，那么我们就说线程A、B间存在synchronized-with关系。C++11默认的原子操作(memory\_order\_seq\_cst)就是synchronized-with的，保证了对X的读和写是互斥的，不会同时发生。



#### happends-before

结果：happens-before。  
 happens-before指明了后发生的动作会看到先发生的动作的结果。还是上图，当线程B读取X时，读到的一定是写入后的X值，而不会是其它情况。happends-before具有传递性。如果A happens-before B，B happens-before C，那么A happends-before C。

概念介绍完了，下面来看看C++11为std::atomic提供的memory order:

#### memory order

在C++11中，设计者给出的解决方式是让程序员来指定内存顺序，共计6种：



按照原子操作的类型，可分为三种,每种可用的类型如下：

1、存储操作(store)

relaxed

release

seq\_cst

2、读取操作(load)

relaxed

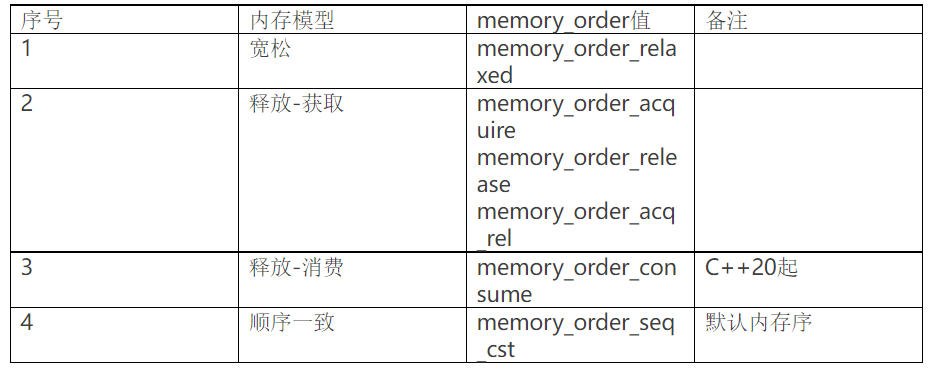
consume

acquire

seq\_cst

3、RMW操作(read-modify-write)

all



这些不同的内存序模型在不同的CPU架构下会有不同的代价。这允许专家通过采用更合理的内存序获得更大的性能升；同时允许在对性能要求不是那么严格的环境中采用默认的内存序，使得程序更容易理解。

##### 宽松次序(relaxed ordering)

在原子变量上采用relaxed ordering的操作不参与synchronized-with关系，无同步操作，它们不会在内存并发访问时强加顺序。Relaxed order只是保证了原子性和修改顺序的一致性。在同一线程内对同一原子变量的操作不可以被重排，仍保持happens-before关系，但这与别的线程无关。尽管如此，relaxed ordering操作仍然是原子的，其值不会因为多线程而被破坏。先看看一个简单的例子：

std::atomic<int> x{0}, y{0};  
 void thread\_1() {  
 auto r1 = y.load(std::memory\_order\_relaxed);  // A  
 x.store(r1, std::memory\_order\_relaxed);  // B  
 }  
 void thread\_2() {  
 auto r2 = x.load(std::memory\_order\_relaxed); // C  
 y.store(42, std::memory\_order\_relaxed);  // D  
 }

执行完上面的程序，可能出现r1 == r2 == 42。理解这一点并不难，因为编译器允许调整C和D的执行顺序。如果程序的执行顺序是 D -> A -> B -> C，那么就会出现r1 == r2 == 42。

Relaxed ordering适用于只要求原子操作，不需要其它同步保障的情况。该操作典型的应用场景是程序计数器：

#include <cassert>  
 #include <thread>  
 #include <atomic>  
 std::atomic<int> cnt{0};  
 void f()  
 {  
 for (int n = 0; n < 1000; ++n) {  
 cnt.fetch\_add(1, std::memory\_order\_relaxed);  
 }  
 }  
 int main()  
 {  
 std::thread threads[10];  
 for (std::thread &thr: threads) {  
 thr = std::thread(f);  
 }  
 for (auto &thr : v) {  
 thr.join();  
 }  
 assert(cnt == 10000); // 永远不会失败  
 return 0;  
 }

##### 释放-获取次序(release-acquire ordering)

Release-acquire中没有全序关系，但它提供了一些同步方法。在这种序列模型下，原子操作对应的内存序为：

  
 Release-acquire有一个特点：线程A中所有发生在release x之前的写操作(包括非原子或宽松原子)，对在线程B acquire x之后都可见。本来A、B间读写操作顺序不定。这么一同步，在x这个点前后，A、B线程之间有了个顺序关系，称作inter-thread happens-before。

Release-acquire中同步是成对出现的，仅建立在释放和获取同一原子对象的线程之间。其它线程有可能看到不一样的内存访问顺序。在我们常用的x86系统(强顺序系统)上，释放-获取顺序对于多数操作是自动进行的，无需为此同步模式添加额外的CPU指令。但在弱顺序系统(如ARM)上，必须使用特别的CPU加载或内存栅栏指令。

一个释放-获取同步的例子是std::mutex：线程A释放锁而线程B获得它时，发生于线程A环境的临界区(释放之前)中的所有内存写入操作，对于线程B(获得之后)均可见。下面我们来看一个释放-获取的例子：

#include <thread>  
 #include <atomic>  
 #include <cassert>  
 #include <string>  
 std::atomic<std::string\*> ptr;  
 int data;  
 void producer()  
 {  
 std::string\* p  = new std::string("Hello");  
 data = 42;  
 // 这句要放在最后，目的是为了在consumer中看到data的副带效应  
 ptr.store(p, std::memory\_order\_release);  
 }  
   
 void consumer()  
 {  
 std::string\* p2;  
 while (!(p2 = ptr.load(std::memory\_order\_acquire))) {  
 ;  
 }  
 // 下面的两个断言永远为真  
 assert(\*p2 == "Hello");  
 // producer中执行store之前的操作，在这里也可以看到  
 assert(data == 42);  
 }  
  
 int main()  
 {  
 std::thread t1(producer);  
 std::thread t2(consumer);  
 t1.join();  
 t2.join();  
 }

根据happens-before，可以知道释放-获取次序是可以传递的，在更多线程下仍然有效。

##### 释放-消费次序(release-consume ordering)

释放-消费顺序的规范正在修订中，C++标准暂不鼓励使用memory\_order\_consume。

##### 顺序一致性次序(sequential-consisten ordering)

顺序一致性原子操作可以看作是释放-获取操作的加强版，它与释放-获取顺序相同的方式排序内存(在一个线程中先发生于存储的任何结果都变成进行加载的线程中的可见副效应)的同时，还对所有内存操作建立单独全序。

换个说法。Release-acquire只针对一个变量x的原子操作进行同步，而Sequential-consisten则是对所有使用memory\_order\_seq\_cst的原子操作进行同步。这么一来所有的使用memory\_order\_seq\_cst的原子操作就跟由一个线程顺序执行似的。

顺序一致性次序是std::atomic的默认内存序，它意味着将程序看做是一个简单的序列。如果对于一个原子变量的所有操作都是顺序一致的，那么多线程程序的行为就像是这些操作都以一种特定顺序被单线程程序执行。从同步的角度来看，一个顺序一致的store操作会与load操作同步。顺序模型还保证了在load之后执行的顺序一致原子操作都得表现得在store之后完成。

顺序一致性模型在所有多核系统上要求完全的内存栅栏CPU指令。这可能成为性能瓶颈，因为它强制受影响的内存访问传播到每个核心。

此示例演示顺序一致次序为必要的场合。任何其他次序都可能触发assert，因为可能令线程c和d观测到原子对象x和y以相反顺序更改。

#include <thread>  
 #include <atomic>  
 #include <cassert>  
 std::atomic<bool> x = {false};  
 std::atomic<bool> y = {false};  
 std::atomic<int> z = {0};  
  
 void write\_x()  
 {  
 x.store(true, std::memory\_order\_seq\_cst);  
 }

void write\_y()  
 {  
 y.store(true, std::memory\_order\_seq\_cst);  
 }  
  
 void read\_x\_then\_y()  
 {  
 while (!x.load(std::memory\_order\_seq\_cst))  
 ;  
 if (y.load(std::memory\_order\_seq\_cst)) {  
 ++z;  
 }  
 }  
  
 void read\_y\_then\_x()  
 {  
 while (!y.load(std::memory\_order\_seq\_cst))  
 ;  
 if (x.load(std::memory\_order\_seq\_cst)) {  
 ++z;  
 }  
 }  
  
 int main()  
 {  
 std::thread a(write\_x);  
 std::thread b(write\_y);  
 std::thread c(read\_x\_then\_y);  
 std::thread d(read\_y\_then\_x);  
 a.join(); b.join(); c.join(); d.join();  
 // 如果不使用顺序一致模型的话，则此处就可能触发断言  
 assert(z.load() != 0);  
 }

#### 总结

需要再次强调的是，当程序对性能没有特殊要求时，首选std::mutex，其次使用memory\_order\_seq\_cst。只有当对cpu架构了解较深，且对性能要求苛刻的场合下才考虑使用其它内存序。

## 常量

### nullptr/nullptr\_t

C++11允许使用nullptr代替0或者NULL。

C++11标准不仅定义了指针空值常量nullptr，也定义了其指针空值类型nullptr\_t，也就表示了指针空值类型并非仅有nullptr一个实例。通常情况下，也可以通过nullptr\_t来声明一个指针空值类型的变量。

大致规则如下：

1. 所有定义为nullptr\_t类型的数据都是等价的，行为也是完全一致；
2. nullptr\_t类型数据可以隐式转换成任意一个指针类型；
3. nullptr\_t类型数据不能转换为非指针类型，即使使用reinterpret\_cast<nullptr\_t>()的方式也是不可以的；
4. nullptr\_t类型数据不适用于算术运算表达式；
5. nullptr\_t类型数据可以用于关系运算表达式，但仅能与nullptr\_t类型数据或者指针类型数据进行比较，当且仅当关系运算符为==、<=、>=等时返回true。

### constexpr

## 变量

### 强枚举类型

### if/switch

在传统C++中，变量的声明虽然能够位于任何位置，甚至于for语句内能够声明一个临时变量int，但始终没有办法在if和switch语句中声明一个临时的变量。

### 一致化初始化uniform initialization

在C++11之前，初始化可以使用{}，()或者=符号，在C++11中引入一致化初始化，你可以只使用()就完成初始化。

### 初始化列表initialize lists

**C++类内可以定义引用数据成员吗？**

可以，必须通过成员函数初始化列表初始化。

初始化是一个非常重要的语言特性，最常见的就是在对象进行初始化时进行使用。在传统C++中，不同的对象有着不同的初始化方法，例如普通数组、POD（Plain Old Data，即没有构造、析构和虚函数的类或结构体）类型都可以使用{}进行初始化，也就是我们所说的初始化列表。而对于类对象的初始化，要么需要通过拷贝构造、要么就需要使用()进行。这些不同方法都针对各自对象，不能通用。

为了解决这个问题，C++11首先把初始化列表的概念绑定到了类型上，并将其称之为std::initializer\_list，允许构造函数或其他函数像参数一样使用初始化列表，这就为类对象的初始化与普通数组和POD的初始化方法提供了统一的桥梁。

初始化列表的优势：

1. 丰富了自动变量和全局变量的初始化；
2. 使用初始化列表最大的优势是可以防止类型收窄（narrowing）。

注：类型收窄：指的是导致数据内容发生变化或者精度丢失的隐式类型转换。

在C++98/03中，类型收窄编译器不会报错，而C++11中可以通过列表初始化来检查及防止类型收窄。

<https://www.cnblogs.com/5iedu/p/7628814.html>

<http://events.jianshu.io/p/58c690713cb3>

### 结构化绑定

## 类型推导

C++11引入了auto和decltype这两个关键字实现了类型推导，让编译器来操心变量的类型。这使得C++也具有了和其他现代编程语言一样，某种意义上提供了无需操心变量类型的使用习惯。

### auto

在传统C++中，如果一个变量没有声明为register变量，将自动被视为一个auto变量。而随着register被弃用（在C++17中作为保留关键字，以后使用，目前不具备实际意义），对auto的语义变更也就非常自然了。

注：auto在类型很长或者类型很复杂的场景常用，其余场景不要滥用。

使用auto进行类型推导的一个最为常见而且显著的例子就是迭代器。



auto的优势：

1. auto推导最大的优势就是在拥有初始化表达式的复杂类型变量声明时简化代码；
2. 可以免除程序员在一些类型声明时的麻烦，或者避免一些在类型声明时的错误；
3. 其“自适应”性能够在一定程度上支持泛型的编程。

### decltype

decltype关键字是**为了解决auto关键字只能对变量进行类型推导的缺陷而出现的，可以让编译器获取表达式的类型**。

注：这类似于typeof的功能，但是typeof功能不是非常完善。

在C++11之前写法：

map<string,float>::value\_type elem; //不能自动由对象推导出类型

在C++11之后：

map<string,float> coll;

decltype(coll)::value\_type elem;

应用：

1. 声明return type；

template<typename T1,typename T2>

decltype(x+y) add(T1 x,T2 y);

这种写法在C++11前是不允许的，因为前面的返回表达式中x+y在编译的时候还不可见。在C++11中，你可以在函数后声明返回值参数：

template<typename T1,typename T2>

auto add(T1 x,T2 y)-> decltype(x+y);

1. 元编程（metaprogramming）
2. 传递Lambda表达式的类型

### 尾返回类型推导

### decltype(auto)

decltype(auto)主要用于对转发函数或封装的返回类型进行推导，它使我们无需显式的指定decltype的参数表达式。

## 控制流

### if constexpr

C++17将constexpr这个关键字引入到if语句中，允许在代码中声明常量表达式的判断条件。

### 区间for迭代range-based for statement

## 模板

### 外部模板

### 尖括号’>’

### 模板表达式空格space intemplate expressions

vector<list<int> >; //C++所有版本

vector<list<int>>; //C++11

### 类型别名模板alias template

别名就是模板的另一个名字，与#define和typedef效果不一样（#define只是简单替换，typedef不接收参数）。

定义：

template <typename T>

using Vec = std::vector<T,MyAlloc<T>>;

使用：

Vec<int> coll;

### 默认模板参数

### 变长参数模板variadic templates

数量不定的模板参数，可以传入任意个数和任意类型的参数，可以帮助我们做递归（不断将包中的参数拆分）。例如：

void print()//最后一次调用，什么都不做

{

}

template <typename T,typename… Types>

void print(const T& firstArg,const Types&…args)

{

cout<<firstArg<<endl;

print(args…);

}

可以下面的并存：

template < typename… Types>

void print(const Types&…args)

{

}

注：二者哪个泛化，哪个特化？调用的时候是否存在歧义？

参数列表中的省略号“...”表示多个参数，就是所谓的包（pack）：

1、用于template parameters，就是模板参数包（template parameters pack）

2、用于function parameter types，就是函数参数类型包（function parameter types pack）

3、用于function parameters，就是函数参数包（function parameters pack）

注：hash function和tuple实现中会用到

### 折叠表达式

### 非类型模板参数推导

## 面向对象

### 委托构造

C++11引入了委托构造的概念，这使得构造函数可以在同一个类中一个构造函数调用另一个构造函数，从而达到简化代码的目的。

### 继承构造

在传统C++中，构造函数如果需要继承是需要将参数一一传递的，这将导致效率低下。C++11利用关键字using引入了继承构造函数的概念。

### 异常noexception

### 显式虚函数重载override/final

override当重载虚函数时，引入override关键字将显式的告知编译器进行重载，编译器将检查基函数是否存在这样的虚函数，否则将无法通过编译。

final则是为了防止类被继续继承以及终止虚函数继续重载引入的。

### 默认函数控制

#### 类与默认函数

#### =default/=deleted

C++标准称“=default”修饰的函数为显式缺省（explicit defaulted）函数，而称“=delete”修饰的函数为删除（deleted）函数（可以应用在任何函数）。

一个类只要存在指针成员变量，几乎可以断定需要自己实现构造函数/拷贝函数/运算符重载（Big-Five），否则直接用默认的即可。

如果不允许拷贝，可以将类中拷贝相关的拷贝函数设置为delete（联系单例模式和深拷贝）。

注：拷贝构造函数不能有两个，因此不能给将拷贝构造函数设置=default，也不能设置为=delete（已经声明了又设置delete，这样编译器会报错）。运算符重载也不可以使用=default/=delete。

一般的函数没有默认版本，不可以使用=default，但是可以使用=delete（但是很少用）。

析构函数需要使用=default，如果设置=delete会造成使用已释放的对象。

**为什么编译器需要给类默认构造函数？**

构造函数除了执行初始化工作，主要的任务是执行一些幕后的工作，比如虚函数表，明确父子调用关系。

### 强类型枚举enum class

在传统C++中，枚举类型并非类型安全，枚举类型会被视作整数，则会让两种完全不同的枚举类型可以进行直接的比较（虽然编译器给出了检查，但并非所有），甚至同一个命名空间中的不同枚举类型的枚举值名字不能相同，这通常不是我们希望看到的结果。

C++11引入了枚举类（enumeration class），并使用enum class的语法进行声明。

## Lambda表达式

### 概述

C++11中的Lambda表达式用于定义并创建匿名的函数对象，以简化编程工作。

对于函数调用C++11之前操作：



C++11引入Lambda表达式之后：



### 参数

Lambda表达式的基本构成（匿名函数声明）：

[capture](parameters)mutable->return-type

{

statement

}

匿名函数调用：

[函数对象参数](操作符重载函数参数)mutable->返回值(函数体)

注：匿名函数声明和匿名函数调用的区别在于最后是否有()。

1. 函数对象参数：

[]，标识一个lambda表达式开始，这部分是必须存在的，不能省略。

函数对象参数是传递给编译器自动生成的函数对象类的构造函数的。函数对象只能使用那些到定义Lambda为止时Lambda所在作用范围内可见的局部变量（包括Lambda所在类的this）。函数对象参数有以下形式：

（1）空。没有使用任何函数对象参数。

（2）=。函数体内可以使用Lambda所在作用范围内所有可见的局部变量（包括Lambda所在类的this），并且是值传递方式（相当于编译器自动为我们的值传递了所有局部变量）。

（3）&。函数体内可以使用Lambda所在作用范围内所有可见的局部变量（包括Lambda所在类的this），并且是引用传递方式（相当于编译器自动为我们按引用传递了所有局部变量）。

（4）this。函数体内可以使用Lambda所在类中的成员变量。

（5）a。将a按值进行传递。按值传递时，函数体内不能修改传递进来的a的拷贝，因为默认情况下函数是const的。要修改传递进来的a的拷贝，可以添加mutable修饰符。

（6）&a。将a按引用进行传递。

（7）a,&b。将a按值进行传递，b按引用进行传递。

（8）=,&a,&b。除a和b按引用进行传递外，其他参数都按值进行传递。

注：推荐使用=，不推荐使用&。在Qt中，做信号-槽连接的时候，默认内部变量会进行锁状态，只读状态，如果进行写操作，就会core。

1. 操作符重载函数参数：

标识重载的()操作符的参数，没有参数时。这部分可以省略。参数可以通过按值（如：(a,b)）和按引用(如:(&a,&b))两种方式进行传递。

1. 可修改标识符：

mutable声明，这部分可以省略。按值传递函数对象参数时，加上mutable修饰符后，可以修改按值传递进来的拷贝（注意是能修改拷贝，而不是值本身）。

1. 函数返回值：

->返回值类型。标识函数返回值的类型，当返回值为void，或者函数体只有一处return的地方（此时编译器可以自动推断出返回值类型）时，这部分可以省略。

1. 函数体：

{}，标识函数的实现，这部分不能省略，但函数体可以为空。

### 使用

### Lambda与仿函数

### Lambda与STL

## 函数对象包装器

### function

### bind

## 右值引用rvalue references

### 拷贝构造函数

### 左值、右值和右值引用

#### 概述

右值引用是一种新的引用，可以解决不必要的拷贝（**当赋值语句右边是右值rvalue时，左边的对象可以不用重新分配内存去steal右边的内存，避免重新分配内存**）。

注：**右值引用主要是为了move的实现服务**，因为左值需要指向右值的地址（原来的指针失效，只有新的指针有效，有点类似深拷贝和浅拷贝结合，C++11之前容器insert原型为insert(…,&x)，C++11后为insert(…,&&x)）。

左值：可以出现在operator=左侧者（变量就是左值，函数是左值）

右值：只能出现在operator=右侧者（临时对象就是右值）

C++11之前不支持右值引用：



右值引用是C++11中引入的新特性 , 它实现了转移语义和精确传递。它的主要目的有两个方面：

1、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。

2、能够更简洁明确地定义泛型函数。

**左值和右值的概念：**

左值：能对表达式取地址、或具名对象/变量。一般指表达式结束后依然存在的持久对象。

右值：不能对表达式取地址，或匿名对象。一般指表达式结束就不再存在的临时对象。

#### 左值引用和右值引用

右值引用和左值引用的区别：

1、左值可以寻址，而右值不可以。

2、左值可以被赋值，右值不可以被赋值，可以用来给左值赋值。

3、左值可变，右值不可变（仅对基础类型适用，用户自定义类型右值引用可以通过成员函数改变）。

### std::move

std::move实际上并不能移动任何东西，它唯一的功能就是将一个左值强制转化为右值引用，继而我们可以通过右值引用使用该值，以用于移动语义。

从实现上将，std::move基本等同于一个类型转换：

static\_cast<T&&>(lvaue);

注：被转化的左值，其生命期并没有随着左右值的转化而改变，如果期望move转化的左值变量lvalue能够立即被析构，那么肯定会失望了。

优点：调用此函数不会引起任何数据争用，高效完成权限转移（不需要拷贝内存）。

参考：<https://blog.csdn.net/grf123/article/details/105379337/>

<https://blog.csdn.net/qq_30683329/article/details/88532673>

<https://zhuanlan.zhihu.com/p/256239799>

### std::forward

多个嵌套调用的时候，需要做到完美转交（perfect forwarding）。

## 容器

### 头文件

头文件新特性：

1. C++标准库的header files不带副档名（.h），例如#include<vector>
2. 新式C header files不带副名称.h，例如#include<cstdio>
3. 旧式C header files（带有副名称.h）仍可用，例如#include<stdio.h>

### 线性容器

#### array

与std::vector不同，std::array对象的大小是固定的，如果容器大小是固定的，那么可以优先考虑使用std::array容器。

另外由于std::vector是自动扩容的，当存入大量的数据后，并且对容器进行了删除操作，容器并不会自动归还被删除元素相应的内存，这时候就需要手动运行shrink\_to\_fit()释放这部分内存。

使用std::array能够让代码变得更加“现代化”，而且封装了一些操作函数，比如获取数组大小以及检查是否非空，同时还能够友好的使用标准库中的容器算法，比如std::sort。

#### forward\_list

std::forward\_list是一个列表容器，使用方法和 std::list基本类似。

和std::list的双向链表的实现不同，std::forward\_list使用单向链表进行实现，提供了O(1) 复杂度的元素插入，不支持快速随机访问（这也是链表的特点），也是标准库容器中唯一一个不提供 size()方法的容器。当不需要双向迭代时，具有比std::list更高的空间利用率。

### 无序容器

#### hash function

哈希函数用来计算hash code，用于unordered\_set和unordered\_map等无序容器。

例如：

hash<int>()(123);

hash<long>()(123L);

hash<string>()(string(“abc”));

hash<const char\*>()(“abc”);

#### unordered containers

无序容器中的元素是不进行排序的，内部通过Hash表实现，插入和搜索元素的平均复杂度为 O(constant)，在不关心容器内部元素顺序时，能够获得显著的性能提升。

注：STL中的set和map是基于红黑树的，hash\_set和hash\_map是基于哈希表，unordered\_set和unordered\_map是基于哈希表。

C++11引入了两组无序容器：

std::unordered\_map/std::unordered\_multimap，std::unordered\_set/std::unordered\_multiset。

### tuple

纵观传统C++中的容器，除了std::pair外，似乎没有现成的结构能够用来存放不同类型的数据（通常我们会自己定义结构）。但std::pair的缺陷是显而易见的，只能保存两个元素。

基本操作：

1. std::make\_tuple: 构造元组

2. std::get: 获得元组某个位置的值

3. std::tie: 元组拆包

## 智能指针

C++里面的四个智能指针: auto\_ptr，shared\_ptr，weak\_ptr，unique\_ptr其中后三个是c++11支持，并且第一个已经被11弃用。

为什么要使用智能指针？

智能指针的作用是管理一个指针，因为存在以下这种情况：申请的空间在函数结束时忘记释放，造成内存泄漏。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。所以智能指针的作用原理就是在函数结束时自动释放内存空间，不需要手动释放内存空间。

### RAII

RAII是Resource Acquisition Is Initialization（“资源获取就是初始化”）的简称，是C++语言的一种管理资源、避免泄漏的惯用法。利用的就是C++构造的对象最终会被销毁的原则。RAII的做法是使用一个对象，在其构造时获取对应的资源，在对象生命期内控制对资源的访问，使之始终保持有效，最后在对象析构的时候，释放构造时获取的资源。

### auto\_ptr

C++98的方案，CPP11已经抛弃。

**采用所有权模式**。

auto\_ptr< string> p1 (new string ("I reigned lonely as a cloud.”));

auto\_ptr<string> p2;

p2 = p1; //auto\_ptr不会报错

此时不会报错，p2剥夺了p1的所有权，但是当程序运行时访问p1将会报错。所以auto\_ptr的缺点是：存在潜在的内存崩溃问题！

### unique\_ptr

std::unique\_ptr是一种独占的智能指针，它禁止其他智能指针与其共享同一个对象，从而保证代码的安全。

make\_unique并不复杂，C++11没有提供std::make\_unique。

既然是独占，换句话说就是不可复制。但是，我们可以利用std::move将其转移给其他的unique\_ptr。

unique\_ptr实现独占式拥有或严格拥有概念，保证同一时间内只有一个智能指针可以指向该对象。它对于避免资源泄露(例如“以new创建对象后因为发生异常而忘记调用delete”)特别有用。

采用所有权模式，还是上面那个例子

unique\_ptr<string> p3 (new string ("auto"));   //#4

unique\_ptr<string> p4；                       //#5

p4 = p3;//此时会报错！！

编译器认为p4=p3非法，避免了p3不再指向有效数据的问题。因此，unique\_ptr比auto\_ptr更安全。

另外unique\_ptr还有更聪明的地方：当程序试图将一个 unique\_ptr 赋值给另一个时，如果源 unique\_ptr 是个临时右值，编译器允许这么做；如果源 unique\_ptr 将存在一段时间，编译器将禁止这么做，比如：

unique\_ptr<string> pu1(new string ("hello world"));

unique\_ptr<string> pu2;

pu2 = pu1;                                      // #1 not allowed

unique\_ptr<string> pu3;

pu3 = unique\_ptr<string>(new string ("You"));   // #2 allowed

其中#1留下悬挂的unique\_ptr(pu1)，这可能导致危害。而#2不会留下悬挂的unique\_ptr，因为它调用 unique\_ptr 的构造函数，该构造函数创建的临时对象在其所有权让给 pu3 后就会被销毁。这种随情况而已的行为表明，unique\_ptr 优于允许两种赋值的auto\_ptr 。

注：如果确实想执行类似与#1的操作，要安全的重用这种指针，可给它赋新值。C++有一个标准库函数std::move()，让你能够将一个unique\_ptr赋给另一个。例如：

unique\_ptr<string> ps1, ps2;

ps1 = demo("hello");

ps2 = move(ps1);

ps1 = demo("alexia");

cout << \*ps2 << \*ps1 << endl;

### shared\_ptr

std::shared\_ptr是一种智能指针，它能够记录多少个shared\_ptr共同指向一个对象，从而消除显示的调用delete，当引用计数变为零的时候就会将对象自动删除。

但还不够，因为使用std::shared\_ptr仍然需要使用new来调用，这使得代码出现了某种程度上的不对称。std::make\_shared 就能够用来消除显式的使用 new，所以**std::make\_shared会分配创建传入参数中的对象，并返回这个对象类型的std::shared\_ptr指针**。

**std::shared\_ptr可以通过get()方法来获取原始指针，通过reset()来减少一个引用计数，并通过use\_count()来查看一个对象的引用计数**。

**shared\_ptr实现共享式拥有概念**。多个智能指针可以指向相同对象，该对象和其相关资源会在“最后一个引用被销毁”时候释放。从名字share就可以看出了资源可以被多个指针共享，它使用**计数机制**来表明资源被几个指针共享。可以通过成员函数use\_count()来查看资源的所有者个数。除了可以通过new来构造，还可以通过传入auto\_ptr，unique\_ptr，weak\_ptr来构造。当我们调用release()时，当前指针会释放资源所有权，计数减一。当计数等于0时，资源会被释放。

shared\_ptr是为了解决auto\_ptr在对象所有权上的局限性(auto\_ptr是独占的)，在使用引用计数的机制上提供了可以共享所有权的智能指针。

成员函数：

use\_count返回引用计数的个数

unique返回是否是独占所有权( use\_count为1)

swap交换两个shared\_ptr对象(即交换所拥有的对象)

reset放弃内部对象的所有权或拥有对象的变更，会引起原有对象的引用计数的减少

get返回内部对象(指针)，由于已经重载了()方法，因此和直接使用对象是一样的.如 shared\_ptr<int> sp(new int(1)); sp与sp.get()是等价的

使用：

比如在使用共享内存的时候，申请内存后可以使用shared\_ptr接收，这样方便多个进程/线程使用该共享内存。

### weak\_ptr

**智能指针的内存泄漏如何解决？**

为了解决循环引用导致的内存泄漏，引入了weak\_ptr弱指针，weak\_ptr的构造函数不会修改引用计数的值，从而不会对对象的内存进行管理，其类似一个普通指针，但不指向引用计数的共享内存，但是其可以检测到所管理的对象是否已经被释放，从而避免非法访问。

weak\_ptr是一种不控制对象生命周期的智能指针，它指向一个shared\_ptr管理的对象。进行该对象的内存管理的是那个强引用的shared\_ptr，weak\_ptr只是提供了对管理对象的一个访问手段。weak\_ptr设计的目的是为配合shared\_ptr而引入的一种智能指针来协助shared\_ptr工作，它只可以从一个shared\_ptr或另一个weak\_ptr对象构造，它的构造和析构不会引起引用记数的增加或减少。

weak\_ptr是用来解决shared\_ptr相互引用时的死锁问题，如果说两个shared\_ptr相互引用，那么这两个指针的引用计数永远不可能下降为0，资源永远不会释放。它是对对象的一种弱引用，不会增加对象的引用计数，和shared\_ptr之间可以相互转化，shared\_ptr可以直接赋值给它，它可以通过调用lock函数来获得shared\_ptr。

class B;

class A

{

public:

shared\_ptr<B> pb\_;

~A()

{

cout<<"A delete\n";

}

};

class B

{

public:

shared\_ptr<A> pa\_;

~B()

{

cout<<"B delete\n";

}

};

void fun()

{

shared\_ptr<B> pb(new B());

shared\_ptr<A> pa(new A());

pb->pa\_ = pa;

pa->pb\_ = pb;

cout<<pb.use\_count()<<endl;

cout<<pa.use\_count()<<endl;

}

int main()

{

fun();

return 0;

}

可以看到fun函数中pa，pb之间互相引用，两个资源的引用计数为2，当要跳出函数时，智能指针pa，pb析构时两个资源引用计数会减一，但是两者引用计数还是为1，导致跳出函数时资源没有被释放（A B的析构函数没有被调用），如果把其中一个改为weak\_ptr就可以了，我们把类A里面的shared\_ptr pb\_; 改为weak\_ptr pb\_; 运行结果如下，这样的话，资源B的引用开始就只有1，当pb析构时，B的计数变为0，B得到释放，B释放的同时也会使A的计数减一，同时pa析构时使A的计数减一，那么A的计数为0，A得到释放。

注意：我们不能通过weak\_ptr直接访问对象的方法，比如B对象中有一个方法print()，我们不能这样访问，pa->pb\_->print(); 英文pb\_是一个weak\_ptr，应该先把它转化为shared\_ptr，如：shared\_ptr p = pa->pb\_.lock(); p->print();

## 垃圾回收

### 分类

### C++垃圾回收

### 最小垃圾回收

## 正则表达式

### 概述

### regex

## 并发

std::thread用于创建一个执行的线程实例，所以它是一切并发编程的基础，使用时需要包含<thread>头文件，它提供了很多基本的线程操作，例如get\_id()来获取所创建线程的线程ID，使用join()来加入一个线程等等。

### 内存模型

参考：<https://www.cnblogs.com/haippy/p/3412858.html>

### 线程/thread

### 原子操作/atomic

### 互斥量与临界区/mutex

std::mutex是C++11中最基本的mutex类，通过实例化std::mutex可以创建互斥量，而通过其成员函数lock()可以进行上锁，unlock()可以进行解锁。但是在在实际编写代码的过程中，最好不去直接调用成员函数，因为调用成员函数就需要在每个临界区的出口处调用unlock()，当然，还包括异常。这时候C++11还为互斥量提供了一个RAII语法的模板类 std::lock\_gurad。RAII在不失代码简洁性的同时，很好的保证了代码的异常安全性。

### 锁/lock

C++11标准为我们提供了两种基本的锁类型，分别如下：

std::lock\_guard，与Mutex RAII相关，方便线程对互斥量上锁。

std::unique\_lock，与Mutex RAII相关，方便线程对互斥量上锁，但提供了更好的上锁和解锁控制。

### 条件变量/condition\_variable

条件变量std::condition\_variable是为了解决死锁而生，当互斥操作不够用而引入的。比如，线程可能需要等待某个条件为真才能继续执行，而一个忙等待循环中可能会导致所有其他线程都无法进入临界区使得条件为真时，就会发生死锁。所以，**condition\_variable实例被创建出现主要就是用于唤醒等待线程从而避免死锁。std::condition\_variabl的notify\_one()用于唤醒一个线程；notify\_all()则是通知所有线程**。

### 期物/future

参考：<https://blog.csdn.net/c_base_jin/article/details/89761718>

期物（Future）表现为std::future，它提供了一个访问异步操作结果的途径。

### 总结