输入输出流

2016年7月3日　　21:45

输出：

```
#include <iostream> //输出输出流库
int main() {
  cout << "Hello, World! I am "          //cout为终端输出
       << 8 << " Today!" << endl; //endl为行末
} ///:~
```

输入：

```
cin >> number;        //读取标准输入赋值给number
```

文件读写：

```
#include <string>
#include <fstream>          //文件流库
int main() {
  ifstream in("Scopy.cpp");         //创建文件输入流ifstream
  ofstream out("Scopy2.cpp");       //创建文件输出流ofstream
  string s;
  while(getline(in, s))     //getline从in流读取一行字符串给s
    out << s << "\n";
} ///:~
```

string类

2016年7月4日 22:56

```
#include <string>     //字符串库
#include <iostream>
using namespace std;          //string在std名字空间中
int main() {
  string s1, s2; //空字符串
  string s3 = "Hello, World."; //用字符串初始化一个string
  string s4("I am");     //用创建对象的方式初始化一个string，效果同s3
  s2 = "Today";          //string的赋值
  s1 = s3 + " " + s4;    //用+连接字符串
  s1 += " 8 ";   //用+=追加
  cout << s1 + s2 + "!" << endl;
}
```

vector容器

2016年7月4日 23:01

最基本的标准容器

```
#include <string>
#include <iostream>
#include <fstream>
#include <vector>              //vector容器库
using namespace std;
int main() {
  vector<string> v;            //创建一个用来装string类的对象的vector容器
  ifstream in("Fillvector.cpp");
  string line;
  while(getline(in, line))
    v.push_back(line);  //通过vector的push_back函数将字符串压入容器末尾
  // Add line numbers:
  for(int i = 0; i < v.size(); i++)                  //vector容器的size成员记录了容器的当前大小
    cout << i << ": " << v[i] << endl;//vector容器可以直接像数组一通过下标进行访问
```

对象

2016年7月4日　　0:20

对象的意义：解决命名冲突的问题

头文件形式

2016年7月4日 22:35

1、基本原则：只放声明

每个事物可以被声明多次，但只能定义一次；

而头文件可能会被多次引用

2、防止重声明

```
#ifndef SIMPLE_H
#define SIMPLE_H
struct Simple {
  int i,j,k;
  initialize() { i = j = k = 0; }
};
#endif // SIMPLE_H
//endif后必须加注释才符合规定
```

3、不去除“名字空间保护”

头文件中通常不加入指令：`using namespace std;`

嵌套

2016年7月4日 22:36

Stack.h

```
#ifndef STACK_H
#define STACK_H
struct Stack {  //外层
  struct Link {   //内层
    void* data;
    Link* next;
    void initialize(void* dat, Link* nxt);
  }* head;
  void initialize();
  void push(void* dat);
  void* peek();
  void* pop();
  void cleanup();
};
#endif // STACK_H ///:~
```

Stack.cpp

```
#include "Stack.h"    //引用同目录下的Stack.h文件
#include "../require.h"   //引用上一层目录下的require.h文件
using namespace std;    //使用名字空间std
void
Stack::Link::initialize(void* dat, Link* nxt) {   //外层结构(Stack)::内层结构(Link)::函数名(参数...)
  data = dat;
  next = nxt;
}
void Stack::initialize() { head = 0; }
```

全局&局部

```
int a;          //全局变量
void f() {}             //全局函数
struct S {
  int a;          //局部变量
  void f();       //局部函数
};
```

```
void S::f() {
  ::f();     // 调用全局函数f()
  ::a++; // 全局变量a
  a--;    // 局部变量a，即S结构体的a
}///:~
```

访问控制

2016年7月4日 22:55

理由：

1. 让库的使用者忽略一些他们不需要使用的工具（内部辅助函数等）
2. 允许库的设计者改变内部实现而不影响使用者的使用

访问说明符

- public：可以被所有人访问
- private：除了类型的创建者和类内部成员函数之外，任何人都不能访问
- protected：与private类似，而继承的结构可以访问protected成员，但不能访问private成员

```
struct B {
private:        //i和f是private成员，只能被该类的创建者和内部成员访问
  char j;
  float f;
public:         //i和func是public成员，可以被所有人访问
  int i;
  void func();
```

访问说明符在编译的时候会被去除，
访问权限的检查只由编译器来完成，而与连接器无关

友元

2016年7月5日 21:20

声明为友元的函数、结构体可以直接访问、修改自己的私有成员

友元实例：friend.cpp

```
struct X;  //声明一个X的结构体，但不给出定义，方便Y结构体引用
struct Y {
 //不直接用X作为参数，因为C++解释器要求使用X前必须先给出X的定义
 //但这里用X的指针，之前声明了X的结构体，即使没有定义也是能使用X的指针的
 void f(X*);
};
struct X { //定义一个X的结构体
private:
 int i;
public:
 void initialize(); //X的成员函数
 //友元g、y::f、Z都可以直接访问、修改X的成员i
 friend void g(X*, int); //全局函数g的友元
 friend void Y::f(X*); //Y结构体的f员函数的友元
 friend struct Z; //Z结构体的友元
};
```

嵌套结构中的友元：

NestFriend.cpp

```
#include <iostream>
#include <cstring> // memset()
using namespace std;
const int sz = 20;
struct Holder {
private:
 int a[sz];
public:
 void initialize();
 struct Pointer;  //必须先声明子结构
 friend struct Pointer;  //将子结构Pointer作为友元
 struct Pointer { //定义子结构
  //...
```

```
  };
};
void Holder::initialize() {
  //用函数menset()将起始于a的sz*sizeof(int)个字节的单元初始化为0
  //也可以用循环赋值，但是该函数更加直观、高效、安全
  memset(a, 0, sz * sizeof(int));
}
```

C++不是纯面向对象语言，添加friend是为了解决一些实际问题，C++追求的是实用，而非理想的面向对象

类

2016年7月6日 14:32

class和struct关键字的作用几乎是一样的，
只不过class默认访问说明符为private，而struct默认访问说明符为public

句柄类

2016年7月6日 15:59

意义：

1. 完全的隐藏实现
   普通的类、结构体虽然加了访问说明符，客户程序员无法轻易的访问私有部分，但却是可见的
2. 减少重复编译
   解决修改一个头文件时，所有包含该头文件的所有文件都需要重新编译的问题

类似于java中的interface/implement技术

客户程序员可见的头文件：Handle.h

```
class Handle {          //句柄类
  struct Cheshire;      //声明一个类但不在这定义
  Cheshire* smile;      //有了声明就可以定义一个对应的指针
public:           //以下为接口函数
  void initialize();
  void cleanup();
  int read();
  void change(int x)
}
```

客户程序员不可见：Handle.cpp

```
#include "Handle.h"
struct Handle::Cheshire {      //类的真正实现
  int i;
};
//以下为接口函数的实现
//因为有了Cheshire的指针，所以Handle可以对Cheshire进行操作
//从而使接口、实现彻底分离，接口的头文件可见，而cpp被编译后隐藏在目标文件当中
void Handle::initialize() {
  smile = new Cheshire;
  smile->i = 0;
}
void Handle::cleanup() {
  delete smile;
}
int Handle::read() {
```

```
  return smile->i;
}
void Handle::change(int x) {
  smile->i = x;
}
```

构造函数&析构函数

2016年7月6日 16:44

构造函数即类所属的与类同名的函数，在对象被创建出来时被自动调用；
析构函数是名为“~类名”的函数，在对象生命周期结束时被自动调用

```
using namespace std;
class Tree {
  int height;
public:
  Tree(int initialHeight); //构造函数，可以带初始化的参数
  ~Tree(); //析构函数，不能带参数
  void printsize();
};
Tree::Tree(int initialHeight) {
  height = initialHeight;
}
Tree::~Tree() {
  cout << "inside Tree destructor" << endl;
  printsize();
}
```

创建对象时：
Tree t(3);
或Tree t = new t(3);

清除定义块

2016年7月6日 21:24

变量的定义应尽可能接近使用的地方

for、while、switch语句允许在表达式内定义变量
如：for(int i = 0; i <10; i++) .....;

应尽可能缩小变量的生命周期，防止被误用，也提高的程序的可读性

内存分配：
通常，C++编译器会和C编译器一样在作用域的开头就直接为所有变量分配好内存；
但是如果变量的定义语句未被执行，即未对内存空间赋予含义，那么这个内存空间是没有意义的，同时构造函数也只有在定义语句执行时才会被调用（而不是在作用域的开始时）；
甚至，当把对象的定义放在条件块等可能不被执行的位置而后续会使用相应的标识符时，编译器会产生相应的警告或错误

C++和JAVA的对象创建：

- C++
  直接定义：如X x;
  new：如X x = new X;
- JAVA
  不能直接定义，X x代表的是X类的一个引用，类似指针；
  只能通过new：即X x=new X;

集合初始化

2016年7月6日　　21:55

数组的初始化：

```
int a[5] = {1,2,3,4,5};            //对所有元素进行初始化
int b[6] = {1};            //将第一个元素初始化为1，其余元素会被初始化为0
int c[6];          //不进行任何初始化操作
int d[] = {1,2,3,4};    //编译器自动从初始化的值的数量确定元素的数量
int e[5];
for(int i = 0; i < sizeof e / sizeof *e; i++)
        //(sizeof e / sizeof *e)为数组元素个数，这种做法在修改数组大小时无需修改循环的次数
        e[i] = i;
```

对于对象的初始化：

如果对象的成员均为public：

```
struct X{
        int i;
        float f;
        char c;
};
X x1 = {1, 2.2, 'c'};      //用花括号把成员的值括起来
X x2[3]  = { {1, 1.2, 'a'}, {2, 2.2, 'b'} }; //第三个元素的成员都被初始化为0
```

如果对象的成员存在private：

此时外部无法直接访问private成员，因此必须借助构造函数来完成初始化操作

```
class Y{
        float f;
        int i;
        //私有成员f和i
public:
        Y(int i);
};
Y y[] = { Y(1), Y(2), Y(3) };       //借助构造函数完成对象的初始化
```

默认构造函数

2016年7月6日 22:40

即不带任何参数的构造函数

当一个类没有定义任何构造函数数，编译器会自动合成一个默认构造函数，
它不会将内存清零，而只是完成最少的工作；

当一个类定义了一个带参数的构造函数，那么编译器要求必须再定义一个默认构造函数，
以确保不管什么情况下构造函数总会被调用；

尽管编译器会自动合成一个默认的构造函数，但一般来说应当尽量少依赖他，尽可能自己定义默认构造函数而不是让编译器来完成

函数重载

2016年7月7日　　21:12

名字修饰：
在C中，函数名与内部名一致，所以即使声明为“void f(char)”，定义“void f(int)”也不会报错，既能编译成功又能链接成功
在C++中，函数名是经过修饰的，即内部名与函数名不同，不同编译器的修饰方式也不同，如“void f(char)”的内部名可能为“f_char”；如果声明为“void f(char)”而定义为“void f(int)”，虽然可以编译成功但连接时找不到相应的函数而连接失败并报错

函数重载与java类似，直接声明、定义多个同函数名而不同参数列表的函数即可

联合union同样可以使用成员函数、构造函数、析构函数、访问控制等等；
但是联合中不同类型的数据放在同一内存区，不能作为继承的基类；
尽管这样可以使union变得规范，但是难以阻止用错误的元素类型去访问数据；
此时可以利用函数重载和枚举类型将联合封装到一个类里：

```
#include <iostream>
using namespace std;
class SuperVar {
  enum {
    character,
    integer,
  } vartype;  //定义一个枚举变量，作为数据类型的标志
  union {   //联合体
    char c;
    int i;
  };
public:
  SuperVar(char ch);    //针对char的构造函数
  SuperVar(int ii);     //针对int的构造函数
  void print();
};
SuperVar::SuperVar(char ch) {
  vartype = character;
  c = ch;
}
SuperVar::SuperVar(int ii) {
  vartype = integer;
  i = ii;
```

```
}
void SuperVar::print() {
  switch (vartype) {
    case character:
      cout << "character: " << c << endl;
      break;
    case integer:
      cout << "integer: " << i << endl;
      break;
```

union的目的是为了减少开销，而此处额外使用了枚举类型；
不过如果存在多个union实例，则可以让他们共用一个枚举类型，此时可以消除额外的开销

默认参数

2016年7月9日 10:40

默认参数的用法与python类似，但有一些要注意的地方：

1、默认值只能放在声明中，而不能放在定义中（定义中通常用注释标出默认值）

```
class A{
        int i;
        void lalala(int a=0);
}
A::lalala(int a /* = 0*/){ … }
```

2、带默认值的参数必须放在参数列表的尾部

占位符参数

C++允许参数不给出参数名而只给出参数类型，如void lalala(int a, int, int c){…}

此时a和c可以被引用，中间的int起占位符的作用，以便以后可以修改函数定义时不需要修改函数调用；

调用时也必须给这个占位符参数赋值；

如果一个函数的某个明确的参数没有被使用，则编译器会给出警告信息，如果是占位符参数则不会警告

函数重载&默认参数

2016年7月9日 11:01

使用函数重载还是默认参数，一般应着重考虑程序的开发效率和易读性，其次再考虑程序的运行效率；
默认参数无论如何都会为参数开辟空间，而函数重载会选择相应的函数体而不会额外开辟空间；
不能把默认参数作为标志去决定执行函数的哪一块，这不易阅读；
一个默认的参数往往是默认值出现的可能性比其他值要大，通常可以忽略

通常两个函数体如果比较接近的话，使用默认参数；
如果两个函数体相差甚远，那么用函数重载会更好

值替代

2016年7月9日 11:08

define：宏命令，只在预处理阶段出现，不进入编译阶段，编译器无法查错
const：限定符，在编译阶段处理，编译器能够对它进行查错，而且会对一些复杂的常量表达式进行浓缩简化，因此最好用const代替define来定义常量

const通常代替define定义在头文件中，
默认为内部连接，也即只有被const定义过的文件里才是可见的，此时常量只存在于符号表中，不分配存储空间，但定义时必须对常量进行初始化；
如果需要引用其他文件的const定义，那么可以手动用extern声明为外部连接，此时会分配存储空间，无需对常量进行初始化。定义如extern const int bufsize = 1，声明如extern const int bufsize;

如果const仅起值替代作用，那么该常量会被折叠到代码当中，而不会占用存储空间

另外，const还可以用于限定一个变量在其生命周期内是不可改变的（但不是常量）
这种情况下，定义变量的同时也必须给它赋初值

```
#include <iostream>
using namespace std;
const int i = 100; //常量，不分配存储空间
const int j = i + 10; //常量表达式，仍为常量
long address = (long)&j; //此处由于取了j常量的地址，就迫使编译器为j分配存储空间
char buf[j + 10]; //即使分配了存储空间，j仍是常量
int main() {
  cout << "type a character & CR:";
  //c和c2为不可改变的字符变量（不是常量）
  const char c = cin.get();
  const char c2 = c + 'a';
  cout << c2;
```

集合中的const
const可以用于集合，此时集合会分配存储空间，而且是一个不可改变的存储空间；
集合的元素并不是常量，因此不能在编译期间使用它的值，比如不能在数组定义时作为数组大小的值

与C语言的比较

1. C语言中不能把const常量作为数组大小的值
2. C语言中允许在定义常量时不给初值
3. C语言中默认const是外部连接，即一定会分配存储空间

指针

2016年7月10日 23:11

指向const的指针

const的原则：修饰最接近它的那个类型

const int* u和int const* u都定义了一个普通的指针，它指向const int类型（const最靠近的都是int），为了避免混淆，应使用第一种定义方式

const指针

int* const w = &d;

int const* const x = &d; 或 const int* const x = &d;

格式：

通常星号*附着于前边的数据类型，使起看起来是一个整体的类型，比较直观易读

将一个const对象的地址赋给一个非const指针是不允许的

函数参数和返回值

2016年7月10日　　23:40

传递const值

给参数加上限定符const,表示该参数在函数内不可变；

此时这个参数是函数创建者的工具，而不是函数调用者的工具；

为了不让调用者混淆，在函数内部用const限定比在参数表里限定更加合适。即参数表中不加const限定符，而在函数开头定义一个const限定的变量并初始化为相应的参数（使用引用），如——

```
void f2(int ic){
    const int& i = ic;
    //....
}
```

返回const值

对于内部类型来说，返回值是否带const是没有意义的 ，因此应该去掉const，避免客户程序员混淆；

对于对象来说，返回值带const表示该返回的对象不可改变；

临时量

有时在求表达式期间，编译器必须创建一个程序员不可见的临时对象，编译器负责决定他们的去留和存在细节，编译器将临时量都作为const，当尝试改变临时量时就会报错；

```
X f5();          //f5返回一个非const对象
void f7(X& x);          //f7的参数为一个非const对象，并且在函数体中对x进行修改
//f7( f5() )      //错误！
```

f5()返回一个非const对象，但作为参数，编译器会为它创建一个const的临时量，如果f7是值传递，则不会出现错误；但如果f7是指针、引用传递，则函数内部实际修改的是一个临时对象，此时会出现错误；

另外，尽管以下表达式是合法的：

```
f5() = X(1);
f5().modify();
```

但是第二行的f5()由编译器产生一个临时量，即使被modify修改了，原对象并没有发生改变，是没有意义的~

传递和返回地址

当尝试把一个const指针赋为一个非const指针时编译器会报错；

对于参数为const指针的函数，实参用const指针和非const指针都是可以的；

对于传递对象来说，地址（引用）传递往往比值传递更有效，因为无需重新拷贝一个对象，但是为了对象不被函数改变，往往加入const限定；

由于临时量为const，所以当形参为const时可以接受临时对象，但形参不为const时不能接受临时对象，因此形参中通常为引用加上const限定；

```
X f();              //f返回一个对象
void g1(X&);  //g1形参为非const引用
void g2(const X&);              //g2形参为const引用
int main() {
//!  g1(f());              //编译器对f()产生一个临时对象传递给g1，g1不接受非const引用，错误！
  g2(f());              //g2接受const引用，正确
}
```

类

2016年7月12日 14:47

类内const与构造函数初始化列表

```
class Fred {
  const int size;          //只声明一个const而不赋初值
public:
  Fred(int sz);
  void print();
};
//构造函数初始化列表：
//构造函数参数列表之后可以带一个初始化列表，如这里的size(sz)表示把内部成员size初始化为sz
//初始化列表可以在构造函数的之前完成初始化
//对于const内部成员初始化必须通过初始化列表而不能在函数体里内赋值
Fred::Fred(int sz) : size(sz) {}
void Fred
```

内部类型的“构造函数”

有时候可以通过为内部类型构造一个类，达到使内部类型具有“构造函数”的目的；

如下面一个初始化示例，很多编译器都会把这个初始化优化成一个很快的过程，其开销要低于memset函数：

```
#include <iostream>
using namespace std;
class Integer {                    //为int内部类型设计一个Integer类
  int i;  //只有一个int内部成员
public:
  Integer(int ii = 0);    //构造函数的默认参数值为0
  void print();
};
Integer::Integer(int ii) : i(ii) {}          //通过初始化列表为内部成员初始化
void Integer::print() { cout << i << ' '; }
int main() {
  Integer i[100];          //由于默认参数值为0，此处没有给出初始化值，i的所有元素将被初始化为0
  for(int j = 0; j < 100; j++
```

编译期间的类内常量

上述const变量都只是生命周期内不可改变得变量，而不是实际上的常量，也就是说他们不能用来作为数组大小；

如果要定义一个编译期间的类内常量，必须同时用static const限制符，此时必须在定义处初始化而不能再构造函数初始化列表中进行初始化操作；

在旧版本的C++中没有static const特性，往往采用enum来实现编译期间的类内常量，如：enum { size = 100 }; int i[size]，但没有绝对的理由应当优先使用static const

const对象和成员函数

const对象的定义方式与const内部类型类似，它表示一个生命周期内不可改变的对象；

而const对象只能调用const成员函数，对于任何非const成员函数，编译器都将其视为一个企图修改对象的成员函数而报错；

定义和声明const成员函数，参数列表后加const限定符即可，如“int f() const {}”此时函数如果存在任何修改对象的行为都会被编译器标记为错误；

按位const和按逻辑const

按位const：对象中每个字节都是不可变的；

按逻辑const：对象整体概念不可变，但可以改变其成员；

定义一个const对象时，默认是绝对保护的按位const，如果要实现按逻辑const，只能通过内部const成员函数去修改内部成员；

为了使内部成员可以被const成员函数所修改，需要为成员声明添加mutable限定符；

在旧版本的C++中，不支持mutable特性，通常采用“强制转换常量性”的方法，即对this（即对象本身）强制转换为非const指针，再通过它访问内部成员，如“((Y*)this)->i++”

volatile

2016年7月12日 15:52

volatile语法与const一样，表示“在编译器认识范围之外，这个数据可以被改变”，即不知何时可能会被环境所改变，告诉编译器不要对该对象进行任何假定和优化；

比如，把一个数据放到寄存器中而不再对寄存器进行操作，编译器会认为寄存器中的值是不变的，因此在优化过程中可能不会去读这个寄存器而直接告诉你寄存器中的值；

甚至，可以建立const volatile对象，该对象不能为客户程序员改变，但可以被外部改变；

预处理器

2016年7月12日 15:58

预处理器的缺陷

1、宏隐藏了许多难以发现的问题

如：define名称与参数列表间存在空格导致错误、优先级错误……

2、预处理器不允许访问类的成员数据

在C++中，通常使用内联函数而非预处理器，但以下两种情况需要用到预处理器

1. 标志粘贴##

   预处理器中，默认以空格为分隔符，但这样做替换出来的结果也带空格，如果想替换一个单词上的某一段字母就需要用到标志粘贴##

   如“*#define FIELD(a) char* a##_string; int a_size*”，

   其中a和_被##强制分隔开，因此a会被替换；而a_size被作为整体，此处的a不会被替换

   那么“FIELD(one);”将被替换为“char* one_string; int a_size;”

2. 将变量、语句作为字符串输出

   - 如“*#define DEBUG(X) cout << #X " = " << X << endl*”

     其中#X表示把X变量作为一个字符串，

     因此“int a = 0; DEBUG(a);”将输出“a = 0”

   - 又如“*#define TRACE(s) cerr << #s << endl; s*”

     其中#s表示把语句s作为一个字符串，第二个s重申了这个语句，也就是执行语句s，

     因此“TRACE(int a = 0);”将输出字符串“int a = 0”并且执行这个语句！

内联函数

2016年7月12日 16:09

内联函数由编译器控制，是真正意义上的函数，可以在适当的位置像宏一样展开而不需要函数调用的开销；另外，编译器会针对内联函数的声明对参数、返回值等进行检查

类内定义的函数都将自动成为内联函数，而对于类外的函数可以加上inline关键字使其成为内敛函数；

内联函数和普通函数一样，需要声明；

内联一般定义在头文件中，内联代码也会占用一小部分空间，但对于一个比较小的函数，内敛函数的开销要比普通函数少；

一些比较小的函数，甚至构造函数和析构函数也可以作为内联函数放在类内；

访问函数
类似java中的访问器和更改器，定义在类中，作为内联函数，使得外部能够访问私有成员并不会产生压栈、返回等额外开销；
可以通过函数重载来区分访问器和修改器，但往往会将访问器名设为getXxxx，更改器名设为setXxxx；同时访问器往往也设置为const内敛函数；

```
class Rectangle {
  int width, height;
public:
  Rectangle(int w = 0, int h = 0)
    : width(w), height(h) {}
  int getWidth() const { return width; }
  void setWidth(int w) { width = w; }
  int getHeight() const { return height; }
  void setHeight(int h) { height = h; }
};
```

内联函数和编译器

2016年8月21日 15:17

有两种情况编译器不能执行内联

1. 内联函数语句太多，以致编译器认为调用函数的开销更少；
   内联函数中如果出现了任何种类的循环，编译器也不能执行内联
2. 如果企图显式或隐式地取内联函数地址，编译器也不能执行

内联函数需要先声明或定义，才能被引用

对于作为内联函数的析构函数和构造函数，有可能会进一步引用子对象的构造函数或析构函数，从而使得开销更多，甚至超过调用函数

为了减少混乱使类的定义更加简洁，也可以把函数的定义都放到类外，通过显示地声明inline来使其成为内联函数

来自C语言的static

2016年8月21日 16:08

两种语义

1. （存储）对象存放在一个特殊的静态数据区（static data area），而不是放在堆栈上
2. （可见性）指定一个名字是局部的，在外部是不可见的

函数内部的静态变量

- 仅在第一次调用时分配存储单元，函数返回时该存储单元仍有效；
  仅在程序从main()中退出，或exit()来结束程序时，才会销毁该存储单元；
  注意析构函数中不应出现exit()等函数，否则退出程序时会出现无限递归；
  特别的，如果程序是由abort()函数退出的，那么静态对象的析构函数并不会被调用！
- 初始化可以是任意常量表达式，默认初始化为0
- 要注意多线程问题

控制连接

- 一般默认文件作用域（如全局变量、函数）是对所有翻译单元（包括其他文件）是可以见的，称为外部连接
- static可以限制一个名字的可见性，使一个全局变量、函数只在本文件可见，即转变为内部连接
- 在C++中，默认常量、内联函数的名字都是内部连接的

其他存储类型说明符

- auto：局部存储，通常编译器自动从上下文中判断出来，不需要显式说明
- register：寄存器存储，表明该变量经常用到，可以放到寄存器里提高效率；一般不需要显式说明，交由编译器来自动判断是否需要用寄存器存储

名字空间

2016年8月21日 16:42

创建

```
namespace MyLib{
        //...变量定义
}
int main(){}
```

- 只能在全局范围内进行定义
- 可以嵌套定义

```
namespace A{
        using namespace B;
        //...
}
```

- 可以被多次定义，但不是重定义，而是修改添加
- 可以定义别名

```
namespace BobsSuperDuperLibrary{
        //.....
}
namespace Bob = BobsSuperDuperLibrary;
```

- 每个翻译单元可以有一个没有名字的名字空间
  其中的变量都默认为内部连接而在该翻译单元中无限有效

使用

- 作用域运算符

```
namespace X {
  class Y {
    static int i;
  //...
}
int X::Y::i = 9;  //为X名字空间的Y类的静态成员赋值
```

- using指令
  通过using指令表明在该代码块内引入名字空间里的所有名字，但缺点是名字的引用不太直观；
  而且引入的多个名字空间之间可能存在冲突
  如“using namespace std;”
- 使用声明
  通过using指令表明在该代码块内引入名字空间内的某个名字

如“using A::f;”引入A名字空间里的名字f；
在头文件中，通常不直接引入整个名字空间，而是通过使用声明引入一部分名字，避免对.cpp造成污染

C++中的静态成员

2016年8月21日　　17:10

- 通常在头文件中声明，在.cpp中定义（直接赋值，或者通过构造函数），如——

```
//A.h
class A{
        static int i;     //声明
public:
        //...
}
```

```
//A.cpp
int A::i = 1;     //真正的定义
```

- 既然有了声明，就必须有定义，否则会报错
- 定义只能定义一次，只能定义在类的外部而且是在全局范围
- 静态成员是private的，却能够被外部访问；
  然而这样依旧维持着类结构的保护性
  a. 初始化只能在定义时完成
  b. 定义有且只有一次，否则就会出错；保证它是由类的构造者所控制
- 定义时可以使用之前定义过的静态成员所构成的常量表达式

```
int WithStatic::x = 1;
int WithStatic::y = x + 1;      //注意这里的x是类内的静态成员，而不是外部变量
```

- 静态常量在类内部定义，使用限定符“static const”，同时赋予初值；
  但静态常量数组得再类外定义，跟普通的静态成员定义方法一致（在类内声明时可以不指定数组长度）
- 不仅仅是内部类型必须遵循“类内声明，类外定义”，对象也必须这么做（构造函数必须在类外调用）
- 全局类或全局类的嵌套类才有静态成员，
  局部类（如某个函数体内定义的一个类）是不能声明静态成员的
- 静态成员函数
  可以通过static在类内定义成员函数（注意必须是public）；
  静态成员函数为整个类服务，而不是为类的某个对象服务；
  由于静态成员函数没有隐含的this，所以它只能访问静态成员而不能访问一般成员；

```
class A{
        static int i;
        //...
public:
        static int f(){
                //..只能访问静态成员
        }
```

```
}
int A::i = 1;
int main(){
        A a;
        A* ap = &a;
        a.f();          //通过对象调用
        ap->f();        //通过指向对象的指针调用
        A::f();         //直接通过类名用
}
```

- 定义只能创建唯一对象的类

```
#include <iostream>
using namespace std;
class Egg {
  static Egg e;         //创建一个自身的对象作为静态成员（对于一个类来说静态成员只能有一个）
  int i;
  //注意这里把构造函数作为private而不是public！！
  //这样一来外部就不能直接创建Egg类的实例
  Egg(int ii) : i(ii) {}
  Egg(const Egg&);   //拷贝构造函数
public:
  static Egg* instance() { return &e; }                 //通过一个静态成员函数来向外部返回实例
  int val() const { return i; }             //访问器
};
Egg Egg::e(47);                 //初始化这个唯一的对象（静态成员），静态成员的初始化必须在全局范围
int main() {
//!  Egg x(1); // 错误！！构造函数为private，不能被外部调用
  // 使用这个对象：
  // 通过public的instance函数取得这个对象的指针，再通过它的访问器来取得值
  cout << Egg::instance()->val() << endl;
} ///:~
```

静态初始化的相依性

2016年8月22日 12:20

- 静态变量（对象）的初始化顺序
  对于一个翻译单元来说，初始化的顺序参照定义的顺序进行，清除的顺序则相反；
  但对于多个翻译单元来说，初始化的顺序是不确定的，因此可能带来一些问题

```
//1.cpp
extern int y;
int x = y + 1;
```

```
//2.cpp
extern int x;
int y = x + 1;
```

  对于上面这个程序，全局、静态变量默认初始化为0，因此——
  如果1.cpp先翻译，那么初始化的结果为“x = 1, y = 0”；
  如果2.cpp先翻译，那么初始化的结果为“x = 0, y = 1”；
  甚至对于一些无法初始化为0的东西，如一个对象，那么在创建一个包含它的类前如果没有定义过这个对象，那么编译会出错
- 解决方法
  - 避免初始化的相互依赖
  - 将关键的静态对象的定义放在一个文件中
  - 采用两种程序设计技术解决
    - 一种提出比较早，但比较复杂的方法

```
//通过宏命令防止该头文件被同一个文件重复包含
//x和y是待初始化的静态变量
#ifndef INITIALIZER_H
#define INITIALIZER_H
#include <iostream>
//声明静态变量x,y
//如果在外部没找到，则会被初始化为0，这是在程序层面上真正意义的初始化
extern int x;
extern int y;
class Initializer {
 //静态成员，用来记录初始化次数
 //无论有几个文件定义了该类，它们都是同一个类，也就是共用一个initCount
 static int initCount;
public:
 Initializer() {
  //initCount为0时进行“初始化”操作
```

```
  //无论是否执行“初始化”，initCount都会+1
  if(initCount++ == 0) {
   //这是我们所想要的真正的“初始化”环节
    x = 100;
    y = 200;
  }
 }
 ~Initializer() {
  --initCount;
 }
};
//为每个包含该头文件的.cpp创建一个该类的实例，创建时计数都会+1
//等到该文件执行结束，也即生命周期结束，init会被析构，计数自然被-1
//此处用static使得该名字仅由本文件可见，所以文件与文件之间不会冲突
static Initializer init;
#endif
```

只要给所有和x，y相关的.cpp文件中包含这样的头文件即可解决问题

- 借助函数内的静态变量（比较简单）

`//A.h`
`class A{`
`  //...`
`};`
	`//B.h`
	`#include <A.h>`
	`class B{`
	`  A a;`
	`public:`
	`  B(const A& aa) : a(aa){}`
	`};`

`//AStatFun.h`
`#include "A.h"`
`A& a1();`
	`//BStatFun.h`
	`#include "B.h"`
	`B& b1();`

`//AStatFun.cpp`
`#include "AStatFun.h"`
`A& a1() {`
`  static A a;`
`  return a;`
`} ///:~`
	`//BStatFun.cpp`
	`#include "BStatFun.h"`
	`B& b1() {`
	`  static B b( a1() );`
	`  return b;`
	`} ///:~`

通过函数返回实例来访问对象，

函数内定义的是static静态变量，第一次调用时即可完成初始化，且不会发生冲突

替代连接说明

2016年8月22日 14:50

C++编程中如果调用C的库，
C++编译器会把返回值类型、参数类型融入函数名组合成函数的内部名以支持函数重载，
可是C并不会这么做，此时就会出现编译过程中找不到对应函数的情况

为了解决这个问题，可以通过extern来指明连接类型，通知编译器将该函数按照C的方式组织内部名

```
//为单个函数指明连接方式
extern "C" float f(int a, char c);
```

```
//为多个函数之名连接方式
extern "C" {
        float f(int a, char c);
        double d(int a, char c);
}
```

引用

2016年8月22日 14:57

- 引用可以理解为一个奇特的指针，使用时编译器会自动将其解引用，并且强迫它初始化
- 引用既可以直接定义，也可以作为参数，与普通变量无异
- 引用 & 指针
  - 引用被创建时必须进行初始化
  - 引用一旦指向一个对象，就不能改变为另一个对象
  - 不能有NULL引用，必须确保引用的是一块合法的存储空间
- 当一个函数返回一个引用时，要跟指针一样，确保返回的引用是有效的
- 函数的常量引用

  对于函数的参数来说，常量引用非常重要

```
void f(int&);
void g(const int&);
int main(){
        int a = 0;
        //变量作为参数
        f(a);    //a作为引用传参（在函数g内可变）
        g(a);    //a被作为常量引用传参（在函数g能不可变）
        //常量作为参数
        // f(1); //错误！传参时，编译器会为1创建一个临时对象，与int&的声明冲突
        g(1);    //正确，临时对象是一个常量
}
```

- 指针引用

  指针本身也可以被引用，如“int*& i;”定义了一个指针的引用，该指针指向一个int
- 传参方式的选择

  C++的传参方式有：值传递、指针传递、引用传递

  在C++编程中，通常采用引用传递来取代指针传递（更加直观），

  用常量引用传递来取代值传递（不需要调用构造函数、析构函数，只需要将地址简单的出入栈）；

  一般只有引用传递不能满足需要时才考虑值传递、指针传递

拷贝构造函数

2016年8月22日　　21:33

- 解决问题：传统的值拷贝（函数的参数传递、返回值等）是简单的位拷贝，并不会调用创建对象的构造函数
- 拷贝构造函数是一个以同类的常量引用为参数的拷贝函数，
  如“HowMany2(const HowMany2& h) : name(h.name) {}”
- 内部类型、不具备拷贝构造函数的类，在进行值传递时编译器会创建一个默认的拷贝构造函数，但它只是简单的进行位拷贝，不会调用相应的构造函数来创建新对象；
  而具备拷贝构造函数的类，在进行值传递时会调用相应的拷贝构造函数
- 对于多个类组合成的新类，它在值传递时会依次调用各个包含类的（默认）拷贝构造函数；
  组合类——

```
class A{
        //...
};
class B{
        //...
};
class Composite{
        A a;
        B b;
        //...
};
```

- 代替拷贝构造函数的方法
  - 防止值传递
    在类内声明一个私有的拷贝传递函数，
    这样编译器就不会创建默认的拷贝传递函数来进行位拷贝，而是发出错误信息来提示你发生了值传递而且没有拷贝传递函数可用
  - 用常量引用而避免值传递

成员指针

2016年8月23日 16:19

- 指向int的指针并不能直接指向A类里的int类型成员（当然，必须是public成员），这时需要使用成员指针
- 定义、初始化

```
class A{
public:
        int a;
        int b;
        //...
}
//定义一个指向A类内int类型成员的指针
//指针名为pa，初始化为指向A类中的a成员
int A::*pa = &A::a;
```

- 限制

  成员指针只能用来指定某个类中的某个成员，

  不能像普通指针那样执行加减、比较操作
- 成员函数指针

```
class A{
public:
        int f(float a) const{}   //const成员函数
        //...
};
//注意这里的初始化，
//A::f不能直接作为函数地址，仍然需要&取地址操作
int (A::*fp)(float) const = &A::f;
```

基本使用

2016年8月23日 17:40

参数个数

- 全局函数，X元运算符需要X个参数
- 成员函数，X元运算符需要X-1个参数（因为this为隐含参数）

规则

- 几乎所有C运算符都能重载，但不能使用C中没有意义的运算符
- 运算符=只能作为成员函数
- 运算符的优先级、参数个数不可改变
- 参数、返回值可以是同一个类，也可以是不同类

语法

- 和普通函数类似，但是有特殊的函数名
- 函数名规定为“operator@”其中@为运算符，如“operator+”、“operator-”等

a++和++a

- 两者函数名相同，也都是一元运算符，C++中通过附加一个哑元参数来区分它们
- ++a调用函数operator++(a)
- a++调用函数operator++(a, int)
  这里的int是一个哑元参数，函数体中并不需要它，所以定义时不需要给出参数名而是作为一个匿名参数；
  以此来区分两种不同的++

参数和返回值的一般类型

- 如果参数只读不改，那么采用const引用；
  如果是成员函数，还应采用const成员函数
- 返回值一般返回一个新的运算结果的常量对象
- 对于赋值运算符，为了能够支持链式表达式（a=b=c），应该返回一个对象的非常量引用
- 对于逻辑运算符，应当返回一个int，最好是bool

直接返回对象 & 先创建对象后返回

- 直接返回对象

```
return Integer(left.i + right.i);
```

  它将通过构造函数创建一个临时对象并返回
- 先创建对象后返回

```
Integer tmp(left.i + right.i);
return tmp;
```

  创建一个对象tmp，再通过拷贝构造函数创建一个临时对象并返回，最后调用tmp的析构函

数并清除

特殊的运算符重载

- operator[]
  - 必须作为成员函数
  - 只能接受一个参数
  - 通常返回一个引用
- operator()
  - 唯一一个支持任意个参数的运算符重载
- operator,
  - 对于全局函数， 第一个参数对应运算符左边，第二个参数对应运算符右边
  - 对于成员函数，自身对应运算符左边，参数对应运算符右边
- operator->
  - 必须是成员函数
  - 必须返回一个对象、对象的引用或一个指针
    返回的对象（或其引用）必须支持->运算符
- operator->*
  - 必须返回一个对象
    该对象必须支持()运算符

不能重载的运算符

- 成员选择~~operator.~~
- 成员指针间接引用~~operator.*~~
- C++不提供求幂运算符

成员运算符 & 非成员运算符

运算符	建议使用
一元运算符	成员
= () [] -> ->*	必须成员
复合（ += -= /= *= ^= &= |= %= >>= <<= ）	成员
其他二元运算符	非成员

赋值运算符重载

```
MyType b;
MyType a = b;          //a原先不存在，这里的=会调用MyType的拷贝构造函数
a = b;          //a是已经存在的对象，这里的=会调用MyType的operator=函数
```

- 最好要检查自赋值（如果简单的将自己赋值给自己，不做操作而直接返回自己）

  ```
  if(&dog != this){...}
  ```
- 如果对象存在指针成员，最好把指针所指的内容也拷贝一份
- 如果没有显式地声明、定义operator=，编译器将自动创建一个默认的；
  默认的operator=将简单地进行位拷贝，同时，如果类包含对象，那么也会递归的调用相应

的operator=；
如果想禁止对象的赋值操作，那么应当显式地声明一个私有化的operator=

引用计数 & 写拷贝技术

2016年8月27日 17:53

- 引用计数
  计数一个对象自身被引用的次数，注册则引用次数加一，解除引用则次数减1，当引用次数归零时则delete
- 写拷贝技术
  如果一个对象被多个使用者同时使用，要么只能被只读，要么就得被做一份写拷贝再进行修改

```
//省略头文件包含和命名空间的指定
class Dog {
  string nm;      //狗的名字
  int refcount;   //引用次数的记录
  //私有化的构造函数，不允许外部直接创建对象
  //引用次数初始化为1
  Dog(const string& name) : nm(name), refcount(1) {}
  //私有化的赋值运算符重构函数声明
  //只声明不定义，用于禁止外部对Dog类使用赋值运算符
  Dog& operator=(const Dog& rv);
public:
  //静态成员函数，用于创建一个Dog对象
  static Dog* make(const string& name) {
    return new Dog(name);
  }
  //拷贝构造函数
  Dog(const Dog& d)
    : nm(d.nm + " copy"), refcount(1) {}
  ~Dog() {}
  //注册——引用次数+1
  void attach() {
    ++refcount;
  }
  //解除引用——引用次数-1
  //如果解除后引用次数归零，则将对象删除
  void detach() {
    if(--refcount == 0) delete this;
  }
  //取得一个可修改对象（原对象或其副本）
  Dog* unalias() {
    //引用次数为1，说明函数调用者为该对象的唯一引用者
    //则直接返回该对象的指针，允许直接修改
```

```
    if(refcount == 1) return this;
    //否则解除本次引用（引用次数-1）
    //由拷贝构造函数创建一个副本并返回
    --refcount;
    return new Dog(*this);
  }
  //重命名
  void rename(const string& newName) {
    nm = newName;
  }
  //辅助输出的运算符重载
  //作为友元，该函数允许访问Dog类的私有成员
  friend ostream&
  operator<<(ostream& os, const Dog& d) {
    return os << "[" << d.nm << "], rc = "
      << d.refcount;
  }
};
class DogHouse {
  Dog* p;  //指向Dog对象的指针
  string houseName;
public:
  DogHouse(Dog* dog, const string& house)
   : p(dog), houseName(house) {}
  //拷贝构造函数
  DogHouse(const DogHouse& dh)
    : p(dh.p),
      houseName("copy-constructed " + dh.houseName) {
    p->attach();
  }
  DogHouse& operator=(const DogHouse& dh) {
    //自赋值检查
    if(&dh != this) {
      houseName = dh.houseName + " assigned";
      p->detach(); //解除当前引用
      p = dh.p;    //从源DogHouse取得目标引用的指针
      p->attach(); //注册引用（引用次数+1）
    }
    cout << "DogHouse operator= : "
         << *this << endl;
    return *this;
  }
```

```
  ~DogHouse() {
    p->detach();    //解除引用
  }
  void renameHouse(const string& newName) {
    houseName = newName;
  }
  void unalias() { p = p->unalias(); }
  void renameDog(const string& newName) {
    unalias();
    p->rename(newName);
  }
  //取得一个可以修改的Dog（或副本）
  Dog* getDog() {
    unalias();
    return p;
  }
  //辅助输出的运算符重载
  friend ostream&
  operator<<(ostream& os, const DogHouse& dh) {
    return os << "[" << dh.houseName
      << "] contains " << *dh.p;
  }
};
```

自动类型转换

2016年8月27日 17:58

自动类型转换及其规则可由用户自定义；

构造函数转换

- 构造函数转换，即定义一个用于自动转换的构造函数
  该构造函数与拷贝构造函数类似，接受一个常量引用，
  不同的是，该构造函数接受的对象类型不是自身类型，而是其他类型，函数体即为自动转换的规则；
- 当函数调用时，如果没有参数类型匹配的函数定义，那么就会来检查各个类是否存在构造函数转换能够将给出的实参类型转换成目标的形参类型

```
class One {
public:
  One() {}
};
class Two {
public:
  //当函数调用时，如果没有参数类型匹配的函数定义，
  //那么就会来检查目标类是否有对应的转化规则，
  //该函数负责将One类的对象自动转换成Two类的对象
  Two(const One&) {}
};
void f(Two) {}  //函数f接受参数类型为Two类的对象
int main() {
  One one;
  //调用函数f，实参为One类的对象，将自动调用Two类的构造函数转换成One类的对象
  f(one);
} ///:~
```

- 阻止构造函数的自动转换
  有时候需要避免构造函数用于自动类型转换，只需要给构造函数加上explicit修饰符
  如“**explicit** Two(**const** One&) {}”
  但是，此时依旧能够进行手动的“强制类型转换”（实际上是手动调用构造函数创建临时对象）“Two(one)”

运算符转换

- 用operator关键字在类内定义一个将本类对象转换为其他类的运算符重载函数

```
class Four(){
        //...
public:
```

```
        operator Three() const {}
}
```

将Four对象转换为Three对象，函数体即为转换规则

比较

- 构造函数转换，是目的类执行转换（把其他类转换成本类）；
  运算符转换，是源类执行转换（把自己转换成其他类）
- 构造函数转换无法实现自定义类型转换为内置类型（无法为内置类型定义构造函数），而运算符转换可以

自动类型转换：全局运算符重载、成员运算符重载

- 运算符重载可以定义在全局，也可以定义为成员函数
- 通常是将运算符重载定义为全局：
  全局运算符重载在做自动类型转换时，左右操作数都能够进行自动类型转换；
  成员运算符重载却只能为右操作数进行自动类型转换（因为左操作数必须是对应的类型才能调用其中的成员函数——运算符重载）

缺陷

- 构造函数转换、运算符转换的冲突
  如果A存在一个只接受参数B的构造函数，B也存在一个转换为A的运算符重载函数，那么在调用f(B)时，编译器不知道该调用哪一个函数来完成自动类型转换而引起混乱并引发错误
- “扇出”问题
  当A可以自动转换为B和C时，而函数f被重载，既可接收B参数也可以接受C参数，那么调用f(A)时，究竟要将A转换成B还是将A转换成C呢？这会使编译器产生混乱而报错
- 默认的函数
  如果定义了一个构造函数转换或运算符转换，而不显示地定义operator=，那么

  ```
  Fo fo;
  Fee fee = fo;
  ```

  尽管编译器会创建默认的operator=，但此处会进行自动类型转换而不是调用operator=

自定义的自动类型转换有很多潜在的操作问题，它在减少代码、使代码直观上是非常出色的，但使用要非常的谨慎

对象创建

2016年8月29日　　23:25

- 对于库函数malloc和free，编译器无法控制它们，因此也无法确保对象的初始化和清理
- 创建一个对象需要发生两件事
    - 为对象分配内存
        - 三种存储空间
            - 静态存储区域：程序开始前就进行分配，在整个程序的生存周期内都存在
            - 栈空间：在某个程序执行点自动创建，在另一个执行点自动释放，其操作由处理器的内置指令完成，因此需要编译器明确知道需要多少存储单元，以便生成正确的指令
            - 堆空间：在程序的任何执行点、时间点都可以动态的申请内存，申请、清理内存都是自由的
    - 调用构造函数初始化那个内存

运算符new

- 集成内存分配、初始化操作，返回指针

  ```
  MyType *fp = new MyType(1,2);
  ```

  等价于调用malloc分配合适的内存空间，同时调用构造函数MyType(1,2)完成对象的初始化操作
- 如果构造函数没有参数，那么连括号都可以省略，如——

  ```
  MyType *fp = new MyType;
  ```
- new运算符还将自动检查内存分配是否成功

运算符delete

- 先调用析构函数，再清理内存，需要提供对象的地址

  ```
  delete fp;
  ```
- 如果用malloc创建，那么用free清理；
  如果用new创建，那么用delete清理；
  否则，其执行结果是不确定的
- 如果指针的值为0，那么delete不会执行任何操作；
  因此通常在执行delete之后将指针赋值为0防止误操作
- 如果指针类型为void *，那么delete只会释放相应的内存，而不会调用任何析构函数（缺乏类型信息）

内存申请的过程

- 在堆里搜索一块足够大的内存（可能需要经过多次的试探，所以其具体开销是不确定的）
- 记录内存的大小和地址，防止这块内存被重复使用
- 返回指向这块内存的指针

数组

- 创建对象数组

```
MyType* fp = new MyType[100];
```

- 释放内存

  由于创建返回的指针为MyType*类型，所以如果直接delete fp那么只会释放第一个元素的内存；

  此时需要用一个空的方括号告诉编译器（当然也可以指明数组长度，但这是没必要的），这是一个数组

```
delete []fp;
```

- 最好还应创建一个常量指针，使其更像数组（数组名即一个常量指针）

```
MyType const* fp = new MyType[100];
```

  或

```
const MyType* fp = new MyType[100];
```

- 在new一个数组时，实际上系统还为其分配了一部分隐藏的字节，用来记录数组的当前信息，方便对数组的操作

内存耗尽

2016年8月30日 15:57

- 如果new运算符找不到合适大小的内存块，则会调用一个new-handler的特殊函数；new-handler默认抛出一个异常
- 也可以自己定义new-handler的操作
  定义一个无参数、返回void的函数，通过set_new_handler指定操作函数

```
#include <iostream>
#include <cstdlib>
#include <new>                  //new-handler相关的头文件
using namespace std;
int count = 0;
//自定义的new-handler函数
void out_of_memory() {
  cerr << "memory exhausted after " << count
    << " allocations!" << endl;
  exit(1);                  //一般最后都要终止程序或抛出异常
}
int main() {
  set_new_handler(out_of_memory); //设置自定义new-handler函数
  while(1) {
    count++;
    new int[1000]; // Exhausts memory
  }
} ///:~
```

重载new和delete

2016年8月30日　　16:01

- new先分配内存，后调用构造函数；
  delete先调用析构函数，再释放内存；
  其中构造函数、析构函数的调用是程序员不可控的，
  也就是说重载new和delete，只是重载其内存管理功能
- 意义：
  - 解决对象频繁地创建、释放明显影响程序效率的问题
  - 解决堆大小有限，多次创建、释放内存造成碎片化的问题
- 全局重载
  - 重载new
    - 必须接受一个size_t类型的参数，该参数的值由编译器传递给我们，指示着所需分配内存的大小
    - 必须返回指向大于等于所需内存大小的内存起始位置的指针，类型为void*，至于如何转换成目标类型的指针，是编译器的事情
    - 如果找不到存储单元，必须返回0，而且需要做调用new-handler或抛出异常等处理措施
    - 重载函数执行完后随即调用构造函数进行初始化操作
  - 重载delete
    - 析构函数执行完后才会调用该重载函数
    - 必须接受一个void*类型的指针，它指向需要释放的内存的起始地址
    - 返回类型必须是void
  - 如——

```
void* operator new(size_t sz) {
  printf("operator new: %d Bytes\n", sz);
  void* m = malloc(sz);  //调用malloc来申请内存
  if(!m) puts("out of memory");
  return m;
}
void operator delete(void* m) {
  puts("operator delete");
  free(m);   //调用free来释放内存
}
```

    - 重载函数使用printf、puts来打印信息，而不是创建iostream对象
      因为如果创建iostream对象，那么又调用了new，形成死锁
- 类重载
  - 在同时存在全局重载、类重载的情况下，优先调用类重载（即局部优先于全局的原则）
  - 类重载的使用方法与全局重载类似，只不过重载函数放在类内

- 数组版本的重载
  - 如果类内只定义了operator new()或operator delete()，那么当创建一个该类的数组时，它将调用的是全局重载而不是类重载
  - 如果在创建一个类的数组时调用类内重载，那么就要定义一个类重载的数组版本——

```
class A{
public:
        void* operator new[](size_t sz){
                //...
        }
        void operator delete[](void* p){
                //...
        }
}
```

  - 注意这里new的参数sz不再是单个元素的大小，而是整个数组的大小

- 定位new和delete
  - 意义：
    - 在嵌入式开发中，有时候需要在一个确定的内存上动态地创建对象
    - 有时候要利用new，在栈上选择不同的内存分配方案
  - 利用new可以带参数的机制实现

    new函数除了必须带size_t参数之外，还可以带额外的参数

```
#include <cstddef> // Size_t
#include <iostream>
using namespace std;
class X {
  int i;
public:
  X(int ii = 0) : i(ii) {
    cout << "this = " << this << endl;
  }
  ~X() {
    cout << "X::~X(): " << this << endl;
  }
  //带额外参数的new重载
  void* operator new(size_t, void* loc) {
    return loc;
  }
};
int main() {
  int I[10];
  cout << "I = " << I << endl;
  //带参数的new运算符，注意这里的参数的I（即上边创建的数组首地址）而不是1
```

```
  //这里的new直接在栈上已有的10个int的空间上创建了对象X
  //这里的对象是动态创建的，但却在栈上而非堆中
  X* xp = new(l) X(47);
  //由于是在栈上创建的动态对象，所以不能通过delete释放空间
  //如果需要调用它的析构函数来释放掉这个对象（但无法释放空间）
  //可以采取以下这种方式，直接访问它的析构函数
  xp->X::~X();
  //但是要注意，但程序脱离这个程序段，那么这个栈空间将被释放
  //空间被释放时，对象的析构函数又会被调用一次
}
```

- 定位delete
  定位delete一般仅在一个定位new的构造函数出现异常时被调用，并且定位delete将与定位new有一个对应的参数列表

语法与用法

2016年8月30日　　22:18

组合

- 由多个类组合成一个新类
  直接在新类下定义源类的对象即可

继承

```
class X {
  int i;
public:
  X() { i = 0; }
  X(int ii) { i = ii; }
  void set(int ii) { i = ii; }
  int read() const { return i; }
  int permute() { return i = i * 47; }
};
//由已有的类X创建新类，它继承了X的所有成员，称为派生类
//这里继承时使用修饰符public，
//则X的private成员在Y中仍为private，public成员仍为public
//如果使用private，那么X的public成员将变为private
class Y : public X {
  //Y类在X类的基础上又定义了一个i
  //则对于Y类来说，直接访问的i将是自己定义的i，派生类没有权利访问基类的私有域
  int i;
public:
  Y() { i = 0; }
  int change() {
        //派生类没有同名的permute函数，可以直接访问基类X的成员函数
    i = permute();
    return i;
  }
  void set(int ii) {
    i = ii;
        //因为基类和派生类都有set()函数
        //所以派生类中不能直接调用基类的set，而需要指明是X类下的set
    X::set(ii);
  }
};
```

继承的细节

2016年9月2日 14:11

派生类的初始化

- 由于派生类无法直接访问基类的私有域，所以派生类的构造函数并不能直接对它们进行初始化
  但是可以借助初始化列表来对基类对象进行初始化
- 如前边的X、Y类
  Y类为X类的派生类，X类有一个构造函数X(int)，那么就可以用如下方式对Y类对象进行初始化

```
Y::Y(int i1, int i2) : X(i1), i(i2) { }
```

  X(i1)调用了X类的构造函数进行初始化，其次也将自己的私有成员i初始化为i2；
  注意这里对Y类的私有成员i的初始化，是通过一个“为构造函数”实现的，它采用了构造函数的语法但并非真正的构造函数，除了在初始化列表之外，在类外也可以采用这种方式对内部类型进行初始化操作，如——
  “int i = 0;”和“int i(0);”是等价的

派生类的析构

- 对象在其生命周期结束时会自动调用析构函数
- 对象自动调用的析构函数是自己的析构函数，并不会自动地去调用基类的析构函数
- 不过对于组合，“基类”是以一个对象作为“派生类”的私有成员，因此在“派生类”对象生命周期结束时，为了释放成员，“基类”对象的析构函数还是会被调用的

构造函数、析构函数的调用次序

- 当一个对象被创建时，会优先调用基类的构造函数，再去调用成员对象的构造函数，而不是按照初始化列表的顺序
- 析构函数的调用次序与构造函数相反

成员函数重定义对重载的影响

- 如果派生类对基类的某一个成员函数进行重定义，那么基类中被重定义函数的函数名会被隐藏，也就是说，在派生类中无法直接访问该函数名的任何版本（无论该版本是否被重定义）

不能被继承的成员函数

- 构造函数和析构函数
- operator=（其他运算符重载都会被继承）

多重继承

- 多重继承只需要要继承列表中列出多个基类，用逗号隔开
- 但是多重继承会引起很多含糊的可能性，如非十分了解应尽量避免

渐增式开发

- 有时候为了改进一个程序，想为一个类添加一些东西来使用，但又怕影响到其他使用了此类的程序段；此时可以创建一个继承于该类的新类，通过使用新类、不影响旧类的方式来进行渐增式的开发

组合与继承的选择

2016年9月2日 14:11

- 组合表示一种has-a的关系；
  继承表示一种is-a的关系
- 比如，小轿车、车、车门、车轮、车窗：
  车门、车轮、车窗为车的一部分，是has-a的关系，适合用组合；
  小轿车是车的一种，是is-a的关系，当然它也有车门、车轮、车窗，适合用继承；
- 子类型化
  如果想要派生来严格使用基类的接口，也就是创建一个基类的子类，那么用继承更好
- 私有继承private
  - 如果使用私有继承，那么基类的所有成员都会变成私有化，用户将不能通过派生类的对象直接访问基类的成员，此时这个对象就不能视为基类的一个实例；
  - 通常如果能使用组合，是不会使用私有继承的，这不易理解
  - 如果想要使私有继承的部分成员公有化

```
class Pet {
public:
  char eat() const { return 'a'; }
  int speak() const { return 2; }
  float sleep() const { return 3.0; }
  float sleep(int) const { return 4.0; }
};
class Goldfish : Pet { //继承时默认为private
public:
  using Pet::eat; //将其中的eat成员公有化
  using Pet::sleep; //将其中的sleep成员（包括两个载版本）公有化
};
```

如果需要将基类的部分成员隐藏起来，则可以使用这种方法

protected

2016年9月2日 14:00

- 将基类成员定义为protected，则派生类内部可以访问这些成员，但用户不能访问

```
class Base {  //Derived的基类
  int i;
protected:  //两个protected成员函数
  //这两个函数可以被派生类Derived内部调用
  //但不可以在类外部调用，也就是一个特殊的private
  int read() const { return i; }
  void set(int ii) { i = ii; }
public:
  Base(int ii = 0) : i(ii) {}
  int value(int m) const { return m*i; }
};
class Derived : public Base {   //Derived是Base的派生类，且为公有继承
  int j;
public:
  Derived(int jj = 0) : j(jj) {}
  void change(int x) { set(x); }  //在派生类内可以调用派生类的protected成员
};
```

- 保护继承
  继承既可以是private、public，也可以是protected的，但是这非常少用，只是为了确保语言的完备性；

向上类型转换

2016年9月2日 14:25

- 派生类的对象是可以向上转换为基类的
- 意义：
  a. 既然小轿车是车，那么小轿车理应具备车的所有特点，自然可以作为车来使用
  b. 基类是派生类的一般情况，基类的成员也是派生类的一部分，因此向上转换时安全的
- 转换后多余的成员会被抛弃，只留下基类具有的成员
- 派生类如果要自定义拷贝构造函数，要记得在初始化列表中调用基类的拷贝构造函数

```
Child(const Child& c)
        : Parent(c), i(c.i), m(c.m){}
```

本来Parent拷贝构造函数只接受类型const Parent&的，但是类型可以向上转换，Child类型的c可以向上转换为Parent从而成功的调用Parent拷贝构造函数，此时c中由Child从Parent继承来的成员都可以被该构造函数完成初始化，而Child类自己定义的额外的成员如i、m就需要另外初始化了

- 组合和继承
  组合不能向上类型转换，继承则可以；
  这也是在选择组合和继承时可以考虑的角度
- 除了函数调用之外，指针、引用也会发生向上类型转换

```
Child c;
Parent& pr = c;                 //pr将是Parent的引用，只能访问Parent的成员
Parent* pp = &c;        //pp将是指向Parent的指针，只能访问Parent的成员
```

pr、pp将不知道自己指向的是一个Child

基础

2016年9月2日 14:49

函数调用捆绑

- 捆绑：函数体和函数调用相联系
- 传统的早捆绑：捆绑在程序运行之前，由编译器和连接器完成，这是C语言仅有的一种函数调用方式
- 晚捆绑（动态捆绑）：捆绑在程序运行时

虚函数

- 问题

```
class Parent{
public:
	void play(){}
};
class Child : public Parent{
public:
	void play(){}
};
void f(Parent& i){
	i.play();
}
int main(){
	Child c;
	f(c);
}
```

  - 基类Parent、派生类Child都定义了成员函数play
  - 函数f接受Parent对象，当传入的参数为Child对象时，将向上转换为Parent，函数f内会调用Parent的play函数而不是Child函数
- 虚函数
  基类的成员函数添加修饰符virtual声明为虚函数，派生类则无需virtual修饰符（加上了也不会报错，但不易理解）；
  此时，该函数为动态捆绑，当派生类的对象的一个基类引用调用一个基类、派生类都定义的成员函数时，将调用的是派生类的版本而非基类的版本；
  如上一点的示例程序中，f(c)调用的是Child版本而非Child版本；
  如果是通过基类指针调用基类对象的虚函数，则不会发生晚捆绑

- C++的实现机制（典型的编译器）
  - 对每个包含virtual函数的类创建一个表VTABLE，VTABLE中放置特定类的虚函数地址
  - 每个带虚函数的类中秘密地放一个指针VPTR，指向该对象的VTABLE

        - 该指针通常放在对象的开头
        - 无论有几个虚函数，都只插入一个void指针VPTR
        - C++要求每个对象都必须有长度（以便分配一个明确的地址），如果没有定义类数据成员，那么编译器会自动插入一个“哑”成员，但如果有虚函数，即使没有显式地定义数据成员，也会因为有指针VPTR而占用“哑”成员的位置
        - VPTR的初始化发生在构造函数中
    - 当通过基类指针调用虚函数时，编译器插入一段代码，能够取得这个VPTR并从表中查找相应的函数地址
    - 虚函数虽然会在空间（内存、代码）、时间上有一些开销，但这个开销不会很大；通常在初次开发时放心大胆地用虚函数来实现多态，当性能出现瓶颈时再来考虑哪些虚函数是不必要的
- 运算符重载
  运算符重载函数也可以是虚的，这种情况会比较复杂，一般只有在解决一些数学问题的时候才会使用（多个数学系统间的运算转换）
- 向下类型转换
  通常向下类型转换的安全是不能保证的，因此应当尽可能避免（如采用多态等）；
  但C++也支持向下类型这一操作
- 借助dynamic_cast进行显示类型转换，该转换是安全的，它会根据对象的VTABLE判定是否能够向下转换。如果转换成功将返回指向目标类型的指针，否则返回0；

```
class Pet { public: virtual ~Pet(){}};
//Dog、Cat是Pet的派生类
class Dog : public Pet {};
class Cat : public Pet {};
int main() {
  Pet* b = new Cat; //将本质为Cat对象的b向上转换为Pet
  Dog* d1 = dynamic_cast<Dog*>(b);       //尝试将b向下转换为Dog，失败，返回0
  Cat* d2 = dynamic_cast<Cat*>(b);       //尝试将b向下转换为Cat，成功，返回Cat指针
}
```

  dynamic_cast需要判断转换是否允许，因此有一定的开销，但也不多；
  - 如果明确知道正在处理的对象类型，可以直接使用static_cast以减小开销；
    但static_cast不会替换前进行判断，如果出现错误将会报错~
    与传统的类型转换比较，static_cast则不允许将类型转换到无关的其他类上去，因此它更加安全
  - 运行时类型识别机制（RTTI）
    可以借助typeid关键字来检测指针的实际类型（typeid在头文件<typeinfo>中声明）

```
#include <iostream>
#include <typeinfo>  //typeid
using namespace std;
class Shape { public: virtual ~Shape() {}; };
class Circle : public Shape {};
```

```
class Square : public Shape {};
class Other {};
int main() {
  Circle c;
  Shape* s = &c; // Upcast
  s = static_cast<Shape*>(&c);  //显式的向上转换，正确但无必要，甚至会引起误解
  Circle* cp = 0;
  Square* sp = 0;
  //先检测指针，再向下转换，其实就类似于dynamic_cast
  if(typeid(s) == typeid(cp)) //如果s是Circle*
    cp = static_cast<Circle*>(s);
  if(typeid(s) == typeid(sp)) //如果s是Square*
    sp = static_cast<Square*>(s);
  if(cp != 0)
    cout << "It's a circle!" << endl;
  if(sp != 0)
    cout << "It's a square!" << endl;
  // 错误！Other* op = static_cast<Other*>(s);
  // static_cast不允许向无关的其他类转换
  Other* op2 = (Other*)s; // 但传统的类型转换是允许的（不安全）
} ///:~
```

抽象基类和纯虚函数

2016年9月4日 12:46

- 语法

```
class A{
public:
        virtual void f() = 0;
        virtual void g() = 0;
};
```

    - 纯虚函数：只声明而不定义的虚函数，并在末尾追加“=0”
    - 抽象类：包含纯虚函数的类（如果该类只有纯虚函数，那么就是纯抽象类）
- 抽象类一般作为接口类
- 编译器将为纯虚函数在VTABLE中保留一个位置，但由于抽象类的VTABLE不完全，不能定义一个抽象类的实例
- 抽象类的派生类一般需要给出所有纯虚函数的定义，否则该派生类也会作为抽象类
- 纯虚函数在类内只能声明，不可定义！
  但有时候我们希望把一些虚函数改造为纯虚函数而不做过多的代码改动——
  （只需要把类内虚函数的函数体移动到类外，然后在派生类下定义同名函数来直接调用基类的纯虚函数）

```
class Pet {
public:
  virtual void eat() const = 0;
  // 不可以直接在类内对纯虚函数作出定义
  //!  virtual void sleep() const = 0 {}
};
// 但是可以在类外对类内的纯虚函数进行定义
void Pet::eat() const {
  //...
}
class Dog : public Pet {
public:
  // 在派生类中定义一个同名函数，函数体内则直接对基类的纯虚函数（已定义）进行调用
  void eat() const { Pet::eat(); }
};
```

继承与虚函数

2016年9月5日 21:38

- 继承的虚函数机制
  - 编译器会为新类创建一个新的VTABLE
  - 如果是重写了基类的虚函数，那么该函数地址在VTABLE中的位置会与基类的VTABLE中对应函数地址对应
  - 值传递 & 地址传递
    如果函数采用值传递的方式传参，编译器会对派生类对象进行切片处理，将其转换为基类对象，此时再调用虚函数，由于函数内的对象的本质确实是基类，所以调用的是基类版本而不会是派生类版本；
    而地址传递，不会强制对对象进行转换，对象内部隐藏着类型信息，可以正确地调用合适的函数；
    除此之外，如果是纯虚函数，那么编译器会及时给出报错（抽象类不能创建实例）
- 重载与重定义
  虚函数和普通成员函数一样，如果重定义了一个函数，那么该函数在基类中的其他版本都会被隐藏；
  如果派生类向上转换成了基类，则派生类版本失效，只能使用基类版本；

```
class Parent{
public:
        virtual int f(){}
        virtual void f(){}
};
class Child : public Parent{
public:
        int f(int){}
        //即使定义了一个基类所没有的版本，基类的各个版本依会被隐藏
}
```

  - 一般重载函数，只能修改参数类型和数量，不能改返回值的类型；
    否则编译器会搞不清楚该调用哪个版本；
    但有个特例：派生类重载基类函数时，如果返回类型是指针或引用，派生类中的返回类型可以是基类中的返回类型的派生类型，如——

```
class Food{
        //...
};
class Pet{
public:
        virtual Food* eat() = 0;        //纯虚函数返回指向Food的指针
};
class Cat : public Pet{
```

```
public:
        class CatFood : public Food{
                //...
        };
        CatFood* eat(){}        //重载时改变了返回类型，返回CatFood类型
}
```

构造函数与虚函数

2016年9月6日 21:18

- 对于包含虚函数的类，创建对象时调用的构造函数需要初始化VPTR、检查this（避免operator new返回零）、调用基类构造函数，这都是隐藏操作，有可能会使得一个看起来似乎很小的内联函数却有比较大的开销，因此在提高效率时应当注意
- 构造函数的调用顺序
  先调用基类构造函数，再调用派生类的
- 在构造函数中调用虚函数，只能调用这个函数的本地版本
  - 构造函数中只能确定基类已经被初始化而不能确保其他成员也完全初始化，即该对象可能未完全初始化，此时调用函数，可能会出现操作未被初始化的成员
  - 构造函数中，初始化的VPTR只能是指向当前类，而无法得知这个类是否基于其他类；VPTR的状态是由最后被调用的构造函数所确定的
  - 对于许多编译器来说，在构造函数中使用虚函数会直接作为早捆绑
- 构造函数不能是虚函数

析构函数和虚拟析构函数

2016年9月6日 21:48

- 析构函数可以而且常常是虚函数
- 析构函数的调用顺序与构造函数相反；
  它可以知道自己派生于哪些类，但不能知道自己派生了那些类
- 如果对一个基类引用进行delete操作，它将调用基类的析构函数而不是真正准确的析构函数；
  解决这个问题可以借助虚拟析构函数，即将基类的虚构函数声明为虚函数
- 纯虚析构函数
  析构函数不止可以是虚的，还可以是纯虚的；
  纯虚析构函数的唯一作用是——阻止基类创建实例；
  但它比较特殊，要求纯虚析构函数必须有定义函数体（在类外定义）；
  类是必须要有函数体的，既然阻止编译器创建默认析构函数，就必须提供一个完整的析构函数；
  即使抽象类不能创建对象，但是派生类析构之后紧接着也会调用基类的析构函数，也就是说抽象类的析构函数是有必要的！
- 虚拟析构函数中，虚机制被忽略，函数体中调用的任何成员函数都是本地版本；
  析构函数的调用顺序是——从派生类到基类；
  既然派生类的析构函数已经调用，那一部分已经被释放，那么调用派生类版本的成员函数也就变得没有意义
- 单根继承
  - 使得容器内的所有对象都是由统一的一个基类继承而来，于是用一个基类指针即可正常的处理他们（delete时也能正确的调用析构函数，如果基类析构函数为虚函数的话）
  - 大多数面向对象的语言都会使所有类型都继承于一个公共基类（如Python的Object类），但C++认为这样会引起太多的开销，所以没有使用它
  - 可以定义一个公共基类Object，它只有一个纯虚析构函数（防止Object创建实例），各个类都继承于它（因为Object类结构非常简单，所以一般多重继承也不会有什么问题）；然后使指向容器对象的指针都设为Object*，这样即使delete也能正常的调用正确的析构函数了

语法

2016年9月7日 16:22

- 模板的思想与java的泛型类似

```
template<class T>  //注意没有分号，它是class的一个修饰符，声明下边的T为一个未指明的类型
class Array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index) {
    return A[index];
  }
};
int main() {
  Array<int> ia;   //创建一个Array类的实例，而且表明这个实例中的T的实际类型为int
  Array<float> fa; //实际类型为float
  //...
}
```

编译器会生成两个Array的类（比如Array_int和Array_float）来处理ia和fa

- 如果成员函数非内联，则在类外定义成员函数前也要加上“**template<class T>**”
- 模板通常是放在头文件中

这不违背头文件不要放置分配存储空间的任何东西的规则，

因为模板的定义并不分配存储空间，只有在模板被使用时才会相应的分配出存储空间来

- 模板的参数除了可以是类定义外，还可以是一个常量

```
template<class T, int size = 100> //定义了一个size整型常量
class Array {
  T array[size];
public:
  T& operator[](int index) {
    return array[index];
  }
  int length() const { return size; }
};
```

```
//懒惰初始化~
//初始化不发生在构造函数中，而是在第一次使用时初始化
//使用场合：需要创建大量的对象，但不访问每一个对象
template<class T, int size = 20>
class Holder {
```

```
  Array<T, size>* np;
public:
  Holder() : np(0) {}
  T& operator[](int i) {
    if(!np) np = new Array<T, size>;
    return np->operator[](i);
  }
  int length() const { return size; }
  ~Holder() { delete np; }
};
```

迭代器

2016年9月7日 21:06

- 迭代器体积非常小，一般就一个指针
- 泛型、带迭代器的栈结构

```
template<class T, int ssize = 100>
class StackTemplate {
  T stack[ssize]; // 用一个数组来存放栈数据
  int top;  // 栈顶序号
public:
  StackTemplate() : top(0) {}
  void push(const T& i) {
    require(top < ssize, "Too many push()es");
    stack[top++] = i;
  }
  T pop() {
    require(top > 0, "Too many pop()s");
    return stack[--top];
  }
  class iterator; // 声明一个迭代器
  friend class iterator; // 将迭代器声明为友元
                         // 注意是friend class iterator不是friend iterator
  class iterator { // 迭代器的定义
    StackTemplate& s;   // 栈的引用
    int index;  // 栈指针（序号）
  public:
    iterator(StackTemplate& st): s(st),index(0){}
    // 用于创建作为“尾部哨兵”的迭代器
    iterator(StackTemplate& st, bool)
      : s(st), index(s.top) {}
    T operator*() const { return s.stack[index];} // 解引用
    T operator++() { // 前缀自增
      require(index < s.top,
        "iterator moved out of range");
      return s.stack[++index];
    }
    T operator++(int) { // 后缀自增
      require(index < s.top,
        "iterator moved out of range");
      return s.stack[index++];
```

```
    }
    // 栈指针跳转
    iterator& operator+=(int amount) {
      require(index + amount < s.top,
        " StackTemplate::iterator::operator+=() "
        "tried to move out of bounds");
      index += amount;
      return *this;
    }
    // 比较两个栈序号是否相等
    bool operator==(const iterator& rv) const {
      return index == rv.index;
    }
    bool operator!=(const iterator& rv) const {
      return index != rv.index;
    }
    // 输出栈元素
    friend std::ostream& operator<<(
      std::ostream& os, const iterator& it) {
      return os << *it;
    }
  };
  iterator begin() { return iterator(*this); }
  // 取得“尾部哨兵”
  iterator end() { return iterator(*this, true);}
};
#endif // ITERSTACKTEMPLATE_H ///:~
```

```
// 使用方法——
int main(){
        StackTemplate<int> is;        // 创建一个int栈
        // ...为is压入一系列数据
        StackTemplate<int>::iterator it = is.begin();        //从栈的起始位置开始迭代
        while(it != is.end())    //迭代直至栈的尾部
                cout << it++ << endl;
}
```