C++ Primer 手记

作者：devcui
原文链接：https://ld246.com/article/1618325065990
来源网站：链滴

1.数据类型

1.1 类型/字宽

- 1.bool
- 2.char 8bit
- wchar_t 16bit
- char16_t 16bit
- char32_t 32bit
- short 16bit
- int 16bit
- long 32bit
- long long 64bit
- float 6位有效数字
- double 10位有效数字
- long doubel 10位有效数字
- 可寻址最小内存块为 “字节” 8bit = 1byte
- 存储的基本单元为“字” 1 word = 32/64bit = 4/8bytes
- float = 1 word = 32bit = 4bytes
- double = 2 word = 64bit = 8bytes
- long double = 3word - 4word = 96-128 bit = 12 - 16 bytes
- signed +/-/0
- unsigned +/0
- int,short,long,long long -> singed
- 8bit unsigned char 00000000 - 11111111 : 0-255
- 8bit signed char 1111 1111 - 0111 1111: -127-127

1.2 类型转换

- bool b = 0 /false
- bool b = !0的数值 /true
- int i = 1; i= 3.14; /i的值为3
- i = 3.14;double pi = i; /pi的值为3.00
- unsigned char c = -1; /c的值为255
- signed char c2 = 256;/ 未定义的值

1.3字面量

- 20十进制,024八进制,0x14十六进制
- -42的字面值是42，-为取负运算，不计入存储
- 'a'为char的字面值,"asdf"为'a','s','d','f',' \0'(空字符)组成的数组
- 不可打印\n\t\a\v\b\?\r\f 转译序列标志为,
译序列被当作一个字符
- \1234 = \123 4 两个字符
- \x1234 = 一个字符
- L'a' 为宽字符,wchar_t
- u8"hi!" 为utf-8字面值
- 42ULL = unsigned long long
- 1E-3F float
- 3.14159L long double
- nullptr为指针字面量

字符前缀

- u unicode 16字符 char16_t
- U unicode 32字符 char32_t
- L 宽字符 wchar_t
- u8 utf-8 char

整形后缀

- u or U unsigned
- l or L long
- ll or LL long long

浮点后缀

- f or F float
- l or L long double

1.4变量

- 要素，类型说明符，初始值，初始化，列表初始化，默认初始化，不被初始化
- int a=0;int a={0};int a{0};int a(0);
- long double ld = 3.14;int c(ld),d = ld;

1.5声明/定义

- int i 声明
- int i = 1 定义

statically typed 静态类型语言： 编译阶段检查类型，检查过程为type checking

1.6 作用域

- 函数外 全局
- extern 外部
- 函数内 块儿作用域
- 嵌套 内外层作用域

1.7 复合类型

基本数据类型+声明符

1.8 引用类型

- lvalue reference
- rvalue reference
- int ival = 1024; int &refVal = ival; 将refVal指向ival

1.9 指针类型

- 元素的地址
- int ival = 42; int *p = &ival; 将p指向变量ival
- 四种: 指向对象，指向紧邻对象下一个位置，空指针，无效指针

```
int ival = 42;
int *p = &ival;
cout << *p;
```

指向指针的指针

```
int ival = 1024;
int *pi = &ival;
// 指向指针的指针，*以此类推
int **p = *p
```

指向指针的引用

```
int i= 42;
int *p;
int *&r = p;

r = &i;
*r = 0;
```

1.10限定符

- const

- const int ci = 1024; const int &r1 = ci; 正确。 int &r2 = ci 错误，非常量引用指向常量
- const pointer 必须初始化，无法改变指向地址
- top-level const : int *const p1 = &i; const int p1;
- low-level const: const int *p2 = &i;
- top and low : const int *const p1 = &i;

一个对象/表达式是不是常量表达式由他的数据类型和初始值决定

```
const int max_files = 20; max_files为常量表达式
const int limit = maxfiles + 1 ; limit是常量表达式
int staff_size =27; staff_size 不是常量表达式
const int sz = get_size(): sz不是常量表达式
```

constexpr 由编译器检测是不是常量表达式

```
constexpr int mf = 20; 是
constexpr int limit = mf+1; 是
constexpr int sz = size(); 当size为constexpr函数时才是正确的声明语句
```

1.11 类型别名

- typedef double ahaha; ahaha a = 3.14;
- using ahaha = obj;
- typedef char *pstring;pstring 为char *

1.12 auto 自己分析吧

- auto item = val+val1
- auto 忽略顶层const,需要显示声明 const auto f =ci;

1.13 decltype 推断数据类型

- decltype(f()) sum = x;
- const int ci = 0; decltype(ci) a = 1; a类型为 const int;
- int i=0;decltype(i) 为int; decltype((i))为 int&

1.14 结构体

```
struct Sales_data(
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
); 不要忘记加分号
```

2.字符串，向量，数组

2.1 using

using namespace::name

直接访问命名空间的属性/函数/值?

```
#include <iostream>

using std::cin;
using std::out;

int main(){
    int i;
    cin >> i;
    cout << i;
    return 0;
}
```

2.2 string

```
#include <string>
using std::string
```

- string s1;
- string s2 = s1;
- string s3 = "hiya";
- string s4(10,"c"); "ccccccccccc"

直接初始化和拷贝初始化

- = 为拷贝初始化
- != 为直接初始化
- string s5 = "hiya" 拷贝初始化
- string s6("hiya") 直接初始化
- string s7(10,"c")直接初始化

string的操作

- os << s 写入流
- is >> s 从流读出
- getline(is,s) 从is流获取一行
- s.empty() 判空
- s.size() 长度
- s[n] 第n个char的引用
- s1+s2 连接
- s1 = s2 常规赋值

- s1 == s2 比较是否一样
- 大于小于大于等于小于等于，字典顺序比较字符

读写string

```
int main(){
    string s;
    // 读
    cin >> s;
    // 写
    cout << s << endl;
    return 0;
}
```

```
string s1,s2;
// 第一个输入扔到s1
// 第二个输入扔到s2
cin >> s1 >> s2;
// 输出两个字符串
cout << s1 << s2 << endl;
```

```
string word;
// 无限读,读到文件末尾
while(cin >> word){

};
-------
string line;
// 一次读一行
while(getline(cin,line)){
};
```

string::size_type

- size返回的是string::size_type类型的值,无符号类型的值

比较string对象

- >/>=
- </<=
- ==
- !=

处理string中的字符

```
#include <cctype>

int main(){
  // 是否为标点符号
  ispunct()
  // 是否为数字/字母
  isalnum()
```

```
  // 是否为字母
  isalpha()
  // 是否为控制字符
  iscntrl()
  // 是否为数字
  isdigit()
  // 不是空格但可打印
  isgraph()
  // 是否为小写字母
  islower()
  // 是否为可打印字符
  isprint()
  // 空白
  isspace()
  // 是否为大写字母
  isupper()
  // 十六进制
  isxdigit()
  // 大转小，小转大
  tolower()
  toupper()

  return 0;
}
```

处理字符串

for(declaration: expression){

statement;

}

```
string str("some string");
// 循环字符串
for(char c:str){
  cout << c << endl;
}

// 统计标点符号
string s("Hello World!!!");
decltype(s.size()) punct_count = 0;
for(char c:s){
  if(ispunct(c)){
    punct_count++;
  }
}
cout << punct_count << endl;
```

改变字符串

```
string s("asdf");
// 这里c为s中每个char的引用，所以改变c会影响s
for(char &c:s){
  c = touuper(s);
}
```

下标访问

- string s("asdf"); s[0] == 'a',s[s.size()-1] = 'f';

比较另类的迭代方式

```
// 声明index,index不为最后一位，且index不为空白字符
for(decltype(s.size()) index =0;index != s.size() && !isspace(s[index]);++index){
  ...
}
```

2.3 vector

容器

```
#include <vector>
using std::vector;
```

c++既有class template也有 function template,vector属于class template

- vector <int> ivec;
- vector <char> asdf;
- vector <string> zxcv;
- vector <T> exp(value)
- vector <T> exp = {v1,v2,v3}
- vector <T> exp(n,val)
- vector 元素类型为内置类型,比如int,自动初始化为0
- 如果vector对象中的元素类型不支持默认初始化,必须提供初始值
- push_back()末尾添加
- empty()判空
- size()元素个数，不是长度

```
vector<int> a = {1,2,3,4,5,6}
for(int &i:a){
  a = a*a;
}
for(auto i:a){
  // 1,4,9,16,25,36
  cout << i << endl;
}
```

demo:读入一组整数,存入,将每对相邻的整数的和输出

```
vector<int> numbers;
int currentNumber;
while(cin >> currentNumber){
  if(isdigit(currentNumber)){
    numbers.push_back(currentNumber);
  }
```

```
}

for(decltype(numbers.size()) index = 0;index < numbers.size();++index){
 if(index == 0) {
  cout << numbers[index] + numbers[index+1] << endl;
 } else if(index == numbers.size() - 1){
  cout << numbers[index] + numbers[index - 1] << endl;
 } else{
  cout << numbers[index] + numbers[index - 1] + numbers[index + 1] << endl;
 }
}
```

demo2 第一个元素和最后一个元素的和，第二个元素和倒数第二个元素的和

粗略写一下

```
vector<int> numbers;
int currentNumber;
while(cin >> currentNumber){
  if(isdigit(currentNumber)){
    numbers.push_back(currentNumber);
  }
}

if(index%2==0){
// 0-5, 1-4, 2-3
cout << numbers[index] + numbers[numbers.size() - 1 - index] << endl;
}

if(index%2==1){
  numbers[index+1] == numbers[numbers.size() - 1 - index] break
  cout << numbers[index] + numbers[numbers.size() - 1 - index] << endl;

}
```

2.4 迭代器

- *iter 迭代器元素的引用
- iter->mem 解引用iter并获取元素的名为mem的成员，等价于(*iter).mem
- ++iter 下一个元素
- --iter 上一个元素

```
---
string s("some string");
if(s.begin() != s.end()){
  auto it = s.begin();
  *it = toupper(*it);
}
---
for( auto it = s.begin();it != s.end() && !isspace(*it);++it){...}
```

范型编程

- vector <int>::iterator it; it只能读写vector<int>元素
- string::iterator it2; it只能读写string对象中的字符
- vector <int>::const_iterator it3; it3只能读int元素不能写元素
- string::const_iterator it4;只能读字符

```
const vector<int> cv;
auto it1 = v.begin(); it1 类型为 vector<int>::iterator
auto it2 = cv.begin(): it2类型为 vector<int>::const_iterator
// cbegin,cend-> const beigin,const end
auto it3 = v.cbegin(); it3类型为 vector<int>::const_iterator
```

结合解引用

- (*it).empty()
- it->empty() 等价于 (*it).empty()

```
for( auto it = text.cbegin();it != text.cend() && !it->empty();it++){
  cout << *it <<endl;
}
```

运算

- iter1 - iter2 ； 两个迭代指示的距离,类型为difference_type的带符号整形
- iter1 + n ; 移动n个位置
- iter1 += n; iter1 +n的结果赋值给iter1
- iter1.begin() + xxx.size()/2 ; 指示到中间位置

二分搜索

```
// 开始结束
auto beg = text.begin(), end = text.end();
// 中间点
auto mid = text.begin() + (end - beg)/2;
// sought为需要找的数
while(mid != end && *mid != sought){
  // 前半部分?
  if(sought < *mid){
    // 忽略后半部分
    end = mid;
  }else{
    // 忽略前半部分
    beg = mid +1;
  }
  // 拿到新的中间点
  mid = beg +(end-beg)/2;
}
```

2.5 数组

- int a[10] = {xxxxxxx}

- string a1[3] = {"asdf","asd","as"}
- int a2[] = {}
- char c[] = {'','\0'}
- 不允许拷贝值
- int *ptrs[10] 10个指针的数组
- int (*Parray)[10] = &array; Parray 指向一个含有10个整数的数组
- int (&arrRef)[10] = arr; arrRef引用一个含有10个元素的数组arr
- int *(&arry)[10] = ptrs; arry是数组的引用,该数组含有10个指针,arry是一个含有10个int型指针的组的引用

访问数组元素

- 使用数组下标时，定义为size_t类型,与及其相关的无符号类型

```
---
unsigned scores[11] ={};
unsgined  grade;
while(cin >> grade){
  if(grade <= 100){
    // 将当前分数段的计算数值+1
    ++scores[grade/10];
  }
}
---
for(auto i:scores){
  cout << i << endl;
}
```

指针和数组

```
string nums[] = {"one","two","three"};
// "one"
string *p = &nums[0];
// "two"
*p+1
---
int ia[] = {0,1,2,4,5}
auto ia2(ia); // ia2为整形指针，指向ia第一个元素
---
int arr[] = {0,1,2,3}
int *p = arr;
// 指针也是迭代器
++p;
```

标准库begin,end

```
int ia[] = {0,1,2,3,4};
// 首元素指针
int *beg = begin(ia);
// 尾元素指针
int *last = end(ia);
```

```
int *pbeg = begin(arr),*pend=end(arr);
// 迭代demo
while(pbeg != pend && pbeg >0){
  ++pbeg;
}
```

指针运算

int arr[] = {1,2,3,4,....};

int *p = &arr[0];

- p + 1 等价 &arr[1];
- *(p+1) 等价 arr[1];
- *(p+1)+1 等价 arr[1] + 1;

c标准库string函数

- strlen(p) length
- strcmp(p1,p2) compare
- strcat(p1,p2) concat
- strpy(p1,p2) copy

与旧代码的接口

- string s("hello world"); char *str = s; const char *str = s.c_str(); c_str为c风格字符串

2.6 多维数组

```
// 3行4列
int ia[3][4] = {
  {1,1,1,1},
  {1,1,1,1},
  {1,1,1,1},
};
// 初始化为0
int ia[3][4] ={0};
// 初始化行首
int ia[3][4] ={{1},{2},{3}};
// 初始化第一行
int ia[3][4] = {1,2,3,4}
// 多循环处理多维数据
size_t cnt = 0;
for(auto &row:ia){
  for(auto &col:row){
    cout << ia[row][col] << endl;
  }
}
```

指针和多维数组

```
int ia[3][4];
int (*p)[4]  = ia; p指向有4个整数的数组
p = &ia[2]; p指向第二行的四个元素
// p 指向ia的首行4个元素
for(auto p=ia;p!=ia+4;++p){
  //  *p+4为该行第4个元素,q指向一行，也就是指向了p这一行的第一列元素
  for(auto q = *p;q!=*p+4;++q){
    cout << q << endl;
  }
}
---
for(auto p = begin(ia);p!= end(ia);++p){
  for(auto q = begin(*p);q!=end(*p);++q){
    ....
  }
}
```

#3 运算符

3.1一些概念

- 一元运算符,unary operator
- 二元运算符,binary operator
- 组合运算符和运算对象
- 运算对象转换 状态提升
- 重载运算符 overlaoded operator
- 左值右值
- 复合表达式
- 括号无视优先级与结合率
- 优先级与结合率的影响
- 求值顺序

3.2算数运算符

- +,-,*,%

3.3逻辑和关系运算符

- !,<,<=,>,>,==,!=,&&,||

3.4复合运算符

- +=,-=,*=,/=,%=,<<=,>>=,&=,^=,|=

3.5递增递减

- ++
- --

3.6成员访问运算符

```
string s1 = "a string",*p = &s1;
atuo n = s1.size();
n = (*p).size();
n = p->size();
```

3.7条件运算符

- cond?exp1:exp2;

3.8嵌套条件运算符

```
finalgrade = (grade > 30)?" high pass": ((grade < 60) ? "fail" : "pass");
```

3.9在输出表达式中使用条件运算符

```
cout<< ((grade < 60)?"fail":"pass");
```

3.10位运算符

- -位求反 1变0，0变1 ;11=00,00=11
- << 左移
- >> 右移
- & 位与 ;两个都为1则位1，否则为0; 11=1,10=0,00=0;
- ^ 位异或 ;两个有且仅有一个为1则为1，否则为0; 11=0,10=1,00=0;
- | 位或 ;至少有一个为1则为1,否则为0; 10=1;00=0;11=1;

3.11sizeof

sizeof 返回的是表达式结果类型的大小;

3.12逗号运算符

```
vector<int>::size_type cnt = ivec.size();
for(vector(int)::size_type ix =0;ix!=ivec.size();++ix,--cnt){
  ivec[ix] = cnt;
}
```

3.13类型转换

```
// double -> int
// 隐式转换
```

```
int ival = 3.541 + 3;
```

3.14算数转换

```
bool flag;  char cval;
short sval; unsigned short usval;
int ival;   unsigned int uival;
long lval;  unsigned long ulval;
float fval; double dval;


// 基本遵循规则，小转大后面补0，大转小损失n位后的精度
3.14159L + 'a'; a提升为int,int转换成long double
dval + ival; ival 提升为 double
dval + fval; fval 提升为 dval
ival = dval; double 去精度隐式转换为int
flag = dval; 0 false,1 true
cval + fval; cval提升为int,最后转换为 float
sval + cval; 都提升为int
cval + lval; cval 转换成 long
ival + ulval; ival 转换成unsigned long
usval + ival; 根据unsigned short 和 int 所占空间大小进行转换
uival + lval; 根据unsigned int 和 long 所占空间大小进行转换
```

3.15其他隐式类型转换

数组转换成指针

```
int ia[10];
// ia 转换成 指向数组首元素的指针
int *ip = ia;
```

- 指针的转换

```
char *cp = get_string();
// 判0
if(cp){}
// 判空
while(*cp){}
```

- 类类型定义的转换

```
string s = "";
while(cin>>s){
...
}
```

- 转换成常量

```
int i;
// *int -> const *int
const int &j = i;
const int *p = &i;
```

3.16显示转换

```
int i,j;
double slope = i/j;
```

3.17强转

- cast-name <type> expression;

cast-name为

- static_cast; 任何具有明确定义的类型转换; double slope = static_cast <double>(j)/i

```
// 使用static_cast 找回存于 void*指针
void *p = &d;
double *dp = static_cast<double*>(p);
```

- dynamic_cast
- const_cast; 只能改变运算对象的底层const

```
const char *pc;
// 去const
char *p = const_cast<char*>(pc)
```

- reinterpret_cast; 运算对象的位模式提供较低层次上的重新解释

```
int *ip;
// int* -> char*
char *pc = reinterpret_cast<char*>(ip);
```

3.18老旧的强制转换

- type(expr);
- (type) expr;

4.语句

直接写demo了

```
// 流控
if(expr){}else if(expr2){}else

switch(n){
  case 1:
    break;
  default:
    break;
}

// 迭代
```

```
while(expr){}

for(expr;expr;expr){}

for(declaration: expression){
  statement
}

do{}while()

// 跳转

break;
continue;


// goto
end: return;
goto end;

// 异常

try{}
catch(execption){}
catch(execption){}

// 异常类型
exception 常见异常
runtime_error 运行时错误
range_error 值超了
overflow_error 上溢
underflow_error 下溢
logic_error 逻辑错误
domain_error 参数对应的结果不在
invalid_argument 参数无效
length_error 长度超了
out_of_range 超出有效值范围
```

5. 函数

- 可变参
- 重载
- 内联函数
- 断言
- NDEBUG变量
- 函数指针

```
int fact(int val){
  return val;
}
```

```
void nothing(){
}

// 重载

int a (){}
int a(int b){return b;}

// 可变参
initializer_list<T> lst;

void error_msg(initializer_list<string> args){
  for(auto beg=args.begin();beg != args.end();++beg){
      cout << *beg << "\n" << endl;
  }
}


// 内联函数
constexpr int asdf (){return 1}
constexpr int foo = asdf();


// 断言
// 假就退出，真就继续运行
assert(expr);

// NDEBUG类似py
// 输出函数名称
cout << __func__ << endl;
cout << __FILE__ << endl;
cout << __LINE__ << endl;
cout << __TIME__ << endl;
cout << __DATE__ << endl;

// 函数指针
// 首先是函数
bool lengthCompare(const string &,const string &);
// 然后声明一个存储函数的指针
bool (*pf)(const string &,const string &);
// 使用指针存储/指向函数
pf = lengthCompare;
pf = &lengthCompare;
// 调用
pf("asd","asdf");
(*pf)("asd","asdf");
// 重载函数指针
void ff(int*);
void ff(unsigned int);
// 自动指向第二个函数
void(*pf1)(unsigned int) = ff;
// 指向第一个函数
void(*pf2)(int*) = ff;
```

```
// 函数参数
void useBigger(const string &s1,const string &s2,bool pf(const string &,const string &));
void useBigger2(const string &s1,const string &s2, bool (*pf)(const string &,const string &));
// 传一个函数进去
useBigger2("123","1234",lengthCompare);
// 返回函数指针

using F = int(int*,int)
using PF = (int*)(int*,int);
// 声明函数返回函数
PF f1(int);
F *f1(int);
// 直接调用
f1(1)(1,2);
//
string::size_type sumLength(const string&,const string&);
string::size_type largerLength(const string&,const string&);
decltype(sumLength) *getFcn(const string &);
```

6. 类

第一类类

```
#include <iostream>

using namespace std;
// Shape 就是一个抽象类
class Shape
{
// 外部可访问内容
public:
Shape(){};
// 构造函数
Shape(double x, double y)
{
width = x;
height = y;
};
// 虚函数,需要子类实现,必须给默认值
virtual double getArea() = 0;
// 通用函数，可重写
void setWidth(double w)
{
width = w;
};

void setHeight(double h)
{
height = h;
};

// 被保护的部分
protected:
double width;
```

```
double height;

// 私有部分
private:
double total;
};

class ChangFangXing : public Shape
{
public:
double getArea()
{
return width * height;
}
};

class Yuan : public Shape
{
public:
double getArea()
{
return 3.14 * (width * width);
};
};

int main(void)
{
ChangFangXing c;
c.setWidth(10.00);
c.setHeight(10.00);
cout << c.getArea() << endl;

Yuan a;
a.setWidth(10.00);
a.setHeight(10.00);
cout << a.getArea() << endl;
return 0;
}
```

第二类类

```
#ifndef __SALE_HEAD__
#define __SALE_HEAD__
#include <iostream>
using namespace std;
struct Sales_data{
  // 编号
  string bookNo;
  // 总销售收入
  double revenue = 0.0;
  // 某本书的销量
  unsigned units_sold =0;
  string isbn() const{return bookNo;}
  Sales_data& combine(const Sales_data&);
```

```
  // 平均价
  double avg_price() const;
};
Sales_data add(const Sales_data& , const Sales_data& );
ostream& print(const ostream& ,const Sales_data& );
istream& read(istream& ,Sales_data&);
#endif
```

- struct 结构体，也可以看作类
- struct 与c不同允许有内部函数
- 访问内部变量实际上通过this

```
xxx.isbin() 实际上 Sales_data::isbn(&xxx)将调用者指针传入
然后用指针访问调用者的内部值
std::string isbn() const { return this->bookNo; }
```

- 默认情况下 this 为非常量指针,比如在 Sales_data中指针为Sales_data * const,为了防止this转变向对象，所以在形参列表后加入了const 如此指针类型为const Sales_data * const 改变指针指向将报错
- std::string Sales_data::isbn(const Sales_data *const this)将会报错，因为试图改变函数内this的向

6.1 抽象

- 虚基类 开头演示过例子了
- 虚继承

```
class A
{
public:
void setx(int a){x = a;}
int getx(){return x;}
private:
int x;
};

class B: public A
{
public:
void sety(int a){y = a;}
int gety(){return y;}
private:
int y;
};

class C: public A
{
public:
void setz(int a){z = a;}
int getz(){return z;}
private:
int z;
```

```
};

class D: public B, public C
{
//......
};
```

B继承了A,C继承了A,B和C都有A的X,那么D继承了B,C所有A中的x变量在D中有两份

```
#include <iostream>
using namespace std;

class A
{
public:
    void setx(int a){x = a;}
    int getx(){return x;}
private:
    int x;
};

class B: virtual public A
{
public:
    void sety(int a){y = a;}
    int gety(){return y;}
private:
    int y;
};

class C: virtual public A
{
public:
    void setz(int a){z = a;}
    int getz(){return z;}
private:
    int z;
};

class D: public B, public C
{
    //......
};

int main()
{
    D test;
    test.setx(10);
    cout<<test.getx()<<endl;
    return 0;
}
```

如上为虚继承，最后D只有一份A中的函数和变量

6.2 封装

- public 公开
- private 私有
- protected 受保护，派生类中可使用

了解各自的作用域，private 提供getter setter就是封装了。

6.3 继承/派生

```
#include<iostream>
using namespace std;
// 基类
class base
{
public:
    base(){x = 0; y = 0;}
    base(int a){x = a; y = 0;}
    base(int a, int b){x = a; y = b;}
private:
    int x;
    int y;
};

class derived: public base
{
public:
    // 派生类初始化时需要调用基类的构造函数来初始化xy
    derived():base(){z = 0;}
   // 派生类用三个参数，传入了基类中ab来初始化xy,剩下的初始化派生类中的z
    derived(int a, int b, int c):base(a,b){z = c;}
private:
    int z;
};

int main()
{
    derived A;
    derived B(1,2,3);
    return 0;
}
```

6.4 构造函数/析构函数

```
#include <iostream>
using namespace std;

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

```
    ~A(){cout<<"A destructor"<<endl;}
};

class B: public A
{
public:
    B(){cout<<"B constructor"<<endl;}
    ~B(){cout<<"B destructor"<<endl;}
};

class C: public B
{
public:
    C(){cout<<"C constructor"<<endl;}
    ~C(){cout<<"C destructor"<<endl;}
};

int main()
{
    C test;
    return 0;
}
```

运行结果如下：

A constructor

B constructor

C constructor

C destructor

B destructor

A destructor

程序运行结果很好地说明了构造函数和析构函数的执行顺序。构造函数的执行顺序是按照继承顺序自向下的，从基类到派生类，而析构函数的执行顺序是按照继承顺序自下向上，从派生类到基类。

因为每一个类中最多只能有一个析构函数，因此调用的时候并不会出现二义性，因此析构函数不需要式的调用。

转型构造函数

```
class Age
{
public:
    Age(int a){age = a;}
private :
    int age;
}
```

传入age,进行初始化后返回的为对象类型

6.5 多态

```
/*
 * @Author: cuihaonan
 * @Email: devcui@outlook.com
 * @Date: 2021-04-20 17:16:22
 * @LastEditTime: 2021-04-20 17:16:23
 * @LastEditors: cuihaonan
 * @Description: Basic description
 * @FilePath: /kube-console/demo.cpp
 * @LICENSE: NONE
 */


#include <iostream>
using namespace std;

class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape{
   public:
      Rectangle( int a=0, int b=0):Shape(a, b) { }
      int area ()
      {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};
class Triangle: public Shape{
   public:
      Triangle( int a=0, int b=0):Shape(a, b) { }
      int area ()
      {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};
// 程序的主函数
int main( )
{
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);
```

```
    // 存储矩形的地址
    shape = &rec;
    // 调用矩形的求面积函数 area
    shape->area();

    // 存储三角形的地址
    shape = &tri;
    // 调用三角形的求面积函数 area
    shape->area();

    return 0;
}
```

6.6 命名空间

```
namespace Li{   //小李的变量声明
    int flag = 1;
}
namespace Han{   //小韩的变量声明
    bool flag = true;
}
```

6.7 友元函数

```
#include<iostream>
using namespace std;

class book
{
public:
    book(){}
    book(char* a, double p);
   // friend 将下面的函数实现变为友元函数
// 所以display可以访问 private
    friend void display(book &b);
private:
    double price;
    char * title;
};

book::book(char* a, double p)
{
    title = a;
    price = p;
}

void display(book &b)
{
    cout<<"The price of "<<b.title<<" is $"<<b.price<<endl;
}

int main()
```

```
{
    book Alice("Alice in Wonderland",29.9);
    display(Alice);
    book Harry("Harry potter", 49.9);
    display(Harry);
    return 0;
}
```

6.8静态成员静态函数

静态成员使用需要初始化

```
class student
{
public:
    student(){count ++;}
    ~student(){count --;}
private:
    static int count;
    //其它成员变量
};
int student::count = 0;
```

静态函数使用需要初始化

```
#include<iostream>
using namespace std;


class test
{
public:
    static void add(int a);
};

void test::add(int a)
{
    static int num = 0;
    int count = 0;
    num += a;
    count += a;
    cout<<num<<" "<<count<<endl;
}


int main()
{
    test one,two,three;
    one.add(5);
    two.add(4);
    three.add(11);
    return 0;
}
```

static为所有类/派生类共享变量/函数，累加

6.9 内存分配

```
#include<iostream>
using namespace std;

class test
{
public:
    test(int i = 1){num = i;cout<<num<<" Constructor"<<endl;}
    ~test(){cout<<num<<" Destructor"<<endl;}
private:
    int num;
};

int main()
{
// new 相当于malloc
    test * t0 = new test(0);
    test * t1 = new test[5];
    test * t2 = (test *)malloc(sizeof(test));
// delete相当于free
    delete t0;
    delete[] t1;
    free(t2);
    return 0;
}
```

6.10 重载，覆盖，遮蔽

函数重载是指两个函数具有相同的函数名，但是函数参数个数或参数类型不同。函数重载多发生在顶函数之间或者同一个类中，函数重载不需要构成继承关系。

覆盖构成条件和多态构成条件是相同的，覆盖是一种函数间的表现关系，而多态描述的是函数的一种质，二者所描述的其实是同一种语法现象。

函数遮蔽同样要求构成继承关系，构成继承关系的两个类中具有相同函数名的函数，如果这两个函数够成覆盖关系，则就构成了遮蔽关系。遮蔽理解起来很简单，只要派生类与基类中具有相同函数名（意不是相同函数签名，只需要相同函数名就可以了）并且不构成覆盖关系即为遮蔽。