## 数组

1. 数组维度必须是常量表达式。

unsigned int N **=** 42**;**

int a**[**N**];**//错误，N不是常量表达式。常量表达式的意思是：数值不变，且在编译期值可知。在g++中，上述语句正常工作，可能原因是编译器扩展，也有说c++标准c99支持VLA（variable length array）

1. 四不能：不能使用auto自动推断数组元素类型，数组元素不能是引用类型，数组不能拷贝，数组不能赋值。

int array**[**2**]** **=** **{**1**,**2**};** //正确

auto a**[**2**]** **=** **{**1**,**2**};** //错误，数组元素类型不能使用auto自动推断

int **&** b**[**2**]** **=** /\*\*/ //错误，数组元素类型不能是引用

int c**[]** **=** array**;** //错误，数组不能拷贝

c **=** array**;** //错误，数组不能直接赋值

1. 使用auto获得的数组类型是指针类型；使用decltype获得数组类型。

auto a **=** array；//a的类型是指针类型

decltype（array）a **=** **{**1，2**}**；//decltype（array）是数组类型

1. begin（），end（）函数，头文件iterator中。
2. 数组索引类型是int 而不是unsigned int，索引可以为负值。

int array**[]** **=** **{**1**,**2**,**3**,**4**,**5**};**

int **\***p **=** **&**array**[**2**];**

int k **=** p**[-**2**]**//正确，表示array[0]

1. c风格字符串

const char **\*** a**[]** **=** “aaa”**;**

const char**\*** b**[]=**”bbb”**;**

比较：strcmp，若使用==比较的是指针值

连接：strcat，若使用+，表示两个指针相加，错误

1. 旧接口

string s **=** “sss”**;**

const char**\*** a**[]** **=** s**.**c\_str**();**//a[4] = “sss\0”;

1. 多维数组

除了最后一层范围for循环，其他for循环必须使用引用。

int a**[**2**][**2**]** **=** **{{**1**,**1**,},{**2**,**2**}};**

**for(**auto a1**:** a**){**

**for(**auto a2 **:** a1**){**//错误，因为a1退化为指针，类型为 int\* ，对int\*遍历是错误的

cout**<<**a2**<<**endl**;**

**}**

**}**

int a**[**2**][**2**]** **=** **{{**1**,**1**,},{**2**,**2**}};**

**for(**auto**&** a1**:** a**){**

**for(**auto a2 **:** a1**){**//正确，a1类型为int [2]

cout**<<**a2**<<**endl**;**

**}**

**}**

## 表达式

1. 优先级、结合性、求值顺序

求值顺序确定的只有逻辑与（&&），逻辑或（||），逗号运算符（，），双目运算符（？：），其他运算符不保证运算顺序：

a**+**b**\***c**+**d**;**//优先级保证先算b\*c，结合性保证接下来计算a+然后+d

cout**<<**i**<<++**i**<<**endl**;**//不保证先做++i还是先输出i，结果未定义

f**()+**g**();** //不保证先计算f()；如果f（）和g（）由共用变量，结构未定义

1. typedef

**typedef** int int\_array**[**4**];**//表示int\_array是数组元素类型为int，维度为4

int\_array a**;**//等价与int a[4];

**typedef** char**\*** P**;**

const char**\*** p1**;**//等价与char const \*p1;底层const

const P p2**;**//等价与 P const p2; 即就是 char\* const p2;顶层const

1. decltype（X）当X为左值时，得到类型为引用类型

int **\*** p **;**

decltype**(\***p**)** a**;**//因为解引用得到的是左值，所以 a 的类型是 int&

decltype**(&**p**)** b**;**//取值符 & 获得右值，所以得到 b 的类型是 int\*\*

1. 算术运算符

|  |
| --- |
| 一元正号 +  一元负号 - |
| 乘法运算符 \*  除法运算符 /  取模运算符 % |
| 二元加法 +  二元减法 - |

* 1. 运算结果都为右值
  2. 一元正号、加法、减法可用于指针
  3. 取模运算符 % 操作数都为整数，且符号和操作符左边的数相同：

21 **%** **-**5 **=** 1； 21**/-**5**=-**4；//必须保证 m = （m/n）\*n + m%n = m ，xain't所以有余数符号和操作符左边数相同

**-**21 **%** 5 **=** **-**1； **-**21**/**5**=-**4；

* 1. 整数除法向0取整：

**-**1**/**3**=**0；

**-**4.3**=-**1；

1**/**3**=**0；

1. 关系运算符

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

只有？：、&&、||、，这四种运算符保证求值顺序

1. 赋值运算符

赋值运算符优先级较低，复合赋值运算符和赋值运算符之间有差异：

a **+=** b**;**//a取值一次

a **=** a **+** b**;**//a取值两次

1. 位运算符

|  |
| --- |
| 求反 ~ |
| <<  >> |
| & |
| 异或^ |
| | |

1. 整型提升
2. 有符号右移未定义，一般来说用于无符号
3. int只保证16位，long保证32位
4. 优先级小于算术运算符，大于关系运算符、赋值运算符、条件运算符

左结合性

1. 条件运算符

只有当两个表达式都为左值时，结果为左值。

1. sizeof
   1. 优先级和解引用\*等同

右结合性

* 1. 优先级等同与解引用\*，右结合律

int **\***p **=** **nullptr;**

**sizeof** **\***p**;**//右结合性兵器优先级等同于 \* ，不对表达式求值所以正确

* 1. c++11新特性允许使用域运算符获取成员大小

class ClassA**{**

public**:**

int a**;**

**}**

**sizeof(**ClassA**::**a**);**//正确，一般来说访问类成员需要类对象，sizeof无须我们提供一个对象

1. 类型转换
   1. 隐式转换：
      1. 算术提升

char a**;**

short b**;**

a**+**b**;**//a 和 b 都转换为 int

* + 1. 指针和数组

int a**[**2**]** **=** **{**2**,**1**};**

int f**(**int**\*** p**){...};**

f**(**a**);**//a 转换为指针

* + 1. 指针

int**\*** pi**;**

void**\*** pv **=** pi**;**//int\* pi转换为 void\* pv

* + 1. bool

**while(**cin**>>**a**){**//istream对象转换为bool类型

/\*...\*/

**}**

* + 1. const

int a **=** 8**;**

const int b **=** a**;**//int 类型 隐式转换为 const int

* + 1. 类类型定义

string s **=** "hello world"**;**//const char\* 隐式转换为 string

派生类转换基类

* 1. 显式转换
     1. static\_cast<type>()：任何具有明确定义的类型转换，并且没有底层const
     2. const\_cast<type>()：去掉const属性
     3. reinterept\_cast<type>()：位模式下对对象重新解释

int a **=** 8**;**

char**\*** p **=** **reinterpret\_cast<**char**\*>(&**a**);**

* + 1. dynamic\_cast<type>()：派生类转换为基类类型

## 语句控制

1. switch
   1. default不影响先匹配case

int a **=** 1**;**

**switch(**a**){**

**default:**cout**<<**"default"**<<**endl**;break;**

**case** 1**:**cout**<<**"1"**<<**endl**;break;**

**}**

//结果输出 1

* 1. 不能跳过变量的初始化语句

int a **=** 1**;**

**switch(**a**){**

**case** 0**:**

int b **=** 1**;** //#1

**break;**

**case** 1**:**

cout**<<**b**<<**endl**;**//错误，不允许跳过 #1

**break;**

**default:**

cout**<<**"default"**<<**endl**;**

**break;**

**}**

1. for

**for(**int i **=** 0 **;** **;** **++**i**)**//省略判断表示始终为true

**{**/\*...\*/**}**

1. while do…while
   1. do..while至少执行一次
   2. do…while()中不允许使用while中的变量。

**do{**

cout**<<**i**<<**endl**;**//错误，使用了while（）括号中的i

**}while(**int i **<** 0**);//注意有分号**

1. 4种跳转
   1. break；可用于switch；for；while（包括do…while）
   2. continue；不能用于switch
   3. goto；
   4. return；
2. try
   1. 没有finally关键字
   2. 没有catch则最终会调用系统的terminate
   3. 只有exception、bad\_cast、bad\_alloc默认初始化，不需要初始值

## 函数

1. 函数参数
   1. 函数实参不保证求值顺序
   2. 作用域和生命周期区别。作用域表示变量可见不可见，生命周期指该对象在内存中存在的一段时间
   3. 局部自动变量执行默认初始化（对于内置类型其值未定义），局部静态变量执行值初始化
   4. 编译器只能执行一次隐式类类型转换

class A**{**

A**(**string s**);**

void f**(**A a**);**

**}**

int main**(){**

f**(**"123"**);**//错误，“123” 类型为 const char\* ，编译器可以将其隐式转换为 string ，但是不能在使用构造函数将其转换为A

A a**(**"123"**);**//正确，“123” 隐式转换为 string ，然后使用构造函数

**}**

1. 函数返回值
   1. 优先级：等同于成员选择符 . 和 –>；结合性：左结合
   2. 左值右值：返回引用的时左值
   3. 返回数组指针：

int **(\***fun**(**int i**))[**10**];**//可以对fun的返回结果解引用，解引用的结果得到大小是10的数组；数组元素是int

auto fun**(**int a**)** **->**int**(\*)[**10**];**//尾置返回类型

int a**[**4**]={**1**,**2**,**3**,**4**};**

decltype**(**a**)** **\*** fun**(**int i**);**//decltype得到维数是4的数组类型，fun返回指向维数是4的数组的指针

1. 函数参数
   1. const形参忽略顶层const

int f**(**const int a**);**

int f**(**int a**);**//错误，等价于上一个函ge数，不能构成重载；赋值时顶层const被忽略

const int aa **=** 1**;**

f**(**a**);**//aa赋值给形参时，aa的const属性被忽略，不会匹配到f(const int a);

* 1. 数组形参
     1. 结束标志（c字符串的结束符’\0’）
     2. 传递数组长度
     3. 传递起始结束位置
  2. main函数参数,自定义参数从argv[1]开始

void main**(**int argc**,**char**\*** argv**[]){**

**for(**int i **=** 0 **;** i **<** argc**;++**i**){**

cout**<<**"argv["**<<**i**<<**"]:"**<<**argv**[**i**]<<**endl**;**

**}**

**}**

**./**main 'aa' 'bb'

输出：

argv**[**0**]:**main

argv**[**1**]:**aa

argv**[**2**]:**bb

* 1. 可变参数列表
     1. 参数列表：initializetor\_list

void f**(**initializer\_list**<**int**>** a**){**

**for(**auto p **:**a**){**

cout**<<**p**<<**" "**;**

**}**

cout**<<**endl**;**

**}**

int main**()**

**{**

f**({**1**,**2**,**3**,**4**});**

initializer\_list**<**int**>** a **=** **{**1**,**2**,**3**,**4**};**

initializer\_list**<**int**>** b **=** a**;**

initializer\_list**<**int**>** c**;**

c **=** a**;**

cout**<<**begin**(**a**)<<**endl**;** //输出0x46e038

cout**<<**begin**(**b**)<<**endl**;** //输出0x46e038；表名拷贝共享列表元素

cout**<<**begin**(**c**)<<**endl**;** //输出0x46e038；表名赋值共享列表元素

**return** 0**;**

**}**

* + 1. 可变参数模板
    2. c省略符号

1. 函数重载
   1. 顶层const不能作为重载条件

引用都是底层const，可以作为重载条件

* 1. cosnt\_cast 与重载

const string**&** f**(**const string**&** s**){}**//#1

string**&** f**(**string**&** s**){**//#2和#1构成重载

**return** **const\_cast<**string**&>(**f**(const\_cast<**const string**&>(**s**)));**

**}**

* 1. 重载作用域

int f**(**double**);**//#1

int f**(**string**);**//#2

int main**(){**

int f**(**int**);**//使得#1，#2不可见

f**(**"111"**);**//错误，f(string)不可见

f**(**2.3**);**//正确，调用 f(int),等价于 f(2);

**}**

名字查找发生于类型检查之前

1. 特殊用途语言特性
   1. 函数默认参数
      1. 使用：具有默认值的参数必须全部放在参数列表右边

void f**(**int j**,**int i **=** 1**);**//正确

void f**(**int i **=** 1**,** int j **);**//错误

* + 1. 声明：函数可以重复声明，但是同一作用域下，默认参数不能被重复赋值

void f**(**int i**,** int j**);**

void f**(**int i **=** 1**,**int j**=**2**);**//正确

void f**(**int i**,** int j**=**2**);**

void f**(**int i **=** 1**,**int j**=**2**);**//错误，j被重复赋值

* + 1. 作用域

int di **=** 1**;**

int dj **=** 2**;**

void f**(**int i **=** di**,**int j**=**dj**);**

int main**(){**

f**();** //输出3

di **=** 2**;**

f**();** //输出4，参数赋值发生在调用阶段，所以 i = di = 2;

int dj **=** 1**;**

f**();** //输出4，dj 隐藏 外层dj，但是没有改变参数中的dj，名字解析发生于赋值之前，此时，i = 2 ,j = 2

**return** 0**;**

**}**

void f**(**int i**,**int j**){**

cout**<<**i**+**j**<<**endl**;**

**}**

* 1. inline和constexpr
     1. inline只是请求，并非要求
     2. constexpr能用于常量表达式的函数
     3. constexpr参数、返回值是字面值类型，只能有一个return

constexpr int g**(**int a**){**

**if(**a **<** 0**){**

**return** **-**1**;**

**}**

**else{**

**return** 1**;**

**}**

**}**

int main**(){**

g**(**1**);**

**return** 0**;**

**}**

error**:** body of constexpr function ‘constexpr int g**(**int**)**’ **not** a **return-**statement

只能出现一个 return 语句

* + 1. constexpr函数隐式内联
    2. constexpr不一定返回常量表达式

constexpr int g**(**int a**){** //若 a 是常量表达式，则函数返回常量表达式；若 a 不是常量表达式，函数返回不是常量表达式

**return** a**\***2**;**

**}**

int main**(){**

int i **=** 1**;**

cout**<<**g**(**i**)<<**endl**;** //正确，说明函数不一定返回常量表达式

int sa**[**g**(**1**)];** //正确，g(1) 是常量表达式

int sa**[**g**(**i**)];** //c++ primer说是错误，g(i)不是常量表达式，g++测试正确，应该是编译器扩展

**return** 0**;**

**}**

* 1. assert和NDEBUG
     1. assert依赖于NDEBUG，如果定义了NDEBUG，则assert什么也不做

#include<iostream>

#include<cassert> //包含头文件

**using** **namespace** std**;**

int main**(){**

int i **=** 1**;**

assert**(**i**>**1**);**

**return** 0**;**

**}**

输出：int main**():** Assertion `i**>**1' failed**.**

g**++** **-**D NDEBUG test**.**cpp **-**o main //作用是在程序开始位置添加 #define NDEBUG

**./**main

2

* + 1. NDEBUG

\_\_func\_\_:函数名

\_\_TIME\_\_:编译时间

\_\_DATE\_\_:编译日期

\_\_LINE\_\_:当前行号

cout**<<**\_\_func\_\_**<<**endl**;**

cout**<<**\_\_DATE\_\_**<<**":"**<<**\_\_TIME\_\_**<<**endl**;**

cout**<<**\_\_LINE\_\_**<<**endl**;**

1. 函数确认
   1. 函数范围
      1. 函数名
      2. 作用域
   2. 可行函数
      1. 函数参数个数
      2. 参数类型
   3. 最佳匹配
      1. 所有参数都不劣于其他
      2. 至少有一个参数优于其他

int f**(**int**);**

int f**(**float**);**

f**(**3.4**)**//产生二义性，3.4是double，其到转换到float和转换到int优先级相同，所以错误

int f**(**int**,**double**);**

int f**(**double**,**int**);**

f**(**1**,**2.4**);**//产生二义性，不满足最佳匹配的两个条件

* 1. 类型转换优先及顺序
     1. 精确匹配：完全匹配，数组和函数与指针的转化，顶层const忽略
     2. const转换
     3. 类型提升
     4. 算术转换
     5. 类类型转换

1. 函数指针
   1. 函数指针的类型包括返回值和参数类型、函数指针不存在类型转换
   2. 重载函数

int f**(**int**);**

int f**(**float**);**

int **(\***p**)(**int**)** **=** f**;**

* 1. 函数指针作为参数

int f**(**int**);**

**typedef** int **(\***FE**)(**int**);** //#1 PE类型是 int (\*)(int)

**typedef** decltype**(**f**)** FE**;** //#2 等价于 #1

**typedef** decltype**(**f**)** **\***PE**;** //#3 对于函数参数来说，其等价于#2 ；对于返回值来说不等价，编译器不能自动将函数类型转换为指针

**using** PE **=** int **(\*)(**int**);**//#4 等价于#3

**using P = int(int);// #5 等价于#2**

* 1. 函数指针作为返回值

总结：函数作为参数时：函数类型和函数指针等价

函数作为返回值：返回类型不能时函数类型，函数类型不能自动转换为函数指针

**using** PE **=** int **(\*)(**int**);**//#4 等价于#3

**using** P **=** int**(**int**);**// #5 等价于#2

F g**(**int**);**//错误，返回类型是函数类型

PE g**(**int**);**//正确，返回类型时函数指针

## static

修饰局部变量时，修改其生存期，不改变作用域。

修改全局变量时，修改其作用域，不改变生存周期。

解释：

链接器将目标文件链接为可执行文件，此时会检查先前标记为外部连接的变量或函数是否可在链接文件中找到定义。

连接分为外部连接和内部连接，外部连接表示该变量或函数可被其他文件使用，内部连接表示该变量或函数只能用于本文件，文件外部不可访问。

举例：B.cpp使用了变量a\_var，并声明extern，编译时编译器将a\_var标记为外部链接的变量，编译通过；

* 若A.cpp定义了变量a\_var并标记该变量为外部链接，那么链接器将会允许B.cpp使用a\_avr变量；
* 若A.cpp定义a\_var时将其标记为内部链接，那么B.cpp将不被允许使用A.cpp的a\_var，链接报错。

问：为什么全局变量声明为static后，外部文件就不能访问该变量了？

答：static强制全局变量或者函数变成内部链接。

问：变量何时是外部链接的？

答：函数体外定义的所有变量（c++中除了const）默认都是外部链接的。

## const

### 作用域

c++中，const声明的变量默认内部链接（文件外部不可见），定义时使用extern将其变为外部链接。

c中，const声明的变量默认外部链接。

解释：

首先const是c++语法提出的用于代替c中#define的（为什么使用const代替#define：因为#define只是单纯的文本替换，编译期进行，并没有类型检查功能，容易出错；const会有类型检查，可能在编译期或者运行期进行），c中的const和c++中const的行为有所不同：c中const一定会给变量分配内存，c++的const不一定：

* 当程序存在给const变量取址或者定义使用extern时，编译器被告知需要给const变量分配内存（之后就外部链接了），运行时会用到，编译期不可以使用该变量了。
* 其他情况，发生变量折叠，即在所有出现该const变量名的地方替换（带类型检查）为变量值，运行时不会给变量分配内存。

所有分配了内存的变量在编译期不能确定其值。

由于声明数组时，会指定其大小，给数组分配内存时，大小应该是编译期可知的。所以有结论：

1. c中const变量不能作为数组大小：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main**()**  **{**  const int sz **=** 100**;**  int a**[**sz**];//讲道理这行会报错，但实际上运行正常，原因是现代编译器对这个做了优化。**  printf**(**"%d\n"**,sizeof(**a**));**  **return** 0**;**  **}** |

1. c++中分配了内存（& 和 extern定义）时，不能作为数组大小

|  |
| --- |
| #include <iostream>  using namespace std**;**  int main**()**  **{**  int ssz**;**  cin**>>**ssz**;**  int aa**[**ssz**];//讲道理会报错**  const int sz**[]** **=** **{**100**,**200**};**  int a**[**sz**[**0**]];//讲道理会报错**  cout **<<** **sizeof(**a**)** **<<** endl**;**  **return** 0**;**  **}**  编译器太坏了，书上说的都给优化掉了 |

### const和指针

const int\* （int const\* ） vs int\* const

从右往左看，const左边是谁，谁就是常量。

### const和类

|  |  |
| --- | --- |
| 修饰成员函数 | 形如int f() const，表示该类的const对象可以调用该函数，而没有声明const的函数，const对象不能调用。  解释：编译器将类成员函数int f()扩展为  int f( A\* const this)，表示this指针不能指向其他对象；  int f()const ，编译器将其扩展为  int f(const A\* const this)表示this指向const对象，this指针和对象都不可变。  必须同时在声明和定义中出现const |
| 修饰成员变量 | 1. const变量必须在初始化列表中赋值   #include <iostream>  using namespace std**;**  class A**{**  const int sz **;**  A**(**int i**){**sz **=** i**;}//改为A(int i):sz(i){}就正确**  **};**  int main**()**  **{**  **return** 0**;**  **}**    解释：const变量不能被改变，只能在定义时赋值，初始化列表表示定义时赋值。   1. 类中数组size使用变量时，变量必须是static const或enum   #include <iostream>  using namespace std**;**  class A**{**  const int sz **;**  int a**[**sz**];**  A**(**int i**):**sz**(**i**){}**  **};**  int main**()**  **{**  **return** 0**;**  **}**    #include <iostream>  using namespace std**;**  class A**{**  const int sz **=** 100**;//改为static const sz = 100正确**  int a**[**sz**];**  A**(**int i**){}**  **};**  int main**()**  **{**  **return** 0**;**  **}**    解释：  static const可以看作编译期常量。   1. 一些结论：  * static非常量变量只能在类外定义。 * static常量变量可以在类内或者类外定义，但是用于数组大小时只能在类内定义。 * 使用static const或者enum定义类中数组的大小参数 |
| 可变const | 1. const分为按位const和按逻辑const，按位const严格的不能改变，按逻辑const部分可变。 2. 类变量声明为mutable，则对象a可以在int f()const中改变该变量。 |

## 类

1. 先编译整个类，然后才处理函数，所以处理函数时，成员变量已经可用
2. 非成员函数应该和类定义在同一头文件中，friend声明只是规定访问权限，没有声明的作用，所以还需要在类外声明非成员函数
3. 以下三种情况编译器不合成默认构造函数
   1. 存在自定义的构造函数
   2. 存在成员是内置类型或者复合类型（引用和指针），但是没有类内初始值，其默认初始化是未定义的
   3. 存在成员变量没有默认构造函数
4. struct（默认public）和class（默认private）唯一区别是默认访问权限的不同
5. 类的其他特性
   1. 成员变量
      1. 类型变量必须先声明后使用
      2. 类内初始值
      3. mutable变量，即使是const对象，该成员变量也可以被改变
   2. 成员函数
      1. friend仅仅是作用域声明，还必须有普通的函数声明，或者在类前，或者在类后
      2. inline只需要在定义时说明就行，必须和类定义放在同一头文件
      3. const限定符可以用作重载
   3. 类的不完全类型
      1. 类内可以使用该类指针或者引用作函数参数或者返回值
      2. 不能在类内声明该类类型
6. 构造函数
   1. 初始值列表
      1. 这时是初始化而非赋值，如果没有初始值列表，在构造函数之前都会对成员变量进行默认初始化。
      2. const、引用、没有默认构造函数的类成员都必须使用初始值列表

class A**{**

B**\*** b **=** **new** B**();**

int a **=** 1**;**

int**&** c**=**a**;**

const int d **=** 1**;**//正确。说明必须使用初始值列表的三种情况可以使用类内初始值

**};**

但是类内初始值不能用于非const static

class A**{**

static int a **=** 1**;**//错误，非const static 变量不允许类内初始值，改为 const static int a = 1;则正确

**}**

* + 1. 成员变量初始化顺序是按照定义顺序
  1. 委托构造函数
  2. 默认构造函数
     1. 默认初始化：非静态局部变量或数组、类没有给他的类类型成员提供初始值。
     2. 值初始化：没有初始值的静态局部变量、有部分初始值的数组、显式使用T();

int f**(){**

int i**;**//默认初始化，值未定义

int a**[**2**];**//同上

static int j **;**//使用值初始化，值为0

int b**[**2**]** **=** **{**1**};**//使用值初始化，b[0]=1,b[1]=0;

**}**

* 1. 隐式类型转换
     1. 只能执行一次隐式类型转换
     2. explicit：声明：只能用于含有一个参数的构造函数

定义：只在声明时使用，定义部分不要explict

使用：只能用在直接初始化形式，不能用于=形式

class C**{**

public**:**

explicit C**(**int i**);**

**};**

explicit C**::**C**(**int i**){**//错误，只需要在声明部分说明explicit

cout**<<**"hello"**<<**endl**;**

**}**

C c**=**1**;**//错误，只能使用直接初始化的方式

C c**(**1**);**//正确，或者C c=C(1);

1. 静态成员
   1. 静态成员函数
      1. 不能是const或者使用this指针
      2. 只出现在声明中
   2. 静态成员变量
      1. 非const static不能使用类内初始值const static可以声明类内初始值，但一般还在类外声明
      2. 指针类型、引用类型、静态成员可以是所属类类型，即不完全类型

class A**{**

A a0**;**//错误，不能时不完全类型

A**\*** a**;**//正确

A**&** b**;**//正确

static A c**;**//正确

**};**

* + 1. 静态变量可以作为函数参数默认值

class A**{**

static A c**;**

void f**(**A a**=**A**::**c **){}**//正确

**};**

## 类再探

1. 基类和派生类定义
   1. 基类
      1. 构造函数不能是虚函数、析构函数是虚函数

class A**{**

virtual **~**A**()=default;**

**};**

class AA**:**public A**{**

virtual **~**AA**()=default;**

**};**

出错：

E**:**\codeblocksProj\test\main**.**cpp**|**11**|**error**:** looser **throw** specifier **for** 'virtual AA::~AA()'**|**

解决：

class AA**:**public A**{**

virtual **~**AA**()** **throw()=default;**

**};**

另外一个问题：virtual函数应该是public

class A**{**

virtual **~**A**()=default;**//#1

**};**

class AA**:**public A**{**

virtual **~**AA**()throw(){};**//报错：#1是private

**};**

* + 1. 显式指明overide

class A**{**

virtual void f**();**//#1

void f1**();**

**};**

class AA**:**public A**{**

virtual void f**(**int**)**override**;**//出错，显式指明覆盖#1，但是参数不匹配，所以编译报错

virtual void f**()** override**;**//正确

void f1**()** override**;**//出错，f1()并不是虚函数

**};**

* 1. 派生类
     1. 访问说明符指定了基类成员对于派生类的可见性
     2. 基类不能是不完全类型，即使用基类时，基类必须已经定义

class A**;**

class AA**:**public A**{**//错误，class A未定义，是不完全类型

virtual void f**()**override**;**

void f1**();**

**};**

class A**{**

virtual void f**();**

void f1**();**

**};**

* + 1. final指定该类不能被继承，指定函数不能被覆盖

示例1：

class A final**{**

**};**

class AA**:**public A**{**//错误，不能从final类派生

**};**

**示例2：**

class A **{**

virtual void f**()**final**;**

**};**

class AA**:**public A**{**

virtual void f**();**//错误，不能覆盖final函数

**};**

* + 1. 指针、引用可用于动态绑定
    2. 派生类到基类隐式转换，基类到派生类使用dynamaic\_cast

1. 虚函数
   1. 虚函数返回值（指针和引用例外）和参数类型严格匹配
   2. 基类虚函数，直接或者间接继承他的派生类中，该函数都是虚函数
   3. 每一个虚函数都应该有定义
   4. 虚函数默认实参根据基类中的确定

class A **{**

public**:**

virtual void f**(**int i **=** 1**){**//#1

cout**<<**"A: "**<<**i**<<**endl**;**

**}**

**};**

class AA**:**public A**{**

public**:**

virtual void f**(**int i **=** 2**){**//#2

cout**<<**"AA: "**<<**i**<<**endl**;**

**}**

**};**

A**\*** a **=** **new** AA**();**

a**->**f**();**

输出：

AA**:** 1

分析：a实际类型是AA，运行时绑定AA的函数 f #2**,**所以调用#2，但是i的值是 1 ，说明默认参数根据基类确定

* 1. 回避虚函数机制：用于在派生类中的虚函数调用基类对应的虚函数

class A **{**

public**:**

virtual void f**(**int i **=** 1**){**

cout**<<**"A: "**<<**i**<<**endl**;**

**}**

**};**

class AA**:**public A**{**

public**:**

virtual void f**(**int i **=** 2**){**

cout**<<**"AA: "**<<**i**<<**endl**;**

**}**

**};**

A**\*** a **=** **new** AA**();**

a**->**A**::**f**();**//回避虚函数机制，输出 A: 1

1. 抽象基类
   1. 类函数声明=0，只出现在声明中
   2. 抽象类不能实例化对象
2. 访问说明符
   1. 派生类访问的是派生类中基类的那一部分，而不是基类本身
   2. 继承访问说明符不影响类内访问权限，只影响派生类对象的访问权限

示例1：

class A**{**

private**:**

int pri\_m **=** 0**;**

protected**:**

int pro\_m **=** 1**;**

public**:**

int pub\_m **=** 2**;**

**};**

class B**:**private A**{**

public**:**

void f**(){**

cout**<<**pri\_m**<<**endl**;**//错误，pri\_m 是private，不能访问

cout**<<**pro\_m**<<**endl**;**//正确,pro\_m 对于B来说属于private

cout**<<**pub\_m**<<**endl**;**//正确，pub\_m 对于B来说属于private

**}**

**};**

**示例2：**

class A**{**

private**:**

int pri\_m **=** 0**;**

protected**:**

int pro\_m **=** 1**;**

public**:**

int pub\_m **=** 2**;**

**};**

class B**:**protected A**{**

**};**

cout**<<**b**->**pub\_m**<<**endl**;**//错误，pub\_m 对于属于B类型的protected，不可类外访问

* 1. 派生类向基类转换的代码点，基类的public成员必须可访问，也就是说要想使用多态性，必须公有继承

class A**{**

**};**

class D**:**protected A**{**

**};**

A**\*** d **=** **new** D**();**//错误，等价于A\* a = dynamic\_cast<A\*>(new D());但是该代码点A的public成员不可访问，所以不能转换

D**\*** d **=** **new** D**();**//正确

* 1. 派生类使用using 变量 改变派生访问说明符的效果，使用using 函数 使得派生类不必重写所有的基类同名函数

示例1：

class A**{**

public**:**

void f**(){**cout**<<**"A"**<<**endl**;}**

**};**

class D**:**public A**{**

public**:**

void f**(**int a**){**cout**<<**"D"**<<**endl**;}**

**};**

D d**;**

d**.**f**();**//错误，D中的f(int)隐藏了A中的f()。

示例2：

class D**:**public A**{**

public**:**

**using** A**::**f**;**

void f**(**int a**){**cout**<<**"D"**<<**endl**;}**

**};**

D d**;**

d**.**f**();**//正确，using相当于重载了所有的f

* 1. struct 和 class 区别只有：成员默认访问说明符和派生默认访问说明符

1. 作用域
   1. 静态类型决定函数是否可用；

class A**{**

**};**

class D**:**public A**{**

public**:**

void f**(){}**

**};**

A**\*** d **=** **new** D**();**

d**->**f**();**//错误，d 的静态类型是 A\* ,而 A 并没有 f 函数，第一步出错

* 1. 从内层作用域到外层作用域依次查找函数；

示例1：

class A**{**

public**:**

void f**(**int**){}**

**};**

class D**:**public A**{**

**};**

D**\*** d **=** **new** D**();**

d**->**f**(**1**);**//正确，首先在 D 作用域找 f 函数，没有找到，继续在外层作用域 A 中找，找到f(int)

* 1. 若函数为虚函数并且调用对象是引用或者指针，进行动态绑定；若函数非虚函数或者调用对象不是引用或者指针，执行静态绑定

1. 继承中的拷贝控制
   1. 基类中析构函数为虚函数
   2. 构造函数和析构函数中调用虚函数，他调用和其构造/析构对应的类类型一致的类成员函数
   3. using设置派生类继承基类中的构造函数：访问控制级别不变、默认参数不继承、代码段重新生成；使用using默认继承所有的基类构造函数，两个例外：1、自定义隐藏基类的2、合成的默认/拷贝/构造不能继承，using不能算作自定议构造函数，派生类还是会合成默认
   4. 移动构造/赋值和拷贝构造/赋值 合成时机：当存在自定义的析构或者拷贝构造/赋值时，编译器不合成移动版本；当自定义移动版本，编译器不合成拷贝版本
   5. 派生类负责基类部分的初始化工作
   6. 基类默认构造/拷贝控制时删除的或者不可访问，派生类中相应版本默认也是删除的；基类析构函数是删除的或者不可访问，派生类中拷贝控制默认是删除的
2. 派生与容器
   1. 容器只能存储同一种类型。当存储类型为基类，存储派生类对象将会导致截断派生类部分，只存储基类部分；当容器存储类型为派生类，基类不能储存

class A**{**

public**:**

int i **=** 1**;**

**};**

class D**:**public A**{**

public**:**

int j **=** 2**;**

**};**

**示例1：**

vector**<**D**>** v**;**

A a**;**

v**.**push\_back**(**a**);**//出错：类型不匹配

**示例2：**

vector**<**A**>** v**;**

D d**;**

v**.**push\_back**(**d**);**//不报错，但是发生了截断

cout**<<**v**.**back**().**j**<<**endl**;**//报错，没有成员 j

* 1. 容器同时存储基类和派生类，只能存储指针。数组和容器不能存储引用。
  2. shared\_ptr可以进行派生类到基类的转换

class A**{**

public**:**

virtual void f**(){**cout**<<**"A"**<<**endl**;}**

**};**

class D**:**public A**{**

public**:**

virtual void f**(){**cout**<<**"B"**<<**endl**;}**

**};**

vector**<**shared\_ptr**<**A**>>** v**;**

v**.**push\_back**(**make\_shared**<**A**>());**//添加A类型对象

v**.**push\_back**(**make\_shared**<**D**>());**//添加B类型对象

v**.**front**()->**f**();**//输出：A ；A中f()

v**.**back**()->**f**();**//输出：B ；B中f().动态绑定

## 智能指针

1. 两类智能指针：shared\_ptr：多个指针共享一个对象；unique\_ptr：指针独占对象；（weak\_ptr：弱指针，指向shared\_ptr指向的对象）

shared\_ptr和unique\_ptr共有的操作：

|  |
| --- |
| shared\_ptr<T> sp;//默认初始化，指针指向空 |
| unique\_ptr<T> up;//同上 |
| p->mem; |
| p.get(); |
| swap(p,q);||p.swap(q); |

shared\_ptr独有的操作：

|  |
| --- |
| make\_shared<T> (args);//类似emplace，用参数构造对象 |
| shared\_ptr<T> p(q); |
| p.unique(); |
| p.use\_count(); |
| shared\_ptr<T> p = q; |

class A**{**

public**:**

A**(**int i**){}**

void f**(){}**

**};**

shared\_ptr**<**A**>** sp**(new** A**(**1**));**//正确

shared\_ptr**<**A**>** sp **=** **new** A**(**1**);**//错误，shared\_ptr构造函数是explicit的，不能执行隐式转换

shared\_ptr**<**A**>** sp **=** make\_shared**<**A**>(**1**);**//正确

sp**->**f**();**//正确

sp**.**get**()->**f**();**//正确，获取p中保存的指针，然后使用 f 函数，需要小心使用

cout**<<**sp**.**unique**()<<**endl**;**//输出：1

cout**<<**sp**.**use\_count**()<<**endl**;**//输出：1

shared\_ptr**<**A**>** sp2**(**sp**);**//正确，递增sp的计数器

cout**<<**sp**.**unique**()<<**endl**;**//输出：0

cout**<<**sp**.**use\_count**()<<**endl**;**//输出：2

1. 需要动态内存的原因
   1. 不知道所需对象的具体类型
   2. 不知道所需对象的个数
   3. 需要共享对象
2. new
   1. 默认初始化和值初始化

int**\*** i **=** **new** int**;**

int**\*** j **=** **new** int**();**

string**\*** s **=** **new** string**;**

string**\*** ss **=** **new** string**();**

cout**<<**"i: "**<<\***i**<<**" j: "**<<\***j**<<**"\ns: "**<<\***s**<<**" ss: "**<<\***ss**<<**endl**;**

输出：

i**:** 7409448 j**:** 0//i执行默认初始化，值为随机值；j值初始化=0

s**:** ss**:**// s执行默认初始化，ss执行值初始化，默认初始化等同于值初始化，为空串

* 1. new const

const int**\*** i **=** **new** const int**;**//错误，const必须初始化

const int**\*** i **=** **new** const int**(**8**);**//底层const不能更改

* 1. 定位new

auto p **=** **new** **(**nothrow**)** int**();**//即使分配内存失败，也不抛出错误

* 1. auto

A a**;**

auto p **=** **new** auto**(**a**);**//根据 a 的类型推断类型

auto p **=** **new** auto**(**a**,**1**);**//只能用于一个参数

1. delete
   1. 不能delete非new 的对象

A a**;**

**delete** a**;**//错误，a 不是new出来的对象

**delete** **&**a**;**//错误，可以正确编译，但是行为是未定义的，&a指向的是局部变量

* 1. 忘记delete
  2. 使用已经delete的对象

class A**{**

public**:**

void f**(){**cout**<<**a**<<**endl**;}**

int a **=** 1**;**

**};**

A**\*** a **=** **new** A**();**

**delete** a**;**

a**->**f**();**//可以编译运行，但是输出的值不确定，行为未定义

* 1. delete多次一个对象

class A**{**

public**:**

void f**(){**cout**<<**a**<<**endl**;}**

int a **=** 1**;**

**};**

A**\*** a **=** **new** A**();**

**delete** a**;**

a**->**f**();**//可以编译运行，但是输出的值不确定，行为未定义

**delete** a**;**//可以编译运行，但是行为未定义

* 1. 野指针

**示例1：**

class A**{**

public**:**

void f**(){**cout**<<**a**<<**endl**;}**

int a **=** 1**;**

**};**

A**\*** a **=** **new** A**();**

**delete** a**;**//a指向的对象已经释放，但是 a 还是指向原内存，a为野指针

a **=** **nullptr;**//保证 a 不指向任何地址

但是并不能完全防止野指针：

**示例2：**

A**\*** b **=** a**;**

**delete** a**;**

a **=** **nullptr;**//b还是野指针

1. shared\_ptr
   1. 接受参数的构造函数是explicit的，不能执行隐式转换

（explicit的构造函数，类型转换只能使用直接初始化，不能使用=操作符）

shared\_ptr**<**string**>** sp **=** **new** string**;**//错误，不能执行隐式转换

shared\_ptr**<**string**>** sp**(new** string**());**//正确，explicit 要求直接初始化

* 1. 能够使用的操作

|  |
| --- |
| shared\_ptr<T> sp();  shared\_ptr<T> sp(sp1);//绑定内置指针，sp1必须是new分配的对象，如果不是，也可以运行，但是行为未定义：  string s**(**"hi"**);**  **{**  shared\_ptr**<**string**>** sp**(&**s**);**  cout **<<** **\***sp**<<** endl**;**//出处hi  **}**  cout**<<**s**<<**endl**;** //输出乱码，因为上一个sp离开作用域后，释放了“hi”所在的空间，s指向的内存被其他程序占用 |
| shared\_ptr<T> sp(up);//使用unique\_ptr初始化shared\_ptr,将up置空，up必须为右值  示例：  unique\_ptr**<**string**>** up**(new** string**(**"hi"**));**  cout**<<\***up**<<**endl**;**  shared\_ptr**<**string**>** sp**(**up**);**//出错：error: cannot bind 'std::unique\_ptr<std::basic\_string<char> >' lvalue to 'std::unique\_ptr<std::basic\_string<char> >  //意思是 up 必须为右值  shared\_ptr**<**string**>** sp**(**std**::**move**(**up**));**//正确  cout **<<** **\***up**<<** endl**;**//错误 |
| shared\_ptr<T> sp(sp2,d);//配置删除器 |
| p.reset();p.reset(q);p.reset(q,d); |

* 1. 注意事项：不要混合使用内置指针和智能指针

1. unique\_ptr
   1. 不能拷贝和赋值

函数返回值可以拷贝

* 1. 支持的操作

|  |
| --- |
| unique\_ptr<T> u;  unique\_ptr<T,D> u;  unique\_ptr<T,D> u(d); |
| u.release();//只是释放指针，并不释放对象空间,返回内置指针  string**\*** s **=** **new** string**(**"hi"**);**  unique\_ptr**<**string**>** up**(**s**);**  cout**<<\***up**<<**endl**;**//输出“hi”  up**.**release**();**  cout**<<\***s**<<**endl**;** //输出“hi”  cout**<<\***up**<<**endl**;**//无输出，说明只是将up置空，不释放 “hi” |
| u.reset();  u.reset(p);  u.reset(p,d); |

* 1. 注意事项：

使用release和reset进行拷贝：

unique\_ptr**<**string**>** up**(new** string**(**"hi"**));**

unique\_ptr**<**string**>** up2**;**

up2**.**reset**(**up**.**release**());**

cout**<<\***up2**<<**endl**;**//输出“hi”

cout**<<\***up**<<**endl**;**//无输出，up将“hi”过继给up2

1. weak\_ptr
   1. 指向shared\_ptr指向的对象，不增加引用计数
   2. 支持的操作：

|  |
| --- |
| weak\_ptr<T> w;  weak\_ptr<T> w(sp); |
| w=sp; |
| w.reset();  w.use\_count();  w.expired();  w.lock() |

shared\_ptr**<**string**>** q**(new** string**(**"hi"**));**

auto p**(**q**);**

cout**<<**p**.**use\_count**()<<**endl**;**//输出：2

weak\_ptr**<**string**>** w**(**q**);**

cout**<<**p**.**use\_count**()<<**endl**;**//输出：2 说明weak\_ptr不影响计数器

1. 动态数组
   1. 动态数组返回元素类型的指针，不能使用begin()、end()、范围for

int**\*** i **=** **new** int**[**10**];**

int j**[**8**];**

auto it1 **=** begin**(**j**);**//正确

auto it2 **=** end**(**j**);** //正确

auto it3 **=** begin**(**i**);**//错误

auto it4 **=** end**(**i**);** //错误

**for(**auto p**:** j**){}** //正确

**for(**auto p**:** i**){}** //错误

* 1. 动态数组初始化，默认情况下使用默认初始化，加()使用值初始化（不能在括号内使用初始化器），还可以使用初始化列表

int **\*** i **=** **new** int**[**2**](**1**);**//编译错

int**\*** a **=** **new** int**[**1**];**

**for(**int i **=** 0 **;** i **<** 1**;++**i**)**cout**<<**a**[**i**]<<**endl**;**//默认初始化，输出随机值

int**\*** b **=** **new** int**[**1**]();**

**for(**int i **=** 0 **;** i **<** 1**;++**i**)**cout**<<**b**[**i**]<<**endl**;**//值初始化，输出0

* 1. unique\_ptr动态数组：unique\_ptr<int[]> u(new int[n]);删除时默认使用delete [];

auto d **=** **[](){**cout**<<**"delete"**<<**endl**;};**//#1

auto d **=** **[](**int**){**cout**<<**"delete"**<<**endl**;};**//#2

auto d **=** **[](**int**\*){**cout**<<**"delete"**<<**endl**;};**//#3

unique\_ptr**<**int**[],**decltype**(**d**)>** a**(new** int**[**2**],**d**);**//只有#3正确，删除器一定需要传入参数，参数类型是智能指针指向的类型的指针

unique\_ptr对于动态数组的删除使用delete[];

string**\*** s **=** **new** string**(**"hi"**);**

unique\_ptr**<**string**>** p**(**s**);**

p**.**reset**();** //#1

p **=** **nullptr;**//#2

cout**<<\***s**<<**endl**;**//对于#1 #2，\*s都输出乱码

* 1. shared\_ptr默认不支持动态数组，因为没有定义delete[]; 不支持下标运算符，不支持指针算术运算。

示例1：

unique\_ptr**<**int**[]>** q**(new** int**[**2**]());**//正确

shared\_ptr**<**int**[]>** p**(new** int**[**2**]());**//错误

示例2：

shared\_ptr**<**int**>** p**(new** int**[**2**](),[](**int**\*** p**){**cout**<<**"delete"**<<**endl**;delete** **[]**p**;});**//自定义删除器，注意和unique\_ptr区别，类型后面不加删除器类型

**for(**int i **=** 0 **;** i **<** 2**;** **++**i**){**

cout**<<**p**[**i**]<<**endl**;** //错误，p不支持下标操作符

cout**<<**p**.**get**()[**i**]<<**endl**;**//正确，使用内置指针类型

**}**

* 1. delete []是从后往前删除

1. allocator
   1. 不能对没有默认构造函数的类型使用动态数组,只申请空间，不初始化
   2. 申请空间支持的函数

|  |
| --- |
| allocator<T> a |
| a.allocator(n) |
| a.deallocator(p,n) |
| a.constructor(p,arg) |
| a.destroy(p) |

* 1. 填充函数

|  |
| --- |
| uninitialized\_fill(b,e,t) |
| uninitialized\_fill\_n(b,n,t) |
| uninitialized\_copy(b,e,b2) |
| uninitialized\_copy\_n(b1.n.b2) |

allocator**<**int**>** a**;**

int n **=** 0**;**

cin**>>**n**;**

int**\*** p **=** a**.**allocate**(**n**);**

uninitialized\_fill\_n**(**p**,**n**,**3**);**

**for(**int i **=** 0 **;**i **<** n **;++**i**)**a**.**destroy**(**p**+**i**);**

a**.**deallocate**(**p**,**n**);**

## 标准库特殊设施

### tuple类

|  |  |
| --- | --- |
| tuple<T1,T2,…,Tn> t;执行值初始化 | tuple**<**int**,**int**,**double**>** t0**;**  cout**<<**get**<**1**>(**t0**)<<**endl**;**//输出0，说明执行的是值初始化，而非默认初始化 |
| tuple<T1,T2,…,Tn> t(t1,t2,…,tn);对于每个类执行直接初始化;explicit | tuple**<**int**,**string**,**string**>** t1**(**1**,**"s1"**,**"s2"**);** |
| make\_tuple(t1,t2,…,tn) | tuple**<**int**,**string**>** t2 **=** make\_tuple**(**1**,**"s3"**);** |
| ==,!=,>,>=,<,<=要求每个元素类型相同，并且数目相等 | tuple**<**int**,**int**,**double**>** t0**;**  tuple**<**int**,**string**,**string**>** t1**(**1**,**"s1"**,**"s2"**);**  tuple**<**int**,**string**>** t3**(**1**,**"s1"**);**  tuple**<**int**,**string**,**string**>** t4**(**1**,**"s1"**,**"s2"**);**  **if(**t1**==**t4**);**//正确  **if(**t0**==**t1**);**//错误，类型不匹配  **if(**t3**==**t1**);**//错误，个数不匹配 |
| get<i>(t) | tuple**<**int**,**string**,**string**>** t1**(**1**,**"s1"**,**"s2"**);**  cout**<<**get**<**2**>(**t1**)<<**endl**;**//输出 s2 |
| tuple\_size<tuple\_type>::value | tuple**<**int**,**string**,**string**>** t1**(**1**,**"s1"**,**"s2"**);**  cout**<<**tuple\_size**<**decltype**(**t1**)>::**value**<<**endl**;**//输出3 |
| tuple\_element<i,tuple\_type>::type | tuple\_element**<**2**,**decltype**(**t1**)>::**type s **=** get**<**2**>(**t1**);**  cout**<<**s**<<**endl**;**//输出s2 |

### bitset

|  |  |
| --- | --- |
| bitset<n> b  bitset<n> b(u)  bitset<n> b(s,start,m,zero,one)  bitset<n> b(cs,start,n,zero,one) | int n **=** 21**;**  bitset**<**n**>** b**;**//错误，n必须是常量  string s **=** "01"**;**  bitset**<**32**>** b**(**s**);**  cout **<<** b**[**0**]** **<<** endl**;**//输出1，说明对于字符串，最低位是bitset的最高位  string s **=** "110"**;**  bitset**<**32**>** b**(**s**,**1**,**2**);**//首先截取“10”赋值给b  cout **<<** b**[**0**]<<**" "**<<**b**[**1**]<<** endl**;**//输出 0 1 ，然后 s 最低位是 b 最高位  string s **=** "110"**;**  bitset**<**32**>** b**(**s**,**1**,**2**,**'1'**,**'0'**);**  cout **<<** b**[**0**]<<**" "**<<**b**[**1**]<<** endl**;**//输出 1 0；字符‘1’使用0表示，字符‘0’使用1表示  string s **=** "112"**;**  bitset**<**32**>** b**(**s**,**1**,**2**,**'1'**,**'2'**);**  cout **<<** b**[**0**]<<**" "**<<**b**[**1**]<<** endl**;**//输出 1 0；字符‘1’使用0表示，字符’2‘使用1表示 |
| b.size()  b.count()  b.any()  b.none()  b.all()  b.test(pos) | bitset**<**32**>** b**(**"11100"**);**  cout**<<**b**.**size**()<<**endl**;** //输出 32  cout**<<**b**.**count**()<<**endl**;** //输出 3；表示3个1  cout**<<**b**.**all**()<<**endl**;** //输出 0；表示不全是1  cout**<<**b**.**any**()<<**endl**;** //输出 1；表示存在1  cout**<<**b**.**none**()<<**endl**;** //输出 0；表示存在1  cout**<<**b**.**test**(**1**)<<**" "**<<**b**.**test**(**2**)<<**endl**;**//0 1 |
| b.flip(pos)  b.flip()  b.set(pos)  b.set()  b.reset(pos)  b.reset() | bitset**<**32**>** b**(**"11100"**);**  b**.**set**(**1**);**  cout**<<**b**[**1**]<<**endl**;**//1  b**.**reset**(**1**);**  cout**<<**b**[**1**]<<**endl**;**//0  b**.**flip**(**1**);**  cout**<<**b**[**1**]<<**endl**;**//1  bitset**<**32**>** b**(**"11100"**);**  b**.**flip**();**  cout**<<**b**<<**endl**;**//反转所有位 |
| b.to\_ulong()  b.to\_ullong()  b.to\_string() | bitset**<**5**>** b**(**"11100"**);**  cout**<<**b**.**to\_ulong**()<<**endl**;**//28  bitset**<**5**>** b1**(**28u**);**  cout**<<**b1**.**to\_string**()<<**endl**;**//11100  bitset**<**5**>** b2**(**28**);**  cout**<<**b2**.**to\_string**()<<**endl**;**//11100；说明默认位无符号解释 |

### regex

|  |  |
| --- | --- |
| regex r(re);  regex r(re,flag);  r.assign(re); | codeblocks和linux编译出错 |
| regex\_match  regex\_search  regex\_replace |  |
| smatch:(cmatch,wcmatch,..)  m.prefix();  m.ready();  m.empty()  m.format(..)  m.size();  m.suffix();  m.length(n)  m.position(n);  m.str(n);  m[n];  m.begin();  m.end();  s\_submatch:  m[i].matched;  m[i].first();  m[i].end();  m[i].str(); |  |
| sregex\_iterator |  |

### 随机引擎和随机分布

* + - 1. 随机引擎生成unsigned随机数，随机分布可以指定分布、类型、分布概率
      2. 随机引擎：

|  |  |
| --- | --- |
| Engine e;  Engine e(s); | default\_random\_engine e**;**  uniform\_int\_distribution**<**unsigned**>** d**(**0**,**9**);**  **for(**int i **=** 0 **;** i **<** 9 **;++**i**){**  cout**<<**d**(**e**)<<**" "**;**//每次结果都一样  **}** |
| e.max()  e.min()  e.seed(s) | default\_random\_engine e**;**  cout**<<**e**.**max**()<<**" "**<<**e**.**min**()<<**endl**;**//2147483646 1 |
| Engine::result\_type  e.discard(u) | default\_random\_engine**::**result\_type a **=** 0xffffffff**;**  int b **=** a**;**  cout**<<**a**<<**endl**;**//4294967295  cout**<<**b**<<**endl**;**//-1；说明result\_type类型使unsigned |

* + - 1. 随机分布类，不同于随机引擎，所有随机分布类是模板类（除了bernoulli\_distribution,伯努利分布），需要类型参数：

|  |  |
| --- | --- |
| 函数对象，使用引擎作为参数，原因是有的分布可能需要多次调用引擎生成随机数 |  |
| d.max()  d.min()  d.reset() | default\_random\_engine e**;**  uniform\_int\_distribution**<**unsigned**>** d**(**0**,**1**);**  cout**<<**e**.**max**()<<**" "**<<**e**.**min**()<<**endl**;**  cout**<<**d**.**max**()<<**" "**<<**d**.**min**()<<**endl**;//0 1** |

* + - 1. 随机发生器：

随机引擎和随机分布的组合。两种方法使程序每次运行生成不同的随机数序列：

|  |  |
| --- | --- |
| 函数内将引擎类设为static。  primer说明的是将引擎类和分布类都声明位static，但是程序表名只需设置引擎即可 | void f**(){**  static default\_random\_engine e**;** //#1  uniform\_int\_distribution**<**unsigned**>** d**(**0**,**9**);**  **for(**int i **=** 0 **;** i **<** 9 **;++**i**){**  cout**<<**d**(**e**)<<**" "**;**  **}**  cout**<<**endl**;**  **}**  f**();**  f**();**//如果#1不加static，则两次调用f将得到相同的序列 |
| 设置种子 | default\_random\_engine e**(**time**(**0**));**//设置种子，只有时间间隔使秒级或者更长，才会不同  uniform\_int\_distribution**<**unsigned**>** d**(**0**,**9**);**  **for(**int i **=** 0 **;** i **<** 9 **;++**i**){**  cout**<<**d**(**e**)<<**" "**;**  **}** |

* + - 1. 其他分布

|  |  |
| --- | --- |
| 正态分布 | default\_random\_engine e**(**time**(**0**));**  normal\_distribution**<>** n**(**4**,**1.5**);**//平均数4，标准差1.5  vector**<**int**>** v**(**9**);**  **for(**int i **=** 0 **;** i **<** 200**;++**i**){**  int num **=** n**(**e**);**  **if(**num**<**v**.**size**()){**  **++**v**[**num**];**  **}**  **}**  **for(**int i **=** 0 **;** i**<** 9**;++**i**){**  cout**<<**i**<<**" :"**<<**string**(**v**[**i**],**'\*'**)<<**endl**;**  **}**  输出：  0 **:\*\*\***  1 **:\*\*\*\*\*\*\*\*\*\*\*\***  2 **:\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  3 **:\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  4 **:\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  5 **:\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  6 **:\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  7 **:\*\*\***  8 **:\*\*** |
| 伯努利分布 | default\_random\_engine e**(**time**(**0**));**  bernoulli\_distribution n**(**.7**);**//伯努利分布，不需要类型参数，默认50/50,可以指定比例  vector**<**int**>** v**(**2**);**  **for(**int i **=** 0 **;** i **<** 10 **;++**i**){**  int num **=** n**(**e**);**  **++**v**[**num**];**  **}**  **for(**int i **=** 0 **;** i**<** 2**;++**i**){**  cout**<<**i**<<**" :"**<<**string**(**v**[**i**],**'\*'**)<<**endl**;**  **}**  输出：  0 **:\*\*\***  1 **:\*\*\*\*\*\*\*** |

### IO再探

格式化IO（控制数据格式和控制空白填充）

|  |  |
| --- | --- |
| del  oct  hex  uppercase  nouppercase  left  right  internal | cout**<<**hex**<<**17**<<**endl**;**//输出： 11  cout**<<**17**<<**endl**;** //输出： 11 ；因为hex状态会保持  cout**<<**showbase**<<**uppercase **;**  cout**<<**hex**<<**17**<<**endl**;**//0X11  cout**<<**noshowbase**<<**nouppercase**;**  cout**<<**hex**<<**17**<<**endl**;**//11 |
| showpoint  showpos  showbase | cout**<<**showpos**<<**17**<<**noshowpos**<<**endl**;**//+17  cout**<<**showpoint**<<(**double**)**17**<<**endl**;**//17.0000  cout**<<**showbase**<<**hex**<<**17**<<**noshowbase**<<**endl**;**//0x11 |
| setw()  setfill()  setprecision()  setbase() | cout**<<**setw**(**10**)<<**setfill**(**'\*'**)<<**setbase**(**16**);** cout**<<**17**<<**endl**;**  cout**<<**17**<<**endl**;**  输出：  **\*\*\*\*\*\*\*\***11  11  //说明setw和setfill只对下一行有效果，setbase保持流状态 |
| scientific  fixed  hexfloat  defaultfloat | cout**<<**100**\***sqrt**(**2**)<<**endl**;**  cout**<<**scientific**<<**100**\***sqrt**(**2**)<<**endl**;**  cout**<<**fixed**<<**sqrt**(**2**)\***100**<<**endl**;**  //cout<<hexfloat<<sqrt(2)\*100<<endl;//codeblocks不能识别  输出：  141.421  1.414214e+002  141.421356  说明精度默认指所有位数，使用了scientific、fixed、hexfloat精度指小数点后位数 |
| skipws  noskipws | char ch**;**  **while(**cin**>>**ch**){**  cout**<<**ch**;**  **}**  默认跳过空格、tab等不可见字符；如输入：a b c 输出：abc  cin**>>**noskipws**;**  **while(**cin**>>**ch**){**  cout**<<**ch**;**  **}**  cin**>>**skipws**;**  设置了noskipws，不跳过不可见字符，输入：a b c 输出：a b c |

面向字符的原生IO

|  |  |
| --- | --- |
| is.get(ch)  is.get()//返回int  os.put(ch) | char ch**=**'c'**;**// #1  cin**.**get**(**ch**);**// #2，输入a  cout**.**put**(**ch**);**  cout**<<**cin**.**get**()<<**endl**;**//#3  //#1：#3输入 a 输出：c97  //#2: #3无输入，输出：a10 因为回车符在缓存内，cin.get直接从缓存读取到了换行，#3无需输入。  cin.get()返回int |
| is.peek()//返回int  is.putback()  is.unget() | peek示例：  char ch**=**'c'**;**  cin**.**get**(**ch**);**//输入：fuck  cout**.**put**(**ch**);**//输出：f  cout**<<**cin**.**peek**()<<**endl**;**//peek返回int，输出117  cout**<<**cin**.**get**()<<**endl**;** //peek并不从缓冲区移除u，输出117  putback()示例：  cin**.**get**(**ch**);**//输入：fuck  cout**.**put**(**ch**);**//f  cin**.**putback**(**'a'**);**//向buffer最后添加a  cout**<<**cin**.**peek**()<<**endl**;**//读取输入流最后一个，输出：a  cin.unget()示例：  cin**.**get**(**ch**);**//输入：fuck  cout**.**put**(**ch**);**//输出：f ，此时cin指向u  cin**.**unget**();**//将cin回退一个字符，指向f  cout**<<**cin**.**get**()<<**endl**;**//输出：f |

面向块的原生IO

|  |  |
| --- | --- |
| is.get(buf,size,delim)  is.getline(buf,size,delim) | char b**[**5**];**  cin**.**get**(**b**,**5**);**//#1  cin**.**getline**(**b**,**5**);**//#2  cin**.**get**();**//#1会得到换行；#2阻塞；原因：cin.get不会把换行从输入流中清除，getline会清除输入流中的换行 |
| is.read()  is.write() | char b**[**5**];**  cin**.**read**(**b**,**5**);**  cout**.**write**(**b**,**5**);** |
| is.gcount()  is.ignore() | char b**[**5**];**  cin**.**get**(**b**,**5**);**//输入：fu  cout**<<**cin**.**gcount**()<<**endl**;**//输出：2 |

随机访问(不能应用于iostream，只能用于fstream和sstream)

|  |  |
| --- | --- |
| tellp()  tellg() |  |
| seekp(pos)  seekg(pos)  seekp(offset,from)  seekg(offset,from) | stringstream ss**;**  string s**;**  cin**>>**s**;**//输入：1234  ss**<<**s**;**  cout**<<**ss**.**tellp**()<<**" "**<<**ss**.**tellg**()<<**endl**;**//输出：4，0；表示输入位置4，输出位置0  ss**.**seekp**(**1**);**//将输入位置移动到位置1，即‘2’所在位置  ss**<<**'a'**;**//写入‘a’  cout**<<**ss**.**str**()<<**endl**;**//输出：1a34  ss**.**seekp**(-**1**,**stringstream**::**end**);**//将输入位置移动到尾后位置前一个位置，即就是‘4’所在位置  ss**<<**'b'**;**//写入‘b’  cout**<<**ss**.**str**()<<**endl**;**//输出：1a3b |

## 多重继承和虚继承

|  |  |
| --- | --- |
| 1. 基类必须已定义并且派生列表不重复 2. 继承而来的构造函数不能有相同的参数列表 3. 构造顺序和派生列表一致 4. 允许派生类继承相同的函数，但是使用时必须指定基类前缀 5. 向基类转换的优先级是相同的，有可能出现二义性 | 示例1：  class B1 **{};**  class B2**;**  class D**:**public B1**,**public B2**{};**//出错，D之前只有声明没有定义  class B2**{};**  示例1：  class B1**{};**  class B2**{};**  class D**:**public B1**,**public B2**,**public B1**{}**//出错：B1重复出现  示例4：  class B1**{**  public**:**  void f**();**  **};**  class B2**{**  public**:**  void f**();**  **};**  class D**:**public B1**,**public B2**{**  void ff**(){**f**();};**//二义性  **};**  示例5：  class B1**{**  **};**  class B2**{**  **};**  class D**:**public B1**,**public B2**{**  **};**  void f**(**B1**&){};**  void f**(**B2**&){};**  D d**;**  f**(**d**);**//二义性  f**((**B1**&)**d**);**//正确 |
| 1. 虚基类隔代影响 2. 虚基类构造优先于非虚基类 3. 虚基类成员可见性。作用域查找顺序由内而外。 | 示例2：  class B**{**  public **:**  B**(){**cout**<<**"B"**<<**endl**;}**  **};**  class B1**:**virtual public B**{**  public**:**  B1**(){**cout**<<**"B1"**<<**endl**;}**  **};**  class B2**:**virtual public B**{**  public **:**  B2**(){**cout**<<**"B2"**<<**endl**;}**  **};**  class D**:**public B1**,**public B2**{**  public**:**  D**():**B2**(),**B1**(){}**  **};**  D**\*** d**=new** D**();**  //输出：  B  B1  B2  说明虚基类先构造，然后按照派生列表顺序构造，所以虽然D构造函数参数列表B2先于B1，但是B1先于B2构造  示例3：  class B**{**  public **:**  int x **=** 0**;** //#1  **};**  class B1**:**virtual public B**{**  public**:**  int x **=** 1**;** //#2  **};**  class B2**:**virtual public B**{**  public **:**  int x **=** 2**;** //#3  **};**  class D**:**public B1**,**public B2**{**  void f**(){++**x**;}**//#2且#3存在，二义性错；#2或者#3存在一个，正确；  **};** |

## 特殊工具与技术

### 内存分配工具（operator new/operator delete）

1. 功能
   1. new：operator new首先申请空间；使用构造函数构造对象；返回内存指针
   2. delete：operator delete首先调用析构对象；然后释放空间

|  |
| --- |
| class A**{**  private**:**  int a **=** 0**;**  public**:**  A**()=default;**  void**\*** **operator** **new(**size\_t st**){**  void**\*** ret **=** malloc**(**st**);**  **if(**ret **==** **nullptr){**  cout**<<**"operator new malloc error"**<<**endl**;**  **throw** bad\_alloc**();**  **}**  **else{**  cout**<<**"malloc bytes: "**<<**st**<<**endl**;**  **return** ret**;**  **}**  **}**  void **operator** **delete(**void**\*** p**,**size\_t st**)**noexcept**{**  free**(**p**);**  cout**<<**"A free bytes: "**<<**st**<<**endl**;**  **}**  **};**  输出：  malloc bytes**:** 4  A free bytes**:** 4 |

1. 原型

|  |
| --- |
| void\* operator new (size\_t);  void\* operator new[](size\_t);  void operator delete(void\* )noexcept;  void operator delete[](void\* )noexcept; |
| void\* operator new(size\_t,nothrow\_t&) noexcept;  void\* operator new[](size\_t,nothrow\_t&) noexcept;  void operator delete(size\_t ,nothrow\_t&) noexcept;  void operator delete[](size\_t,nothrow\_t&) noexcept; |

1. 重载

|  |  |
| --- | --- |
| operator new:   1. 返回void\*，第一个参数size\_t 2. 可以有第二个参数，地址，必须使用定位new 3. 不允许重载void\* operator new(size\_t,void\*) | class B**{**  public**:**  char b**[**3**];**  virtual //#1  **~**B**(){}**  void **operator** **delete(**void**\*** p**,**size\_t st**){**  cout**<<**"B free size: "**<<**st**<<**endl**;**  free**(**p**);**  **}**  **};**  class A**:**public B**{**  private**:**  int a **=** 0**;**  public**:**  A**()=default;**  void**\*** **operator** **new(**size\_t st**){**  void**\*** ret **=** malloc**(**st**);**  **if(**ret **==** **nullptr){**  cout**<<**"operator new malloc error"**<<**endl**;**  **throw** bad\_alloc**();**  **}**  **else{**  cout**<<**"malloc bytes: "**<<**st**<<**endl**;**  **return** ret**;**  **}**  **}**  void **operator** **delete(**void**\*** p**,**size\_t st**)**noexcept**{**  free**(**p**);**  cout**<<**"A free bytes: "**<<**st**<<**endl**;**  **}**  **};**  输出：  注释掉 #1 处的virtual  malloc bytes**:** 8  B free size**:** 3  解释：  **+-------+**  **|** B**::**b **|** //对齐，3字节占用4字节  **+-------+**  **|** A**::**a **|**  **+-------+**  因为析构函数不是virtual的，所以delete会调用B的 **~**B ——**>** **operator** **delete**  保留 #1 处的virtual  malloc bytes**:** 12  A free bytes**:** 12  解释：  **+-------+**  **|** vptr **|----->+-------+**  **+-------+** **|** **typeid|**  **|** B**::**b **|** **+-------+**  **+-------+** **|** **&(~**A**)** **|**  **|** A**::**a **|** **+-------+**  **+-------+** |
| operator delete:   1. 返回void，第一个参数void\* 2. 可以有第二个参数size\_t |

1. 定位new
   1. 定位new传入一个地址，它只会构造对象，不会申请内存；

allocator的constructor和定位new区别：constructor只能在同一个allocator对象申请的内存中构造对象；定位new不必是operator new申请的内存，甚至不必是动态内存。

* 1. 析构函数只是析构对象，不释放内存空间。

|  |
| --- |
| class A**{**  public**:**  int i **=** 0**;**  A**()=default;**  A**(**int ii**):**i**(**ii**){}**  void**\*** **operator** **new(**size\_t st**,**A**\*** p**){** //定位 new 一定不要申请空间  **return** p**;**  **}**  void**\*** **operator** **new(**size\_t st**){**  void**\*** ret **=** malloc**(**st**);**  **if(**ret **==** **nullptr){**  cout**<<**"operator new malloc error"**<<**endl**;**  **throw** bad\_alloc**();**  **}**  **else{**  cout**<<**"malloc bytes: "**<<**st**<<**endl**;**  **return** ret**;**  **}**  **}**  void **operator** **delete(**void**\*** p**,**size\_t st**)**noexcept**{**  free**(**p**);**  cout**<<**"A free bytes: "**<<**st**<<**endl**;**  **}**  **};**  A**\*** aa **=** **new** A**();**//特别注意：当此处使用 A aa 时，delete a的时候会试图 free 栈地址，会出错  A**\*** a**=** **new(**aa**)** A**(**2**);**  cout**<<**aa**->**i**<<**endl**;**  cout**<<**a**->**i**<<**endl**;**  cout**<<**"aa: "**<<**aa**<<**endl**;**  cout**<<**"a : "**<<**a**<<**endl**;**  **delete** a**;**  输出：  malloc bytes**:** 4  2  2  aa**:** 0xbb0d68  a **:** 0xbb0d68  A free bytes**:** 4 |

### RTTI（runtime-time type identification）

1. 应用场景

当基类指针或者引用想要使用派生类的非虚函数时需要用到RTTI

1. dynamic\_cast
   1. 使用形式：

|  |
| --- |
| dynamic\_cast<type\*>(e);  dynamic\_cast<type&>(e);  dynamic\_cast<type&&>(e); |
| D b**;**  **if(**D**\*** d **=** **dynamic\_cast<**D**>**b**){**  **}else{**  cout**<<**"error"**<<**endl**;**  **}**  error**:** cannot **dynamic\_cast** 'b' **(**of type 'class B'**)** to type 'class D' **(**target is **not** pointer **or** reference**)|**  转换参数必须时引用或者指针 |

* 1. 返回

指针类型出错时返回0；

引用类型出错返回bad\_cast；

* 1. 注意点
     1. 一般来说基类类型中至少有一个虚函数
     2. 一般在条件部分执行dynamic\_cast，使得类型转换和类型检查在同一表达式

|  |
| --- |
| class B**{**  public**:**  virtual **~**B**(){}** //#1  **};**  class D**:**public B**{**  public**:**  **~**D**(){}**  **};**  B**\*** d **=** **new** D**();**  **if(**D**\*** dp **=** **dynamic\_cast<**D**\*>(**d**)){**  cout**<<**"dynamic\_cast sucess"**<<**endl**;**//#2  **}**  当 #1 倍注释掉时，报错：cannot **dynamic\_cast** 'd' **(**of type 'class B\*'**)** to type 'class D\*' **(**source type is **not** polymorphic**)|**  当 #1 存在时，正常执行 #2  基类类型中至少有一个虚函数 |

1. typeid
   1. 使用形式

|  |
| --- |
| typeid(对象)； |
| B**\*** bd **=** **new** D**();**  D d**;**  cout**<<typeid(**bd**).**name**()<<**endl**;** //P1B：指向B的指针  cout**<<typeid(\***bd**).**name**()<<**endl**;** //1D： D类型  cout**<<typeid(**d**).**name**()<<**endl**;** //1D： D类型 |

* 1. 返回

返回静态常量类型type\_info

* 1. 注意点

|  |  |
| --- | --- |
| typeid(数组/函数)返回数组/函数，而不是指针 | int**\*** p**;**  int a**[**10**];**  int f**();**  cout**<<typeid(**p**).**name**()<<**endl**;** //Pi： 指向int的指针  cout**<<typeid(**a**).**name**()<<**endl**;** //A10\_i：具有10个类型为int的数组  cout**<<typeid(**f**).**name**()<<**endl**;** //FivE： 返回值为int，参数为空的函数类型 |
| 该类型如果不是类类型或者没有虚函数，则返回其静态类型。 | class B**{};**  class D**:**public B**{};**  B**\*** bd **=** **new** D**();**  cout**<<typeid(\***bd**).**name**()<<**endl**;**//输出： 1B |

1. type\_info

没有默认构造函数，拷贝、移动和赋值运算符也是删除的，唯一能产生type\_info对象只有typeid函数

|  |  |
| --- | --- |
| type\_id返回类型是typeid或其派生类型 | cout**<<typeid(typeid(\***bd**)).**name**()<<**endl**;**  输出：N10\_\_cxxabiv117\_\_class\_type\_infoE |
| 能产生type\_info对象只有typeid函数 | type\_info ti **=** **new** type\_info**();**  出错：error**:** no matching function **for** call to 'std::type\_info::type\_info()'**|**  没有默认构造函数 |

## 模板

### 模板定义

|  |
| --- |
| 1. 定义放在头文件中 2. 函数模板非类型参数，必须是常量表达式或者具有静态生存区 3. 类模板没有类型推断 4. 类内可以直接使用模板名，而不必加类型参数 5. 成员模板 6. 模板默认参数 7. 模板声明 8. 模板实例化声明（会实例化所有成员） |