想了解全部内容请移步至

<http://www.cplusplus.com/>

<http://zh.cppreference.com/w/%E9%A6%96%E9%A1%B5>

<https://www.tutorialspoint.com/c_standard_library/>

<https://msdn.microsoft.com/en-us/library/hh875057.aspx>

<http://pubs.opengroup.org/onlinepubs/9699919799/>

<https://www-s.acm.illinois.edu/webmonkeys/book/c_guide/>

|  |  |
| --- | --- |
| c++98(withdrawn) | <https://www.iso.org/standard/25845.html> |
| c++03(withdrawn) | <https://www.iso.org/standard/38110.html> |
| TR1 | <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1745.pdf>  <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf> |
| c++11(withdrawn) | <https://www.iso.org/standard/50372.html> |
| c++14(withdrawn) | <https://www.iso.org/standard/64029.html>  <http://download.csdn.net/download/sirenxiaohuayuan/9410281>  <https://download.csdn.net/download/sdwhfx/9939257> |
| c++17 | <https://www.iso.org/standard/68564.html> |

更新地址：<https://github.com/jhcarl0814/notes>

若发现文档内容与实际不符，请向GitHub上的作者反馈。

红色的是正则表达式算子<https://zh.wikipedia.org/wiki/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F>

蓝色的单元格是未完成（目前无法完成）的内容。

|  |  |  |
| --- | --- | --- |
| 左值 | 可以出现在赋值语句的左边或右边。 | 变量、下标运算符、解引用运算符、一元前置递增运算符、一元前置递减运算符 |
| 不可修改的左值 |  | const、数组名 |
| 右值 | 只能出现在赋值语句的右边。 | 数字字面值、大部分算术表达式、一元后置递增运算符、一元后置递减运算符 |

# 运算符

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 优先级 |  | 用法 | 计算结果 | 副作用 |  | | 结合顺序 |
|  | ()括号 |  | 改变运算符的结合顺序。 |  |  | |  |
|  |  |  | 编译器自动连接相邻的字符串。 |  |  | |  |
| 1 | ::全局作用域 | ::**标识符** | 访问全局变量。 |  |  | | → |
| ::类作用域 | **类名**::**标识符** |  |  |  | |
| ::命名空间作用域 | **命名空间**::**标识符** |  |  |  | |
| 2 | .**成员名**通过对象选择成员 |  |  |  | 1、不能用于构造函数。 | | → |
| ->**成员名**通过指针选择成员 |  |  |  |  | |
| []数组下标 |  |  |  |  | |
| ()函数调用 |  |  |  |  | |
| ()数据类型构造 | **数据类型名**((**表达式**(,**表达式**)\*)?) |  |  |  | |
| 3 | ++一元后置递增 |  |  |  |  | | ← |
| --一元后置递减 |  |  |  |  | |
| typeid数据类型ID | typeid(**数据类型名**) |  |  |  | |
| typeid运行时数据类型ID | typeid(**表达式**) |  |  | ！！！需要RTTI。 | |
| dynamic\_cast<**数据类型名**>运行时类型检查的强制数据类型转换 | dynamic \_cast<**数据类型名**>(**表达式**) | 若**数据类型**是指针且**表达式**的值等于NULL，得到目标数据类型的NULL；  若**数据类型**是void\*且**表达式**的值是指向有虚函数的类的指针，得到void\*数据类型的指针；  若**数据类型**指向的类是**表达式**的数据类型且**表达式**的数据类型中所有有const、volatile的地方**数据类型**都有，得到目标数据类型的值。  若**数据类型**指向的类是**表达式**的数据类型的基类，得到目标数据类型的值。  若**数据类型**指向的类是**表达式**的数据类型的派生类且**数据类型**指向的数据类型是**表达式**指向的对象的**实际**数据类型或其基类，得到目标数据类型的指针。  若在构造函数或析构函数中使用，若**数据类型**不指向当前类或当前类的基类，未定义。  否则得到NULL或抛出std::bad\_cast异常。 |  | ！！！需要RTTI。  1、**数据类型**是void\*或类的指针或引用。  2、若**数据类型**是指针，**表达式**的值是指向类的指针；若**数据类型**是引用，**表达式**的值是类的左值。 | |
| static\_cast<**数据类型名**>编译时类型检查的强制数据类型转换 | static\_cast<**数据类型名**>(**表达式**) | 若**数据类型**不是引用，为操作数创建1个目标数据类型的临时副本；若**数据类型**是引用，改变操作数的数据类型。 | 调用的转换构造函数或数据类型转换运算符函数产生的副作用。 | 1、浮点数转整数，舍去小数部分。  2、void数据类型的值不能转换为其它数据类型的值。  3、若**数据类型**不是**表达式**的数据类型，   |  | | --- | | **数据类型**和**表达式**的数据类型都是类数据类型时，若转换构造函数被定义且有权限且数据类型转换运算符函数被定义，优先显式调用转换构造函数；若都不满足则出错。 | | **数据类型**是类数据类型且**表达式**的数据类型不是类数据类型时，调用转换构造函数。 | | **数据类型**不是类数据类型且**表达式**的数据类型是类数据类型时，调用数据类型转换运算符函数。 | | **数据类型**和**表达式**的数据类型都不是类数据类型时，  2个都不是指针：基本数据类型之间的转换。  1个是指针且另1个不是指针：只有NULL和0可以被转换。  2个都是指针：  **数据类型**是void\*或**表达式**的数据类型是void\*，得到目标数据类型的指针。  **数据类型**指向的数据类型和**表达式**的数据类型指向的数据类型不能相互转换，出错。两个都是类时，继承且继承关系中只有public关系，算作能相互转换；两个不都是类时，转换构造函数被定义且有权限、数据类型转换运算符函数被定义、或两个都是基本数据类型，算作能相互转换。 |   4、若**数据类型**是**表达式**的数据类型，若复制构造函数被定义且有权限，显式调用复制构造函数。  5、若无法确定使用哪条转换路径，错误。（！！！在菱形继承中，即使使用protect或private也不能去除待选路径。） | |
| reinterpret\_cast<**数据类型名**>非标准转换的强制数据类型转换 | reinterpret\_cast<**数据类型名**>(**表达式**) | 改变操作数的数据类型，不执行计算。 |  | 1、**数据类型**和**表达式**的数据类型是2个指针或1个指针1个整型，或**数据类型**是引用且**表达式**是左值。 | |
| const\_cast<**数据类型名**>对常量进行强制数据类型转换 | const\_cast<**数据类型名**>(**表达式**) | 若**数据类型**不是引用，为操作数创建1个目标数据类型的临时副本；若**数据类型**是引用，改变操作数的数据类型。 |  | 1、**数据类型**和**表达式**的数据类型除了&、const、volatile的部分相同。 | |
| 4 | sizeof对象的大小 |  |  |  | 1、对char数据类型和char数据类型的表达式，得到1。 | |  |
| sizeof数据类型的大小 |  |  |  | ← |
| ++一元前置递增 |  |  |  | ！！！得到的值是左值。 | |
| --一元前置递减 |  |  |  | ！！！得到的值是左值。 | |
| ~一元按位取补 |  |  |  |  | |
| !一元逻辑非 |  |  |  |  | |
| -一元负 |  |  |  |  | |
| +一元正 |  |  |  |  | |
| \*解引用 |  |  |  |  | |
| &取地址 |  |  |  |  | |
| (**数据类型名**)C风格的一元强制数据类型转换 | (**数据类型名**)**表达式**  **数据类型名**(**表达式**) |  |  | 1、按顺序匹配const\_cast、static\_cast、先static\_cast后const\_cast、reinterpret\_cast、先reinterpret\_cast后const\_cast五种方案，但此处static\_cast不受继承关系的访问权限的限制。 | |
|  |  |  |  | **数据类型**不是类数据类型 | **数据类型**是类数据类型 |
| new动态内存分配 | new **数据类型名** | 得到分配到的内存空间的地址。  若分配失败，抛出bad\_alloc异常。 | 调用的构造函数产生的副作用。 | 分配到的内存空间不初始化。 | 1、构造函数被定义且有权限调用。 |
| new **数据类型名**() | 分配到的内存空间初始化为0。 |
| new **数据类型名**(**表达式1**(,**表达式**)\*) | 分配到的内存空间初始化为**表达式1**。 |
| new[]动态数组分配 |  | 得到分配到的内存空间的首地址。  若分配失败，抛出bad\_alloc异常。 | 调用的默认构造函数产生的副作用。 | 1、表达式的数据类型是无符号整型。  2、表达式的值可以是0，此时得到1个有效的非0指针。 | |
| new 在**某数据类型**最后1个\*(const )?后（最左侧的基本数据类型当作1个\*处理）插入1个[**表达式**] | 分配到的内存空间不初始化。 | 1、默认构造函数被定义且有权限调用。  2、表达式的值等于0时不调用默认构造函数。 |
| new 在**某数据类型**最后1个\*(const )?后（最左侧的基本数据类型当作1个\*处理）插入1个[**表达式**]() | 分配到的内存空间初始化为0。 |
| delete动态内存释放 | delete **表达式** | 释放内存空间。 | 调用的析构函数产生的副作用。 | ！！！表达式的值可以是0，不产生任何影响。  1、调用析构函数。  2、调用后表达式的值未定义。 | |
| delete[]动态数组释放 | delete[]**表达式** | 释放内存空间。 | 调用的析构函数产生的副作用。 | 1、调用析构函数。 | |
| 5 | .\***成员名**通过对象指向成员的指针 |  |  |  |  | | → |
| ->\***成员名**通过指针指向成员的指针 |  |  |  |  | |
| 6 | \*乘 |  |  |  |  | | → |
| /除以 |  |  |  | 1、整数除法，a-a/b\*b和a%b相等。   |  |  |  |  | | --- | --- | --- | --- | |  | 被除数和除数同号 | 被除数和除数异号 | | | 求模结果符号和分子相同 | 求模结果符号和分母相同 | | / | 截尾取整 | 截尾取整 | 向下取整 | | % | 符号和它们相同 | 得到的值的符号取决于机器 | | | |
| %求模 |  |  |  |
| 7 | +加 |  |  |  |  | | → |
| -减 |  |  |  |  | |
| 8 | <<按位左移 |  |  |  | 1、**表达式2**的值小于0或大于等于**表达式1**占的位数，未定义。 | | → |
| >>按位右移 |  |  |  |
| 9 | <小于 |  |  |  |  | | → |
| <=小于或等于 |  |  |  |  | |
| >大于 |  |  |  |  | |
| >=大于或等于 |  |  |  |  | |
| 10 | ==关系等于 |  |  |  |  | | → |
| !=关系不等于 |  |  |  |  | |
| 11 | &按位与 |  |  |  |  | | → |
| 12 | ^按位异或 |  |  |  |  | |
| 13 | |按位或 |  |  |  |  | |
| 14 | &&逻辑与 |  |  |  |  | | → |
| 15 | ||逻辑或 |  |  |  |  | |
| 16 | ?:三元条件运算 |  |  |  | ！！！表达式2和表达式3的值都是左值，得到的值是左值；否则得到的值是右值。 | | ← |
| 17 | =赋值 |  |  |  | ！！！得到的值是左值。 | | ← |
| +=加赋值 |  |  |  | 1、对基本数据类型和指针数据类型，相当于**表达式1**=**表达式1** **运算符**(**表达式2**)。 | |
| -=减赋值 |  |  |  |
| \*=乘赋值 |  |  |  |
| /=除以赋值 |  |  |  |
| %=取模赋值 |  |  |  |
| &=按位与赋值 |  |  |  |
| ^=按位异或赋值 |  |  |  |
| |=按位或赋值 |  |  |  |
| <<=按位左移赋值 |  |  |  |
| >>=按位右移赋值 |  |  |  |
| 18 | throw抛出异常 | throw **表达式** | 得到void数据类型。 | 抛出异常。 | 1、**表达式**的数据类型不是void。 | | ← |
| 19 | ,逗号 |  |  |  | ！！！**表达式2**的值是左值，得到的值是左值；**表达式2**的值是右值，得到的值是右值。 | | → |

位运算符的操作数是整数。如果操作数小于0，结果依赖于机器。不能确定位运算符如何处理符号位，所以应使用无符号整数类型。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4 |  |  | not | ! |
| compl | ~ |  |  |
| 10 |  |  | not\_eq | != |
| 11 | bitand | & |  |  |
| 12 | xor | ^ |  |  |
| 13 | bitor | | |  |  |
| 14 |  |  | and | && |
| 15 |  |  | or | || |
| 17 | and\_eq | &= |  |  |
| xor\_eq | ^= |  |  |
| or\_eq | |= |  |  |

求值顺序（操作数计算顺序）

|  |  |
| --- | --- |
| 逻辑与&&、逻辑或||、逗号, | 从左向右 |
| 其它二元运算符、函数调用() | 未定义 |
| 条件运算符?: | 先左，后（左不等于0）中或（左等于0）右 |

|  |
| --- |
| #include<iostream>  using namespace std;  int main()  {  int(\*\*p1)(void)=new (int (\*)(void));不初始化  int(\*\*p2)(void)=new (int (\*)(void))();初始化为0  int(\*\*p3)(void)=new (int (\*)(void))(0);初始化为0  cout<<p1<<' '<<p2<<' '<<p3<<endl;  cout<<\*p1<<' '<<\*p2<<' '<<\*p3<<endl;  int(\*\*p4)(void)=new (int (\*[3])(void));不初始化  int(\*\*p5)(void)=new (int (\*[3])(void))();初始化为0  cout<<p4<<' '<<p5<<endl;  cout<<p4[0]<<' '<<p4[1]<<' '<<p4[2]<<endl;  cout<<p5[0]<<' '<<p5[1]<<' '<<p5[2]<<endl;  return 0;  } |

# 标识符、关键字

标识符作用域

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | 定义 | 声明 |
| 全局作用域 |  |  | 非const变量：加static，只能被当前文件访问；不加static，可以被整个程序访问。  const变量：不加extern，只能被当前文件访问；加extern，可以被整个程序访问。  class只能被当前文件访问。  inline函数只能被当前文件访问。 | const变量：加extern，不初始化。 |
| 类作用域 |  |  |  |  |
| 命名空间作用域 |  |  |  |  |
| 局部作用域 |  | 在函数内定义 |  |  |
| 块作用域 |  |  |  |  |
| 语句作用域 |  |  |  |  |

1、不指定数据类型的根据初始值自动确定变量的数据类型。

<http://stackoverflow.com/questions/2192547/where-is-the-c-auto-keyword-used>

|  |
| --- |
| void f(void){}  int main(void)  {  auto a=f;  } |

标识符声明

1、标识符由小写字母、大写字母、数字、下划线组成，第1个字符是小写字母、大写字母、或下划线。

2、标识符区分大小写。

3、变量名长度没有限制。

|  |  |  |
| --- | --- | --- |
|  | 定义（存储类型修饰符auto、register、static、extern、typedef至多使用1个）  至多出现1次。若标识符被使用，恰好出现1次。 | 声明 |
| 数据类型 | 共用体定义 | 共用体声明 |
| 枚举定义 | 枚举声明 |
|  |  |
| 函数 | 函数定义 | 函数声明 |
|  | |
| 变量（不包括函数） | 确定变量的存储类型，为变量分配存储空间。  extern且初始化 | 向编译器描述变量的数据类型。  extern且不初始化。 |

变量访问限制（访问限定符（类型限定词）可以使用任意多个）

|  |  |
| --- | --- |
| const |  |
| volatile |  |
| mutable |  |

关键字

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| auto  register  static  extern  inline  typedef | const  volatile  mutable | |  | | --- | | void | | bool  false  true | | signed  unsigned  char  wchar\_t  short  int  long  float  double | | union  enum | | class  struct  new  delete  this  public  protected  private  explicit  friend  operator  virtual | | |  | | --- | | if  else  switch  case  default | | while  for  do | | break  continue  return  goto | | throw  try  catch | | sizeof  const\_cast  dynamic\_cast  reinterpret\_cast  static\_cast  typeid | namespace  using | template  typename | asm | export |

# 数据

## 基本数据类型

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | |  | 最小存储空间 | 字面值常量 |
| void | | |  |  |  |
| 算术类型 | 整型 | bool | 布尔型 | / | false、true |
| signed char  unsigned char | 字符型 | 8位 | 可以用转义字符表示不可打印字符（不可见字符）和特殊字符（在语言中有特殊意义的字符）。   |  |  |  |  |  | | --- | --- | --- | --- | --- | | \' |  | '  在字符串常量中可以直接写'。 | | | | \" |  | "  在字符常量中可以直接写"。 | | | | \? |  | ? | | | | \\ |  | \ | | | | \a警报字符 | 告警或铃声 | 引发1个听得见的（铃声）或看得见的警报 | | | | \b退格符 | 退格 | 将光标 | 在当前行中向后移动1个位置 | | | \f走纸符 | 新的一页或换页 | 移动到 | 新的一个逻辑页面的开始位置 | | \n换行符 | 新的一行 | 新的一行的开始位置 | | \r回车符 | 回车 | 当前行的开始位置 | | \t水平制表符 | 水平Tab | 下一个水平制表符位置 | | \v垂直制表符 | 垂直Tab | 下一个垂直制表符位置 | | \**o**  \**oo**  \**ooo** |  | **o**表示1位八进制数字。  \**o**和\**oo**的格式需要接下来的字符不是0~7。   |  | | --- | | #include<stdio.h>  int main()  {  printf("%s\n%s\n%s",  "\0616",输出16  "\616",错误，数太大了  "\618");输出18  return 0;  } | | | | | \x(**h**)\* |  | **h**表示1位十六进制数字。  需要接下来的字符不是0~9或A~F或a~f。 | | |   \t和制表符等价。 |
| wchar\_t | 宽字符型 | 16位 | 在char字面值前加L。 |
| (signed )?short( int)?  unsigned short( int)? | 短整型 | 16位 | 无 |
| signed( int)?|int  unsigned( int)? | 整型 | 16位 | |  |  | | --- | --- | |  | 前缀 | | 八进制 | 0 | | 十进制 | 无 | | 十六进制 | 0x或0X |   字面值整数常量的数据类型默认是int。若比int大则数据类型是long。   |  |  | | --- | --- | |  | 后缀 | | 强制转换为unsigned int | u或U | | 强制转换为long | l或L | | 强制转换为unsigned long | ul或UL或lu或LU | |
| (signed )?long( int)?  unsigned long( int)? | 长整型 | 32位 |
| 浮点型 | float | 单精度浮点型 | 6位有效数字 | 字面值浮点数常量的数据类型默认是double。   |  |  | | --- | --- | |  | 后缀 | | 强制转换为float | f或F | | 强制转换为long double | l或L | |
| double | 双精度浮点型 | 10位有效数字 |
| long double | 扩展精度浮点型 | 10位有效数字 |

#include<cstddef>

|  |  |
| --- | --- |
| size\_t | 某无符号整数类型的别名，可以存储内存中对象的大小（可以存储最大数组的长度）。 |
| ptrdiff\_t | 某有符号整数类型的别名，可以存储同一数组中两个指针之间的差距。 |

void用于表示函数不接收参数、函数不返回值。指针的数据类型是void\*表示提供指针的人没有或不需要确定指针指向什么数据类型。

char是signed char的别名或unsigned char的别名，由编译器决定。

|  |  |
| --- | --- |
| 变量的定义声明、引用声明  **基本数据类型名**( **访问限定符**)\* **变量名**((**表达式**)|=**表达式**)?(,  (**访问限定符** )\***变量名**((**表达式**)|=**表达式**)?)\*;  根据存储类型，在前面加上存储类型修饰符。 | 1、不能声明void数据类型的变量。  2、若是定义声明且访问限定符列表中有const，则必须初始化。  3、变量的数据类型是**基本数据类型**( **访问限定符**)\*。  4、**基本数据类型名 访问限定符**和**访问限定符 基本数据类型名**等价。 |

## （某数据类型的）派生数据类型

（**某数据类型**的）数组

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 定义 | 在**某数据类型**的变量定义语句中，   |  |  | | --- | --- | | 将 | 换为 | | **变量名** | **变量名**[**元素个数**] | | (=**表达式**)? | (={(**表达式**(,  **表达式**)\*)?})? |   或   |  |  | | --- | --- | | 将 | 换为 | | **变量名** | **变量名**[] | | (=**表达式**)? | ={**表达式**(,  **表达式**)\*} | | 1、若初始化且初始值个数小于成员个数，剩余元素初始化为0（基本数据类型和指针数据类型）或调用默认构造函数（类数据类型，默认构造函数被定义且有权限调用）。 |
|  |
|  |  |  |
| 操作 | （对sizeof运算符、&操作符、给数组引用初始化以外的操作，数组名自动转换为1个值不能被修改的指向首元素的指针。<https://zhuanlan.zhihu.com/p/24799071>） |  |
|  |

|  |  |
| --- | --- |
|  | 字面值常量 |
| const char[**字符个数**+1] | "(**字符**)\*"  不能在不是\的字符后使用回车。 |
| const wchar\_t[**字符个数**+1] | 在字符串字面值前加L。 |
| **某数据类型**的指针 | 0  #include<stddef.h>/#include<stdio.h>/#include<stdlib.h>/#include<string.h>/#include<time.h>/#include<wchar.h>  #include<cstdlib>   |  |  | | --- | --- | | NULL | 0 | |

（**某数据类型**的值的）引用

|  |  |  |
| --- | --- | --- |
| 声明  在**某数据类型**的变量声明语句中将**变量名**换为&**别名**。  若变量声明语句不是必须初始化的，必须用左值初始化；若变量声明语句是必须初始化的（最后1个\*后（最左侧的基本数据类型当作1个\*处理）有const），必须用左值或右值初始化。 | 1、引用的类型是在**某数据类型**最后1个\*后（最左侧的基本数据类型当作1个\*处理）插入1个&。  2、**某数据类型**不是void。  3、若用左值初始化，**初始化表达式**的数据类型中所有有const的位置，引用的数据类型中的所有相应位置都有const。   |  | | --- | | int \* \*const \* a=0;  int const \* \* \* &b1=a;错误  int \*const \* \* &b2=a;错误  int \* \*const \* &b3=a;正确  int \* \* \*const&b4=a;错误 | |
| 1、const表示不能通过该引用修改数据。（仍然可能可以通过其他途径修改数据。）   |  | | --- | | #include<iostream>  using namespace std;  void f(int const&a1,int&a2)  {  cout<<a1<<" "<<a2<<endl;  a2=1;  cout<<a1<<" "<<a2<<endl;  }  int main()  {  int a=0;  f(a,a);  } |   2、用typedef给引用的类型创建别名。   |  |  | | --- | --- | | typedef int& intref;  const intref a=1;即intref const a=1;，相当于int&const a=1;，错误 | typedef const int& intref;  intref a=1;正确 | |
| 函数的形参是（**某值**的）引用  在函数原型和函数定义中在**形参名**前插入1个&。 | 1、引用的传参和返回值的规则都与引用的声明的规则相同。   |  | | --- | | void f1(int a){}  void f2(int&a){}  void f3(const int a){}  void f4(const int&a){}  int main()  {  int a;  const int b=0;  f1(a);  f2(a);  f3(a);  f4(a);  f1(b);  f2(b);相当于int&a=b;，错误  f3(b);  f4(b);  f1(0);  f2(0);相当于int&a=0;，错误  f3(0);  f4(0);  } | | int a;  const int b=0;  int f1(){return a;}  const int f2(){return a;}  int&f3(){return a;}  const int&f4(){return a;}  int f5(){return b;}  const int f6(){return b;}  int&f7(){return b;}相当于int&f7=b;，错误  const int&f8(){return b;}  int f9(){return 0;}  const int f10(){return 0;}  int&f11(){return 0;}相当于int&f11=0;，错误  const int&f12(){return 0;}  int main(){} | |
| 函数的返回值是（**某值**的）引用  在函数原型和函数定义中在**函数名**前插入1个&。 |

## 自定义数据类型

枚举（数据）类型

|  |  |  |
| --- | --- | --- |
| 定义声明 | enum( **枚举标记**)?{(**标识符**(=**表达式**)?(,  **标识符**(=**表达式**)?)\*)?}; | 1、表达式是常量表达式，数据类型是整型。  2、第1个标识符不指定值相当于指定为0。除了第1个标识符，没有指定值的标识符的值等于前1个的值加1。 |
|  |  |  |
| 变量的定义、声明 | enum **枚举标记** (**访问限定符** )\***变量名**(=**表达式**)?; | 1、表达式的值可以超出enum数据类型含有的标识符的值的范围。  2、表达式的数据类型是enum **枚举标记**( **访问限定符**)\*。 |
|  |  |  |
|  |  |  |

类

没有类数据类型的字面值。

|  |
| --- |
| 定义  (class |struct )**类名**  {  ( (**访问权限标记**|**函数成员引用声明**|**函数成员定义声明**|**数据成员声明**|**友元函数引用声明**)  )\*}; |
| 声明  class **类名**; |
| 访问标号（访问权限标记）  **访问权限符**:  访问权限符   |  |  |  | | --- | --- | --- | |  | 能访问 | 不能访问 | | public | 所有函数 | / | | protected | 当前类函数成员  友元全局函数、友元其它类函数成员  派生类函数成员，通过派生类句柄 | 不属于左边的任意一种 | | private | 当前类函数成员  友元全局函数、友元其它类函数成员 | 不属于左边的任意一种 |   1、定义时使用class，类定义体中默认访问权限是private；定义时使用struct，类定义体中默认访问权限是public。  2、作用域：从访问权限标记到下一个访问权限标记或从访问权限标记到类定义体结束。  ！！！1个对象的函数成员可以访问1个同类对象的private成员。 |
| 数据成员  1、存储类型、访问限定   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 存储类型 | static | | 非static | | | 访问限定 | const | 非const | const | 非const | | 定义 | 1、static const int数据成员和static const enum数据成员可以在类定义体内（必须是常量表达式）或文件作用域内初始化；其他static数据成员在文件作用域内初始化。  2、非对象成员没有初始化相当于初始化为0。 | | （在成员初始化列表初始化。） |  | | 作用域 | 类域 | |  | | |

函数成员

1、编译器尝试内联在类定义体内定义的函数成员。

2、在类定义体外定义函数成员，在函数名前插入**类名**::。

3、this指针：非static成员函数的1个隐含形参

！！！用this指针判断和处理自赋值。

|  |
| --- |
| 1、隐式使用：**成员名**。（没有同名局部变量时有效。）  2、显式使用：this->**成员名**、(\*this).**成员名**。  3、**类名**::**成员名**。（访问非static成员时。） |

4、存储类型、访问限定

|  |  |  |  |
| --- | --- | --- | --- |
| 存储类型 | 非static | | static |
| 访问限定 | const | 非const | 无 |
| 声明 | 在定义和声明时在(**形参列表**)后插入const。  1、不修改数据成员。  2、不直接调用非const函数成员。 |  | 在定义和声明时在**函数名**前插入static。  1、不直接调用非static函数成员。  2、不访问非static数据成员。 |
| 调用 | 不能通过非const对象调用。 | （不能直接被const函数成员调用。） |  |
| （不能直接被static函数成员调用。） | |
| this指针 | 有   |  |  |  | | --- | --- | --- | | （对象的）访问限定 | const | 非const | | 数据类型 | const **类名**\*const | **类名**\*const | | | 没有 |

5、特殊函数成员

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 构造函数 | 1、函数名是**类名**，存储类型和访问限定是非static非const，不指定返回值数据类型，返回值数据类型是**类名**（只能return;，不能return(void)**表达式**;或return **调用返回值数据类型为void的函数**）（自动返回生成的对象）。  2、函数体执行前，先（按类定义中的声明顺序）调用基类的构造函数，后（按类定义中的声明顺序）调用对象成员的构造函数。  3、成员初始化列表   |  | | --- | | 在(**形参列表**)后插入:(**成员名**|**基类名**)(**实参列表**)(,(**成员名**|**基类名**)(**实参列表**))\*。  1、若没有调用基类的构造函数，自动调用默认构造函数（若没有定义或没权限调用则出错）。  2、若没有调用对象成员的构造函数，自动调用默认构造函数（若没有定义或没权限调用则出错）。  3、不能初始化static数据成员。  4、非static数据成员中，const数据成员和引用数据成员必须用成员初始化列表初始化。 |   4、禁止隐式调用复制构造函数、禁止隐式调用转换构造函数：在**函数名**前插入explicit 。  5、按形参列表分类   |  |  |  | | --- | --- | --- | | 默认构造函数（缺省构造函数） | 形参个数大于等于0。  若形参个数大于0，第1个形参有默认实参。 | 1、当且仅当没有声明任何构造函数时，编译器定义1个默认构造函数（合成默认构造函数）。内联，访问权限是public，内置数据类型的数据成员不初始化，隐式调用基类的默认构造函数、每个对象成员的默认构造函数（若没有定义或没权限调用则出错）。 | | 复制构造函数（拷贝构造函数） | 形参个数大于等于1。  若形参个数大于1，第2个形参有默认实参。  第1个形参的数据类型是**当前类名**( **访问限定符**)\*&。 | ！！！用复制构造函数避免浅拷贝。  1、当且仅当没有声明复制构造函数时，编译器定义1个复制构造函数（合成复制构造函数）。内联，访问权限是public，内置数据类型的数据成员赋值，隐式调用每个对象成员的复制构造函数（若没有定义或没权限调用则出错），数组成员的每个元素用相应的方法复制。 | | 转换构造函数 | 形参个数大于等于1。  若形参个数大于1，第2个形参有默认实参。  第1个形参的数据类型不是**当前类名**( **访问限定符**)\*(&)?。 |  | | 移动构造函数 |  |  | | 其它构造函数 |  |  |   ！！！1个构造函数可能同时是默认构造函数和复制构造函数。1个构造函数可能同时是默认构造函数和转换构造函数。（若默认构造函数不同时是复制构造函数或默认构造函数不同时是转换构造函数，在**函数名**前插入explicit 和没插入没有区别。）   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | | 默认构造函数 | 复制构造函数  没说隐式调用的都是显式调用。 | 转换构造函数  没说隐式调用的都是显式调用。 | 其它构造函数 | | 直接调用（制造的对象是临时的） | | **类名**() | **类名**(**表达式**) | **类名**(**表达式**) | **类名**(**实参列表**) | | 定义变量 | 不初始化/用括号初始化 | **类名** **变量名**;。 | **类名** **变量名**(**表达式**);  如果**表达式**的数据类型是**类名**，则编译器可能跳过复制构造函数（跳过当前这次，但即使不调用复制构造函数也要求有权限调用、有定义）。 | **类名** **变量名**(**表达式**); | **类名** **变量名**(**实参列表**); | |  |  | 隐式用等号初始化为当前类数据类型的值。**类名** **变量名**=**表达式**;  如果**表达式**的数据类型是**类名**，则编译器可能跳过复制构造函数（跳过当前这次，但即使不调用复制构造函数也要求有权限调用、有定义、非explicit）。 | 隐式用等号初始化为其它数据类型的值。**类名** **变量名**=**表达式**; |  | | 定义数组 | 不初始化/初始化不提供值 | **类名** **变量名**[1];  **类名** **变量名**[2]={0}; |  |  |  | |  |  | 隐式初始化为当前类数据类型的值。**类名** **变量名**[]={**表达式**};  如果**表达式**的数据类型是**类名**，则编译器可能跳过复制构造函数（跳过当前这次，但即使不调用复制构造函数也要求有权限调用、有定义、非explicit）。 | 隐式初始化为其它数据类型的值。**类名** **变量名**[]={**表达式**}; |  | | 包含该类对象的类的构造函数在初始化列表中 | | 没有调用该类的构造函数  调用该类的默认构造函数。 | 调用该类的复制构造函数。 | 调用该类的转换构造函数。 | 调用该类的其它构造函数。 | | 该类的派生类的构造函数在初始化列表中 | | 没有调用该类的构造函数  调用该类的默认构造函数。 | 调用该类的复制构造函数。 | 调用该类的转换构造函数。 | 调用该类的其它构造函数。 | | 一元强制数据类型转换运算符（制造的对象是临时的） | |  | 有 | 有 |  | | 编译时类型检查的强制类型转换运算符（制造的对象是临时的） | |  | 有 | 有 |  | | new运算符 | 不初始化 | new **类名** |  |  |  | | 用括号初始化，调用构造函数 | new **类名**() | new **类名**(**表达式**)  如果**表达式**的数据类型是**类名**，则编译器可能跳过复制构造函数（跳过当前这次，但即使不调用复制构造函数也要求有权限调用、有定义）。 | new **类名**(**表达式**) | new **类名**(**实参列表**) | | new[]运算符 | | new **类名**[**元素个数**]  new **类名**[**元素个数**]() |  |  |  |   6、[函数传参和返回值时调用构造函数的规则]与[定义变量时用等号初始化调用构造函数的规则]相同。   |  |  | | --- | --- | | class c  {  public:  c(){}  c(c&a){}  c(int a){}  };  void f(c b){}  int main()  {  c a0;  f(c());相当于c b=c();，需要复制构造函数非explicit  f(c(a0));相当于c b=c(a0);，需要复制构造函数非explicit  f(a0);相当于c b=a0;，需要复制构造函数非explicit×2  f(c(0));相当于c b=c(0);，需要复制构造函数非explicit  f(0);相当于c b=0;，需要转换构造函数非explicit  } | class c  {  public:  c(){}  c(c&a){}  c(int a){}  };  c a0;  c f1(void){return c();}相当于c f1=c();，需要复制构造函数非explicit  c f2(void){return c(a0);}相当于c f2=c(a0);，需要复制构造函数非explicit  c f3(void){return a0;}相当于c f3=a0;，需要复制构造函数非explicit×2  c f4(void){return c(0);}相当于c f4=c(0);，需要复制构造函数非explicit  c f5(void){return 0;}相当于c f5=0;，需要转换构造函数非explicit  int main(){} |   7、直接调用构造函数时制造的对象和调用函数返回1个值时制造的对象，析构函数在函数调用结束后调用。  8、定义变量时用空括号初始化**类名** **变量名**();、用括号初始化为单独的默认构造函数返回值**类名** **变量名**(**类名**());、用括号初始化为单独的复制构造函数返回值**类名** **变量名**(**类名**(**表达式**));，错误。<http://stackoverflow.com/questions/12297021/is-there-any-difference-between-list-x-and-list-x>  9、若无法确定使用哪个类的转换构造函数，错误。   |  |  | | --- | --- | | class c1  {  public:  c1(int a){}  };  class c2  {  public:  c2(int a){}  };  void f(c1 a){}  void f(c2 a){}  int main()  {  f(0);错误  } | class c1  {  public:  explicit c1(int a){}  };  class c2  {  public:  c2(int a){}  };  void f(c1 a){}  void f(c2 a){}  int main()  {  f(0);正确  } |   10、若无法确定使用转换构造函数还是数据类型转换运算符函数，错误。   |  |  |  |  | | --- | --- | --- | --- | | class c  {  public:  c(){}  c(int a){}  operator int(){return 0;}  };  void f(int a1,int a2){}  void f(c a1,c a2){}  int main()  {  c a1;  f(a1,0);错误  } | class c  {  public:  c(){}  explicit c(int a){}  operator int(){return 0;}  };  void f(int a1,int a2){}  void f(c a1,c a2){}  int main()  {  c a1;  f(a1,0);正确  } | class c2;  class c1  {  public:  c1(){}  c1(c2&a){}  };  class c2  {  public:  operator c1()  {  c1 a;  return a;  }  };  void f(c1 a){};  int main()  {  c1 a;  c2 b;  f(b);错误  } | class c2;  class c1  {  public:  c1(){}  explicit c1(c2&a){}  };  class c2  {  public:  operator c1()  {  c1 a;  return a;  }  };  void f(c1 a){};  int main()  {  c1 a;  c2 b;  f(b);正确  } |   11、转换构造函数和赋值运算符函数同时存在，优先执行赋值运算符函数。   |  | | --- | | class c2;  class c1  {  public:  c1 operator=(c2&a){c1 b;return b;}  };  class c2  {  public:  operator c1(){c1 a;return a;}  };  int main()  {  c1 a;  c2 b;  a=b;  } | |
| 析构函数 | 1、函数名是~**类名**，存储类型和访问限定是非static非const，不接收实参（所以不能重载），不返回值，不指定返回值数据类型（只能return;，不能return(void)**表达式**;或return **调用返回值数据类型为void的函数**）。  2、编译器定义1个析构函数（合成析构函数）。按每个非static的对象成员创建顺序的相反顺序调用析构函数。若定义了析构函数，析构函数执行结束时自动调用合成析构函数。  3、若至少2个对象同时被删除，析构函数调用顺序与构造函数执行顺序相反。  4、exit不执行自动对象的析构函数。abort不执行任何对象的析构函数。   |  |  |  |  | | --- | --- | --- | --- | | class c1  {  private:  ~c1(){}  };  class c2:c1  {  private:  ~c2(){}错误  };  int main()  {  return 0;  } | class c1  {  friend class c2;  private:  ~c1(){}  };  class c2:c1  {  private:  ~c2(){}  };  int main()  {  return 0;  } | class c1  {  friend class c2;  private:  ~c1(){}  };  class c2:c1  {  private:  ~c2(){}  };  int main()  {  c2 a;错误  return 0;  } | class c1  {  friend class c2;  private:  ~c1(){}  };  class c2:c1  {  friend int main();  private:  ~c2(){}  };  int main()  {  c2 a;  return 0;  } | |
| 赋值运算符函数 |  |

6、防用户复制对象：将复制构造函数和赋值运算符函数的访问权限设置为private。

7、防用户、函数成员、友元函数复制对象：将复制构造函数和赋值运算符函数的访问权限设置为private，声明，不定义。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 函数成员和数据成员的访问   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 存储类型 | static | | 非static | | | static函数成员 | 1、通过句柄访问。  2、**类名**::**成员名**。 | 3、**成员名**。 | / | | | 非static函数成员 | 1、通过句柄访问。 | 2、**类名**::**成员名**。  3、**成员名**。 | | 其它函数 |  |  | |
| 类和（不属于该类的）函数的关系   |  |  |  | | --- | --- | --- | |  | 函数是类的友元函数 | 不是 | | 声明 | 1、将1个函数声明为1个类的友元函数：在类定义体中写函数原型，在原型前插入friend 。  2、将**1个类**的所有函数声明为**另1个类**的友元函数：在**另1个类**的类定义体中写friend class **1个类**名;。  ！！！不受访问权限标记影响。 |  | |
| 类和类的关系：组合  （略） |
| 类和类的关系：继承  1、派生类不继承基类的构造函数、析构函数、赋值运算符函数。   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 声明  在**派生类名**后插入:(**访问权限符** )?**基类名**(,(**访问权限符** )?**基类名**)\*。  访问权限符   |  |  |  |  | | --- | --- | --- | --- | |  | 基类的public成员 | 基类的protected成员 | 基类的private成员 | | public |  |  | 是派生类的成员，派生类的函数成员无法直接访问。 | | protected | 变为protected成员 |  | | private | 变为private成员 | 变为private成员 |   1、省略访问权限符，访问权限是private。 | | 派生类对象的事件   |  |  | | --- | --- | | 创建 | 1、基类的构造函数。  2、成员初始化值  3、构造函数的函数体。 | | 释放 | 1、析构函数。  2、基类的析构函数。 | | | 虚函数  1、声明：在基类类定义体中的函数原型或函数定义前插入virtual 。构造函数不能声明为虚函数。存储类型是static的函数不能声明为虚函数。  2、在派生类中该函数是虚函数。若基类的析构函数声明为虚函数，派生类的析构函数是虚函数。  ！！！将有虚函数且有派生类的类的析构函数声明为虚函数。  3、纯虚函数   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 声明：在函数头后插入=0。  ！！！若成员函数的访问限定设置了const，=0在const后。  定义：不定义。  1、若派生类没有**重写**基类的纯虚函数，则在派生类中该函数是纯虚函数；否则是虚函数。   |  |  | | --- | --- | | 抽象类：有成员函数是纯虚函数的类。 | 具体类：所有函数成员不是纯虚函数的类。 | | 不能实例化。 | 能实例化。 | | | | 访问成员  1、若派生类**重新定义**（函数成员只需函数名相同）基类成员，则访问基类成员时必须在**成员名**前插入**基类名**::。  2、通过指针、引用访问函数成员   |  |  |  | | --- | --- | --- | |  | 指向基类对象 | 指向派生类对象 | | 基类指针 | （略） | 1、调用非virtual函数成员时调用基类的版本。  2、调用virtual函数成员时调用派生类的版本。  （基类指针只能访问派生类中的基类部分和虚函数。）  3、delete：基类的析构函数不是虚函数，未定义（将基类指针转成派生类指针，正常）；基类的析构函数是虚函数，正常。 | | 派生类指针 | 错误，可能访问不存在的函数成员。 | （略） | | |

## 数据类型转换

整数溢出

|  |  |
| --- | --- |
| unsigned整型 | 溢出后的数以2^(8\*sizeof(type))作模运算。 |
| signed整型 | 未定义 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 隐式数据类型转换 | 混合数据类型表达式中，操作数被转为相同数据类型 | 1、enum类型的值提升到能表示所有enum值的不小于int的最小整型，由机器决定。  2、算术转换。   |  |  |  | | --- | --- | --- | | 整型提升 | bool数据类型转为int，false转为0，true转为1。  signed char、unsigned char、short、unsigned short数据类型的值：如果int能存储该数据类型的所有值，转换为int，否则转换为unsigned int。 | | | 包含short和int的表达式 | short转为int。如果int能存储unsigned short的所有值，unsigned short转为int；否则unsigned short和int转为unsigned int。 | 负数转无符号整型，发生整数溢出。 | | 包含int和long的表达式 | 如果long能存储unsigned int的所有值，unsigned int转为long；否则unsigned int和long转为unsigned long。 | |
| 初始化、赋值、函数调用 | 1、指向非const数据类型的指针可以转换为指向const数据类型的指针。  2、指向非const数据类型的引用可以转换为指向const数据类型的引用。  3、非void数据类型的指针可以转换为void\*。  4、0可以转换为任意指针数据类型。 |
| 条件表达式（if、while、for、do、?:）最后一步转为bool数据类型 | 1、基本数据类型和指针数据类型，0转为false，非0转为true。 |
| 显式数据类型转换（强制数据类型转换） | dynamic\_cast |  |
|  | const\_cast | 1、只能用于添加或删除const限定。 |
|  | static\_cast | 1、用于隐式数据类型转换，编译器不会产生警告。  2、用于将void\*转换为非void数据类型的指针。 |
|  | reinterpret\_cast | 1、重新解释操作数的内容，例：非void数据类型的指针转换为非void数据类型的指针。 |
|  | C风格的一元强制数据类型转换 |  |

# 语句和控制转移

|  |  |  |  |
| --- | --- | --- | --- |
| 声明语句（定义语句） | | | |
| 简单语句 | 空语句 | ; | 1 |
| 表达式语句 | **表达式**; | 1 |
| 结构化语句 | if((**表达式**|**声明语句**))  **语句** | 1 |
| if((**表达式**|**声明语句**))  **语句**  else  **语句** | 1 |
| switch((**表达式**|**声明语句**))  {  ((case **表达式**:|default:)  ( **语句**  )\*)\*} | 1 |
| while((**表达式**|**声明语句**))  **语句** | 1 |
| for(**语句1**(**表达式1**|**声明语句**)?;(**表达式2**)?)  **语句2** | 1 |
| do  **语句**  while(**表达式**); | 1 |
| break; | 1 |
| continue; | 1 |
| return( **表达式**)?; | 1 |
| goto **标签名**; | 1 |
| try  {  ( **语句**  )\*}(  catch(**声明语句**)  {  ( **语句**  )\*})\* | 1 |
| 复合语句（代码块、程序块、块） | | {  ( **语句**  )\*} | 1 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 选择结构   |  |  |  | | --- | --- | --- | | if((**表达式**|**声明语句**))  **语句** | 1、**声明语句**只声明1个变量且初始化，数据类型可以转换为bool，去掉末尾的分号。 |  | | if((**表达式**|**声明语句**))  **语句**  else  **语句** |  | | switch((**表达式**|**声明语句**))  {  ((case **表达式**:|default:)  ( **语句**  )\*)\*} |  | |
| 循环结构   |  |  | | --- | --- | | while((**表达式**|**声明语句**))  **语句** | 1、**声明语句**只声明1个变量且初始化，数据类型可以转换为bool，去掉末尾的分号。  2、若使用**声明语句**，每轮循环开始都初始化变量，每轮循环结束都释放变量。 | | for(**语句1**(**表达式1**|**声明语句**)?;(**表达式2**)?)  **语句2** | 1、**声明语句**只声明1个变量且初始化，数据类型可以转换为整型，去掉末尾的分号。 | |  |  | |
| 跳转   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  |  |  | | goto **标签名**; |  | 1、从非static变量作用域内跳到类类型变量作用域外，自动调用析构函数。（向上跳转且再次遇到变量定义语句，再次自动调用构造函数）。  2、从非static变量作用域外跳到该变量的作用域内，错误。 | |
| 异常   |  |  |  | | --- | --- | --- | | 抛出异常 | throw **表达式**; |  | | 处理异常 | try  {  ( **语句**  )\*}(  catch(**声明语句**)  {  ( **语句**  )\*})\* | 1、**声明语句**只声明1个变量且不初始化，去掉末尾的分号。可以省略变量名。  2、若1个catch匹配，则其它并列的catch不执行。  3、若能接收该异常的try收到该异常，略过try中剩余语句，执行catch，控制流转到所有catch后继续执行。  4、若不能接收该异常的代码块收到该异常，略过代码块中剩余语句，异常继续抛给上一层代码块或调用该函数的函数。  5、若抛出的异常没被catch捕获时有异常被抛出（离开try块时自动调用析构函数，抛出异常），自动调用terminate函数（在<exception>中定义）。throw语句调用复制构造函数，抛出异常  6、若抛出的异常一直没被catch捕获，自动调用terminate函数。  7、因抛出异常离开try块时，在try中创建的其它对象自动调用析构函数。  8、若抛出的异常是类类型且被catch捕获，   |  |  | | --- | --- | |  | 自动调用析构函数时机 | | 不在throw语句中创建 |  | | 在throw语句中创建 | 捕获异常对象的catch结束时 |   复制构造函数   |  |  |  |  | | --- | --- | --- | --- | |  | catch形参类型不是引用 | | catch形参类型是引用 | | 不省略形参名 | 省略形参名 | | 不在throw语句中创建 | 调用2次 | 调用1次 | 调用1次。 | | 在throw语句中创建 | 调用1次 | 不调用 | 不调用 | |  |  |  | | --- | --- | | #include<exception> | | | exception | 最常见的问题 | | #include<stdexcept> | | | runtime\_error | 运行时错误：仅在运行时才能检测到的问题 | | range\_error | 运行时错误：生成的结果超出了有意义的值域范围 | | overflow\_error | 运行时错误：计算上溢 | | underflow\_error | 运行时错误：计算下溢 | | logic\_error | 逻辑错误：可在运行前检测到的问题 | | domain\_error | 逻辑错误：参数的结果值不存在 | | invalid\_argument | 逻辑错误：不合适的参数 | | length\_error | 逻辑错误：试图生成一个超出该类型最大长度的对象 | | out\_of\_range | 逻辑错误：使用一个超出有效范围的值 | | #include<new> | | | bad\_alloc | new无法分配内存 | | #include<type\_info> | | | bad\_cast |  | |

# 函数

若非const对象调用函数成员时匹配至少1个非const函数成员和至少1个const函数成员，优先选用非const版本。

|  |  |  |
| --- | --- | --- |
| class c  {  public:  void f(void)const{}  void f(void){}  };  int main()  {  c a;  ((const c)a).f();  } | class c  {  public:  void f(void)const{}  void f(void){}  };  int main()  {  c a;  ((const c&)a).f();  } | class c  {  public:  void f(void)const{}  void f(void){}  };  int main()  {  c a;  ((const c\*)&a)->f();  } |
| 非const版本 | const版本 | |

|  |
| --- |
| 声明  1、函数名后()或(void)表示不接收实参。  2、在返回值类型前插入inline表示建议内联该函数。除最小的函数外，编译器可能忽略inline。 |
| 带默认实参的形参   |  | | --- | | 声明  1、只在定义或1个声明中指定所有要指定的默认实参。  2、若1个形参指定了默认实参，则它右侧的所有形参都指定默认实参。（若1个形参没有指定默认实参，则它左侧的所有形参都不指定默认实参。） | | 调用  1、若省略1个实参，则它右侧的所有实参都省略。（若1个实参没有省略，则它左侧的所有实参都不省略。） | |
| 函数重载  1、main函数不能重载。  2、任意两个版本的函数的形参类型列表不相同。  3、若调用时因为省略实参导致无法确定使用哪个版本，错误。  4、函数签名：编译器由函数签名区分重载函数。函数签名由函数名、每个形参的数据类型、每个形参是否是引用、（如果是引用）引用的访问限定、（如果是函数成员）函数的访问限定唯一确定。   |  |  | | --- | --- | | void f(const int a){}  void f(int a){} | 错误 | | void f(const int&a){}  void f(int a){} | 正确 | | void f(const int a){}  void f(int&a){} | | void f(const int&a){}  void f(int&a){} |   5、重载运算符函数   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 1、函数名   |  |  | | --- | --- | | 是static\_cast<**数据类型名**> | 不是static\_cast<**数据类型名**> | | operator **数据类型名** | operator**运算符** |   2、重载的数据类型转换运算符函数不指定返回值的数据类型，返回值的数据类型是**数据类型**。  3、不能重载：.、.\*、::、?:。不重载就可以使用：=、一元&、,。必须重载为函数成员：()、[]、->**成员名**、所有赋值运算符、所有数据类型转换运算符。  ！！！可以分别重载为一元运算符和二元运算符：&、\*、+、-。  4、不改变运算符优先级、结合顺序、操作数个数。  5、不能创建新运算符。  6、不能重载所有操作数的数据类型都是[基础数据类型或其派生类型]的运算符函数。  ！！！通过指针选择成员->**成员名**是一元运算符，逗号,是二元运算符。   |  |  |  | | --- | --- | --- | |  | 重载为函数成员 | 重载为全局函数 | | 声明 | （一元运算符）唯一操作数或（二元运算符）左操作数的数据类型是(const|volatile)? **当前类名**(&)?，通过this指针访问。  存储类型是非static。 | ！！！声明为当前类的友元函数可以直接访问protected或private数据成员。 | | 形参个数 | 一元运算符：0。  二元运算符：1。  ()：任意。 | 一元运算符：1。  二元运算符：2。 | | 一元后置递增和一元后置递减在形参列表后增加1个int数据类型的形参。 | |   7、重载的数据类型转换运算符函数的调用：若无法确定使用哪个数据类型转换运算符函数，错误。   |  | | --- | | class c  {  public:  operator int(){}  operator long(){}  };  void f(int a){}  void f(long a){}  int main()  {  c a;  f(a);错误  } |   8、一元&被重载后，可以通过非static成员函数的this指针获取对象地址。  9、->**成员名**运算符<https://kelvinh.github.io/blog/2013/11/20/overloading-of-member-access-operator-dash-greater-than-symbol-in-cpp/>   |  |  |  | | --- | --- | --- | | #include<iostream>  using namespace std;  class c1  {  public:  c1():a(1){}  int a;  };  class c2  {  public:  c1\*p;  c1\*operator->()  {  cout<<"c2 operator->\n";  return p;  }  };  class c3  {  public:  c2\*p;  c2&operator->()  {  cout<<"c3 operator->\n";  return\*p;  }  };  class c4  {  public:  c3\*p;  c3&operator->()  {  cout<<"c4 operator->\n";  return\*p;  }  };  int main()  {  c1 a1;  c2 a2;a2.p=&a1;  c3 a3;a3.p=&a2;  c4 a4;a4.p=&a3;  cout<<a2 ->a<<endl;  cout<<a2.operator->()->a<<endl;  cout<<a4 ->a<<endl;  cout<<a4.operator->()->a<<endl;  } | #include<iostream>  using namespace std;  class c1  {  public:  c1():a(1){}  int a;  };  class c2  {  public:  c1\*p;  c1\*operator->()  {  cout<<"c2 operator->\n";  return p;  }  };  class c3  {  public:  c2\*p;  c2\*operator->()  {  cout<<"c3 operator->\n";  return p;  }  };  class c4  {  public:  c3\*p;  c3&operator->()  {  cout<<"c4 operator->\n";  return\*p;  }  };  int main()  {  c1 a1;  c2 a2;a2.p=&a1;  c3 a3;a3.p=&a2;  c4 a4;a4.p=&a3;  cout<<a2 ->a<<endl;  cout<<a2.operator->() ->a<<endl;  cout<<a4 ->p->a<<endl;  cout<<a4.operator->()->p->a<<endl;  } | #include<iostream>  using namespace std;  class c1  {  public:  c1():a(1){}  int a;  };  class c2  {  public:  c1\*p;  c1\*operator->()  {  cout<<"c2 operator->\n";  return p;  }  };  class c3  {  public:  c2\*p;  c2\*operator->()  {  cout<<"c3 operator->\n";  return p;  }  };  class c4  {  public:  c3\*p;  c3\*operator->()  {  cout<<"c4 operator->\n";  return p;  }  };  int main()  {  c1 a1;  c2 a2;a2.p=&a1;  c3 a3;a3.p=&a2;  c4 a4;a4.p=&a3;  cout<<a2 ->a<<endl;  cout<<a2.operator->() ->a<<endl;  cout<<a4 ->p->p->a<<endl;  cout<<a4.operator->()->p->p->a<<endl;  } | | |
| 类型形参   |  | | --- | | 声明  (typename|class) **形参名**(=**数据类型名**)? | | 使用  在需要数据类型名的地方使用，例：函数形参数据类型、函数返回值数据类型、变量数据类型。 | | 操作  sizeof | |
| 函数模板   |  | | --- | | 声明  在函数头前插入模板形参表。  模板形参表  template<(**类型形参声明**|**变量声明**)(,(**类型形参声明**|**变量声明**))\*>  ~~1、所有模板形参不指定默认实参。~~  2、非类型形参的数据类型不是浮点型、类数据类型。 | | 调用  **函数名**(<(**数据类型名1**|**表达式1**)(,(**数据类型名**|**表达式**))\*>)?((**表达式2**(,**表达式**)\*)?)  1、形式类型实参个数不能多于所有版本的函数的模板形参个数。  2、非类型模板实参是常量表达式。 | |

# 标准库

#include<cctype>

|  |
| --- |
| int isxdigit(int c);  c是'0'~'9'、'a'~'f'、'A'~'F'，返回无意义的非0值；否则返回0。 |
| int isalnum(int c);  c是数字或字母，返回无意义的非0值；否则返回0。 |
| int isdigit(int c);  c是数字，返回无意义的非0值；否则返回0。 |
| int isalpha(int c);  c是字母，返回无意义的非0值；否则返回0。 |
| int islower(int c);  c是小写字母，返回无意义的非0值；否则返回0。 |
| int isupper(int c);  c是大写字母，返回无意义的非0值；否则返回0。 |
| int iscntrl(int c);  c是控制字符（'\a''\b''\f''\n''\r''\t''\v'），返回无意义的非0值；否则返回0。 |
| int isprint(int c);  c是（含空格在内的）可打印字符，返回无意义的非0值；否则返回0。 |
| int isgraph(int c);  c是（除空格以外的）可打印字符，返回无意义的非0值；否则返回0。 |
| int ispunct(int c);  c是（除数字、字母、可打印的空白字符以外的）可打印字符，返回无意义的非0值；否则返回0。 |
| int isspace(int c);  c是' ''\f''\n''\r''\t''\v'，返回无意义的非0值；否则返回0。 |

|  |
| --- |
| int tolower(int c);  c是大写字母，将c转为小写字母返回；否则返回c。 |
| int toupper(int c);  c是小写字母，将c转为大写字母返回；否则返回c。 |

#include<string>

|  |  |
| --- | --- |
| string::size\_type | 某无符号整数类型别名，可以存储任意string对象的长度。 |

|  |  |  |
| --- | --- | --- |
| 构造函数 | 默认构造函数 | string() |
| 复制构造函数 | string(const string&) |
| 转换构造函数 | string(const char\*) |
| 其它构造函数 | string(string::size\_type,char) |
| 输入 | | istream>>string |
| getline(istream,string)  相当于c里的char\*fgets(char\*s,int n,FILE\*stream)。 |
| 输出 | | ostream<<string |
| 赋值 | | string=string |
| 连接 | | string+string  char、char\*+string  string+char、char\*  string+=string、char、char\* |
| 度量 | | bool empty()const  string::size\_type size()const  相当于c里的size\_t strlen(const char\*s)。 |
| 访问元素 | | const char& operator[](string::size\_type)const  char& operator[](string::size\_type)  1、不能访问末尾的'\0'。 |
| 比较 | | operator==、operator!=、operator<、operator<=、operator>、operator>=  相当于c里的int strcmp(const char\*s1,const char\*s2)。  区分大小写。 |
| 数据类型转换 | | const char\* c\_str()const  调用后对string对象的操作可能使返回值失效。 |

#include<vector>

|  |  |
| --- | --- |
| vector::size\_type | 某无符号整数类型别名，可以存储任意vector对象的长度。 |
| vector::difference\_type | 某有符号整数类型别名，可以存储任意2个迭代器对象间的距离。 |

|  |  |  |
| --- | --- | --- |
| 构造函数 | 默认构造函数 | vector<T>() |
| 复制构造函数 | vector<T>(const vector<T>&) |
| 转换构造函数 | vector<T>(vector::size\_type)  多次[调用1次默认构造函数，调用1次复制构造函数]。  默认构造函数被定义且有权限调用，复制构造函数有权限调用。 |
| 其它构造函数 | vector<T>(vector::size\_type,const T&)  多次调用复制构造函数。  第一个参数的数据类型是const T&的复制构造函数被定义且有权限调用。  vector<T>(T\*,T\*)  多次调用复制构造函数。  复制构造函数被定义且有权限调用。 |
| 输入 | |  |
| 输出 | |  |
| 赋值 | | vector<T>=vector<T> |
| 添加元素 | | void push\_back(const T&) |
| 访问元素 | | const T& operator[](vector::size\_type)const  T& operator[](vector::size\_type) |
| 度量 | | bool empty()const  vector<T>::size\_type size()const |
| 比较 | | operator==、operator!=、operator<、operator<=、operator>、operator>= |
| 迭代器 | | vector<T>::const\_iterator begin()const  vector<T>::iterator begin()  vector<T>::const\_iterator end()const  vector<T>::iterator end() |

！！！任何改变vector长度的操作都会使已存在的迭代器失效。

vector<T>::iterator、vector<T>::const\_iterator

|  |
| --- |
| vector<int>v;  vector<const int>::iterator i1=v.begin();错误  vector<int>::const\_iterator i2=v.begin();正确 |

|  |  |  |
| --- | --- | --- |
| 访问元素 | T& operator\*()const | const T& operator\*()const |
| 移动 | vector<T>::iterator& operator++()  vector<T>::iterator operator++(int) | vector<T>::const\_iterator& operator++()  vector<T>::const\_iterator operator++(int) |
| 算术操作 | iterator+difference\_type  difference\_type+iterator  iterator-difference\_type  iterator-iterator | |
| 比较 | operator==、operator!= | |

#include<bitset>

|  |  |  |
| --- | --- | --- |
| 构造函数 | 默认构造函数 | bitset<size\_t>() |
| 复制构造函数 | bitset<size\_t>(const bitset<size\_t>&) |
| 转换构造函数 | bitset<size\_t>(unsigned long)  长度过大，高位补0；长度不够，舍弃高位。  bitset<size\_t>(const char\*)  字符串由'0''1'组成，最右侧字符代表第0位。  bitset<size\_t>(const string&) |
| 其它构造函数 | bitset<size\_t>(const string&,string::size\_type pos,string::size\_type n) |
| 输入 | | istream>>bitset<size\_t> |
| 输出 | | ostream<<bitset<size\_t>  字符串由'0''1'组成，最右侧字符代表第0位。 |
| 度量 | | size\_t size()const |
| 访问元素 | | bitset<size\_t>::reference operator[](size\_t)  bool operator[](size\_t)const |
| 查询 | | bool test(size\_t)const  是否为1。  bool any()const  是否有1。  bool none()const  是否没有1。  size\_t count()const  1的个数。 |
| 修改 | | bitset<size\_t>& reset()  bitset<size\_t>& reset(size\_t)  bitset<size\_t>& set()  bitset<size\_t>& set(size\_t)  bitset<size\_t>& flip()  bitset<size\_t>& flip(size\_t) |
| 数据类型转换 | | unsigned long to\_ulong()const  若对象的长度大于unsigned long的长度，抛出异常。 |

bitset<size\_t>::reference

|  |  |
| --- | --- |
| 修改 | bitset<size\_t>::reference flip() |
| 数据类型转换 | bool operator bool()const |

# 施工区域

|  |
| --- |
|  |

istream cin 标准输入

ostream cout 标准输出

ostream cerr 标准错误

ostream clog

操纵流的函数

操纵标准流的函数

#include<iostream>

流操纵符

|  |  |  |  |
| --- | --- | --- | --- |
| endl | 输出缓冲区中数据，输出1个新行。 |  |  |
| setfill(**表达式**) |  | 1、**表达式**的数据类型为char。 | 黏性 |
| setw(**表达式**) |  | 1、**表达式**的数据类型为整型。 | 非黏性 |

操纵流的函数

#include<string>

getline

<https://stackoverflow.com/questions/39718268/why-do-const-references-extend-the-lifetime-of-rvalues>

<https://stackoverflow.com/questions/1565600/how-come-a-non-const-reference-cannot-bind-to-a-temporary-object>

<https://blogs.msdn.microsoft.com/vcblog/2009/02/03/rvalue-references-c0x-features-in-vc10-part-2/>（<https://www.ptt.cc/man/C_and_CPP/DD8B/M.1470851378.A.1B1.html>、<http://peter.xiau.googlepages.com/rvalue_ch.pdf>）

<https://scottmeyers.blogspot.com/2013/05/lambdas-vs-closures.html>

<http://thbecker.net/articles/rvalue_references/section_01.html>

<https://scottmeyers.blogspot.com/2012/11/universal-references-in-c11-now-online.html>（<https://accu.org/var/uploads/journals/Overload111.pdf>、<https://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Scott-Meyers-Universal-References-in-Cpp11>、<https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers>）