面向对象编程基础

本课程入选教育部产学合作协同育人项目
课程主页:**http://cpp.njuer.org**
课程老师:陈明 **http://cv.mchen.org**

```
ppt和代码下载地址
git clone https://gitee.com/cpp-njuer-org/book
```

第4章

表达式

- 基础
- 算数运算符
- 逻辑和关系运算符
- 赋值运算符
- 递增和递减运算符
- 成员访问运算符
- 条件运算符
- 位运算符
- sizeof运算符
- 逗号运算符
- 类型转换
- 运算符优先级表

表达式由一个或多个 **运算对象(operand)** 组成

- 对表达式求值将得到一个 **结果(result)**
- 字面值和变量是最简单的 **表达式(expression)**
    - 结果是字面值和变量的值
- 把一个 **运算符(operator)** 和一个或者多个运算对象组合起来可以生成复杂的表达式.

基础

基本概念

- 一元运算符(unary operator)
    - 作用于一个运算对象
    - 如取地址符& 解引用符*
- 二元运算符(binary operator)
    - 作用于两个运算对象
    - 如相等运算符== 乘法运算符*
- 三元运算符
    - 作用于三个运算对象
- 函数也是一种特殊的运算符
    - 对运算对象数量没有限制

基本概念

- 一些符号既能做一元运算符,也能做二元运算符
    - 由上下文决定
    - 如*,解引用,乘法.

组合运算符和运算对象

- 对于含有多个运算符的复杂表达式,要理解
    - 运算符的优先级(precedence)
    - 运算符的结合律(associativity)
    - 运算对象的求值顺序(order of evaluation)

```
5+10*20/2
```

运算对象转换

- 表达式求值过程中,运算对象常由一种类型转换成另一种类型
    - 二元运算符要求两个运算对象类型相同
        - 若不同,转换成同一种类型
- 类型转换的规则复杂,但大多数合乎情理容易理解
    - 整数转浮点数,浮点数转整数
    - 指针不能转浮点数
    - 小整数类型（bool char short）提升（promoted）成较大整数类型（主要是int）

重载运算符

- c++语言定义了运算符作用于内置类型和复合类型的运算对象时所执行的操作
- 当运算符作用在类类型的运算对象时,用户可以自行定义其含义.
    - 为已存在的运算符赋予另外一层含义,称为 **重载运算符（overloaded operator）**
        - 如 IO库的>> <<, string 对象 vector对象 迭代器所使用的的运算符
- 重载运算符可自定义运算对象类型和返回值,而无法改变运算对象个数、运算符优先级、结合律.

左值和右值

- C中原意
    - 左值**可以**在表达式左边,右值不能.
- C++
    - 当一个对象被用作**右值**的时候,用的是对象的**值**（内容）；
    - 被用做**左值**时,用的是对象的**身份**（在内存中的位置）.

左值和右值

- 需要右值的地方,可以用左值来代替(使用它的值)
    - 但不能把右值当成左值（也就是位置）使用.
- 要用到左值的运算符
    - 赋值运算符,需要一个（非常量）左值作为左侧运算符对象,结果也是一个左值.
    - 取地址符,作用于一个左值运算对象,返回一个指向该对象的指针,该指针是右值.
    - 内置解引用、下标运算符、迭代器解引用、string vector下标运算符的求值结果是左值.
    - 内置类型和迭代器的递增递减运算符,作用于左值运算对象,其前置版本所得结果也是左值.

使用DECLTYPE 作用于左值右值表达式

- 表达值的求值结果是左值时
    - 得到一个引用类型
    - 如p是int*,解引用运算符&生成左值, `decltype(*p)`的结果是int&
- 表达值的求值结果是右值时
    - 如p是int*,取地址运算符&生成右值,`decltype(&p)`的结果是int**,指向整型指针的指针.

优先级与结合律

复合表达式 是含有两个或多个运算符的表达式.

- 优先级和结合律决定运算对象组合方式.
- 表达式中的括号,使被括起来的部分优先运算.
- 先乘除,后加减
    - 算数运算符满足左结合律:若运算符优先级相同,将按照从左向右顺序组合运算对象.

```
//运算符优先级
3+4*5
//运算符结合律
20-15-3

6+3*4/2+2
//等价于
((6+((3 * 4)/2)))+2
```

括号无视优先级和结合律

- 表达式中括号括起来的部分被当成一个单元来求值,然后再与其他部分按照优先级组合.

```cpp
//exp1.cpp
#include <iostream>
using std::cout; using std::endl;

int main()
{
    // 不同的括号组合导致不同的结果
    cout << (6 + 3) * (4 / 2 + 2) << endl;      // prints 36
    cout << ((6 + 3) * 4) / 2 + 2 << endl;      // prints 20
    cout << 6 + 3 * 4 / (2 + 2) << endl;        // prints 9

    return 0;
}
```

优先级和结合律有何影响

```cpp
//exp2.cpp
#include <iostream>
using std::cout; using std::endl;
using std::cin;

int main()
{
    int ia[]={0,2,4,6,8};
    int last=*(ia+4); //ia[4]
    cout<<last<<endl;

    last=*ia+4; //ia[0]+4
    cout<<last<<endl;

    int v1,v2;
    cin>>v1>>v2;

    return 0;
}
```

练习

在下述表达式的合理位置添加括号,使得添加括号后运算对象的组合顺序与添加括号前一致.

```
(a) *vec.begin()
(b) *vec.begin() + 1


*(vec.begin())
(*(vec.begin())) + 1
```

求值顺序

- 优先级规定了运算对象的组合方式
- 没有说明运算对象按照什么顺序求值

```
//无法知道f1与f2那个先被调用.
int i = f1() * f2();
//对于没有制定执行顺序的运算符,如果表达式指向并修改同一个对象,
//将引发错误并产生未定义的行为
//下面的输出表达式是未定义的
// << 运算符没有明确规定何时以及如何对运算对象求值
int i=0;
cout<<i<<" "<<++i<<endl;//未定义
```

求值顺序

- 有4种运算符明确规定了运算对象的求值顺序
    - 逻辑与&& ,先求左侧运算对象的值,为真才继续求右侧
    - 逻辑或||
    - 条件?:运算符
    - 逗号,运算符

求值顺序、优先级、结合律

- 运算对象的求值顺序与优先级、结合律无关

```
f()+g()*h()+j()
- 优先级规定,g()的返回值和h()的返回值相乘.
- 结合律规定,f()的返回值先与g()和h()的乘积相加,所得结果再与j()的返回值相加.
- 对于这些函数的调用顺序没有明确规定.
```

- 如果f、g、h、j是无关函数,既不会改变同一对象的状态也不执行IO任务,那么调用顺序不受限制.
- 反之,其中几个函数影响到同一对象,则它是一条错误表达式,将产生未定义的行为.

处理复合表达式

- 拿不准的时候多加括号强制让表达式的组合关系符合逻辑要求
- 如果改变某个表达式的值，在表达式的其他地方不再使用这个运算对象
    - 例外：当改变运算的子表达式本身就是另一个子表达式的运算对象
    - 如`*++iter`是很常见的用法

算数运算符

算数运算符（左结合律）

运算符	功能	用法
+	一元正号	+ expr
-	一元负号	- expr
*	乘法	expr * expr
/	除法	expr /* expr
%	求余	expr % expr
+	加法	expr + expr
-	减法	expr - expr

- 优先级：一元运算符 再乘除 再加减

- 算数运算符
- 能作用于任意算数类型以及任意能转换算数类型的类型
- 运算对象和求值结果都是右值

- + -能作用于指针、算数值
    - +-指针或+-算数值, 返回运算对象值的一个（提升后）的副本。
- bool类型不应该参与计算

```
//exp3.cpp
#include <iostream>
using std::cout; using std::endl;
int main()
{
    int i = 1024;
    int k = -i; //k  is -1024

    bool b = true;
    bool b2 = -b; // b2 is true!
    //bool类型不应该参与计算
    //b为true，提升为对应整数1，-b=-1
    //b2=-1≠0，所以b2仍为true
    cout << b << " " << b2 << " " << endl;
    return 0;
}
```

溢出和其它算数运行异常

- 算数表达式有可能产生未定义的结果
    - 数学性质，如除数是0
    - 计算机特点，如溢出，计算结果超出该类型所能表示的范围

```
//overflow.cpp
#include <iostream>
using std::cout; using std::endl;

int main()
{
    short short_value = 32767; // max value if shorts are 16 bits

    short_value += 1; // this calculation overflows
    cout << "short_value: " << short_value << endl;

    return 0;
}
//short_value: -32768
```

- 作用算数类型对象时，+-*/含义分别是加减乘除。
    - 整数相除还是整数。商的小数部分直接弃除。

```
//div.cpp
#include <iostream>
using std::cout; using std::endl;

int main()
{
    // ival1 is 3; result is truncated; remainder is discarded
    int ival1 = 21/6;

    // ival2 is 3; no remainder; result is an integral value
    int ival2 = 21/7;

    cout << ival1 << " " << ival2 << endl;

    return 0;
}
//3 3
```

- 运算符%，取余、取模运算符
    - 负责计算两个整数相除所得的余数
    - 参与取余的运算对象必须是整数类型

```
//exp4.cpp
#include <iostream>
using std::cout; using std::endl;

int main()
{
    int ival = 42;
    double dval = 3.14;
    cout <<  ival % 12 <<endl;// 6
    //ival % dval;// error
    return 0;
}
//6
```

- 除法运算，两个运算对象符号相同则商为正（如果不为0），否则商为负。
    - 早期版本允许负值的商向上或向下取整
    - C++11新标准规定商向0取整（即直接切除小数部分）

- 取余运算，如果m和n是整数且n非0，则表达式(m/n)*n+m%n的求值结果与m相等。
    - 如果m%n不为0，结果符号与m相同
    - 早期版本允许m%n符号匹配n的符号，商向负无穷一侧取整
    - C++11新标准，除了-m导致溢出的特殊情况， 其它时候
        - (-m)/n和m/(-n)都等于-(m/n)
        - m%(-n)等于m%n
        - (-m)%n等于-m%n

- 除法运算 取余运算示例

```
//div2.cpp
#include <iostream>
using std::cout; using std::endl;

int main()
{
    cout<<21%6<<" "<<21/6<<endl;//3 3
    cout<<21%7<<" "<<21/7<<endl;//0 3
    cout<<-21%-8<<" "<<-21/-8<<endl;//-5 2
    cout<<21%-5<<" "<<21/-5<<endl;//1 -4
    return 0;
}
```

练习

```
//写出下列表达式的求值结果。
-30 * 3 + 21 / 5  // -90+4 = -86
-30 + 3 * 21 / 5  // -30+63/5 = -30+12 = -18
30 / 3 * 21 % 5   // 10*21%5 = 210%5 = 0
-30 / 3 * 21 % 4  // -10*21%4 = -210%4 = -2
```

```
//溢出是何含义？写出三条将导致溢出的表达式。
//当计算的结果超出该类型所能表示的范围时就会产生溢出。
short v1 = 32767; ++v1; // -32768
unsigned v2 = 0; --v2;  // 4294967295
unsigned short v3 = 65535; ++v3;  // 0
```

```cpp
//exp5.cpp
#include <iostream>
using std::cout; using std::endl;

int main()
{
    cout<<-30 * 3 + 21 / 5<<endl;  // -90+4 = -86
    cout<<-30 + 3 * 21 / 5<<endl;  // -30+63/5 = -30+12 = -18
    cout<<30 / 3 * 21 % 5<<endl;  // 10*21%5 = 210%5 = 0
    cout<<-30 / 3 * 21 % 4<<endl;  // -10*21%4 = -210%4 = -2

    short v1 = 32767;cout<< ++v1<<endl; // -32768
    unsigned v2 = 0; cout<<--v2<<endl;  // 4294967295
    unsigned short v3 = 65535; cout<<++v3<<endl;  // 0

    return 0;
}
```

逻辑和关系运算符

- 逻辑运算符作用于任意能转换成布尔值的类型
- 关系运算符作用于算数类型或指针类型
- 逻辑运算符和关系运算符
    - 返回值都是布尔型
    - 运算对象和求值结果都是右值
- 值为0的运算对象（算数类型或指针）表示假，否则为真

逻辑和关系运算符

结合律	运算符	功能	用法
右	!	逻辑非	!expr
左	<	小于	expr<expr
左	<=	小于等于	expr<=expr
左	>	大于	expr>expr
左	>=	大于等于	expr>=expr
左	==	相等	expr==expr
左	!=	不相等	expr!=expr
左	&&	逻辑与	expr&&expr
左	||	逻辑或	expr||expr

逻辑与和逻辑或运算符

- 逻辑与&&
    - 当且仅当两个运算对象都为真时结果为真
    - 当且仅当左侧运算对象为真才对右侧对象求值
- 逻辑或||
    - 只要两个运算对象中的一个为真结果就为真
    - 当且仅当左侧运算对象为假才对右侧对象求值
- 都先求左侧运算对象的值，再求右侧运算对象的值
    - 当且仅当左侧运算对象无法确定确定表达式的结果时，才会计算右侧表达式的值。这种策略称为 **短路求值**

```cpp
//逻辑或运算符举例
//exp6.cpp
#include <iostream>
#include <vector>
#include <string>
using std::cout; using std::endl;
using std::vector; using std::string;
int main()
{
    vector<string> text = {"abc","abc","","efg.","hi"};
    //s是对常量的引用，元素既没有拷贝也不会改变
    for(const auto& s:text){
        cout<< s ;//输出当且元素
        //遇到空字符串  或者以句号结束的字符串进行换行
        if(s.empty()||s[s.size()-1]=='.'){
            cout<< endl;
        }
        else
            cout<<" ";//否则用空格隔开
    }
    return 0;
}
```

```
//abc abc
//efg.
//hi
```

- 小技巧，声明为引用类型可以避免对元素的拷贝
    - 又因为不需要做写操作，声明成对常量的引用

```
    for(const auto& s:text){}
```

逻辑非运算符

- 逻辑非运算符!将运算对象的值取反后返回

```cpp
//exp7.cpp
#include <iostream>
#include <vector>
#include <string>
using std::cout; using std::endl;
using std::vector; using std::string;
int main()
{
    vector<int> vec = {1,2,3,4};
    //输出vec的首元素 如果有的话
    //子表达式!vec.empty()当empty()函数返回假时结果为真
    if(!vec.empty())
        cout<< vec[0];//1
    return 0;
}
```

关系运算符

- 关系运算符比较运算对象的大小并返回布尔值
    - 关系运算符满足左结合律

```
//拿i<j的布尔值和k比较
if(i < j < k)//若k大于1则为真

//i<j 且 j<k 为真
if(i < j && j < k)
```

相等性测试与布尔字面值

```
//测试算数对象或指针对象的真值
if(val){}//val非0，条件为真
if(!val){}//val为0，条件为真

//有时会写成
if(val==true){}//val等于1时 条件为真
/*
这种写法存在两个问题
- 写法长 不直接
- 如果val不是布尔值，则比较失去意义
*/
//如val不是布尔值，则true转为val的类型，代码等价于
if(val==1){}
```

- 进行比较运算时，除非比较对象是布尔值，否则不要把布尔字面值true和false作为运算对象

练习

```
//解释在下面的if语句中条件部分的判断过程。
const char *cp = "Hello World";
if (cp && *cp )

//首先判断cp，cp 不是一个空指针，因此cp为真。
//然后判断*cp，*cp 的值是字符'H'，非0。因此最后的结果为真。

//假设i、j和k是三个整数，说明表达式i != j < k的含义。
//这个表达式等于i != (j < k)
//首先得到j < k的结果为true或false，转换为整数值是1或0
//然后判断i不等于1或0 ，最终的结果为bool值。
```

赋值运算符

赋值运算符的左侧运算对象必须是一个可修改的左值

```
int i=0,j=0,k=0;     //初始化而非赋值
const int ci=i;      //初始化而非赋值
//下面赋值语句非法
1024 = k;            //错误，字面值1024是右值
i+j = k;             //错误，算数表达式是右值
ci = k;              //错误，ci是常量（不可修改）左值
//赋值运算符左右类型不同，右侧将转换成左侧运算对象类型
k = 0;               //k=0
k = 3.14;            //k=3
//C++11允许花括号初始值列表作为赋值语句右侧运算对象
//C++11花括号初始值列表不能窄化转换
k = {3.14};          //错误，窄化转化
vector<int> vi;      //初始值为空
vi={0,1,2,3,4,5,6,7,8,9};//十个元素 0-9
```

赋值运算符

```cpp
//exp8.cpp
#include <iostream>
#include <vector>
#include <string>
using std::cout; using std::endl;
using std::vector; using std::string;
int main()
{
    int i=0,j=0,k=0;     //初始化而非赋值
    const int ci=i;      //初始化而非赋值
    //1024 = k;            //错误，字面值1024是右值
    //i+j = k;             //错误，算数表达式是右值
    //ci = k;              //错误，ci是常量（不可修改）左值
    //
    //赋值运算符左右类型不同，右侧将转换成左侧运算对象类型
    k = 0;               //k=0
    cout<<k<<endl;
    k = 3.14;            //k=3
    cout<<k<<endl;
```

```
    vector<int> vi;      //初始值为空
    //k = {3.14};          //错误，窄化转化
    vi={0,1,2,3,4,5,6,7,8,9};//十个元素 0-9
    for(auto a:vi){
        cout<<a<<" ";
    }
    return 0;
}
```

赋值运算满足右结合律

这一点和其它二元运算符不太一样。

```
int ival,jval;
ival = jval = 0;//都被赋值0
```

```
//多重赋值语句
int ival,*pval;
ival=pval=0;//错误，不能把指针赋给int
string s1,s2;
s1=s2="ok"; //字符串字面值"ok"转为string对象
```

- 赋值运算优先级比较低,低于关系运算符的优先级，赋值部分通常应该加上括号。

```
//繁琐写法 容易出错
int i = get_value();//得到一个值
while(i!=42){
    //其它处理...
    i = get_value();//得到剩下的值
}
```

```
// 更好的写法
int i;
while((i=get_value())!=42){
    //其它处理...
}
//不加括号含义变化
```

切勿混淆相等运算符和赋值运算符

```
if(i=j)//j不为0，则为真

if(i==j)//判读i是否等于j
```

复合赋值运算符

任意复合运算`a op= b`等价于`a = a op b`

- 复合运算符只求值一次
- 普通运算符求值两次
    - 右边表达式求值
    - 赋值

```
sum += val;

// += -= *= /= %=  //算数运算符
//<<= >>= &= ^= |= //位运算符
```

练习

```
//在下述语句中，当赋值完成后 i 和 d 的值分别是多少?
int i;   double d;
d = i = 3.5; // i = 3, d = 3.0
i = d = 3.5; // d = 3.5, i = 3
```

```
//执行下述 if 语句后将发生什么情况?
if (42 = i)   // 编译错误。赋值运算符左侧必须是一个可修改的左值。而字面值是右值。
if (i = 42)   // true.
```

```
//下面的赋值是非法的，为什么？应该如何修改?
double dval; int ival; int *pi;
dval = ival = pi = 0;
//p是指针，不能赋值给int，应该改为:
dval = ival = 0;
pi = 0;
```

练习

```
//尽管下面的语句合法，但它们实际执行的行为可能和预期并不一样，为什么？应该如何修改？
if (p = getPtr() != 0)
if (i = 1024)
//
if ((p=getPtr()) != 0)
if (i == 1024)
```

递增和递减运算符

- 作用于左值运算对象
- 前置版本
    - 先加(减)1,将改变后的对象作为求值结果
    - 将对象本身作为左值返回
- 后置版本
    - 加(减)1,求值结果是运算对象改变之前值的副本
    - 将对象原始值的副本作为右值返回

```
int i=0,j;
j = ++i;//j=1,i=1,先加后赋值,
j = i++;//j=1,i=2,先赋值后加
// 优先使用前置版本，后置多一步储存原始值。（除非需要变化前的值）
```

混用解引用和递增运算符

递增优先级较高，*iter++等价于*(iter++)

```
auto iter = vi.begin();
while (iter!=vi.end()&&*iter>=0)
    cout <<*iter++ << endl; // 输出当前值，指针向前移1
```

简洁可以成为一种美德

```
cout<<*iter++<<endl;
//比下面的等价语句更简洁、更少出错
cout<<*iter<<endl;
iter++;
```

运算对象可按任意顺序求值

- 递增和递减运算符会改变运算对象的值，不要在复合表达式里错用这两个运算符。

```
//该循环的行为未定义
while(beg!=s.end()&&!isspace(*beg)){
    *beg = toupper(*beg++);//错误，该赋值语句未定义
}

//赋值运算符两端都用到了beg，右侧改变beg的值
//所以赋值语句是未定义的。可能按照下面方式,或其它方式。
*beg=toupper(*beg);      //先求左侧的值
*(beg+1)=toupper(*beg); //先求右侧的值
```

练习

```
//假设ptr的类型是指向int的指针、vec的类型是vector<int>、ival的类型是int
//说明下面的表达式是何含义？如果有表达式不正确，为什么？应该如何修改？
(a) ptr != 0 && *ptr++
(b) ival++ && ival
(c) vec[ival++] <= vec[ival]

- (a) 判断ptr不是一个空指针，并且ptr当前指向的元素的值也为真，
        然后将ptr指向下一个元素
- (b) 判断ival的值为真，并且(ival + 1)的值也为真
- (c) 表达式有误。C++并没有规定<=运算符两边的求值顺序，
        应该改为vec[ival] <= vec[ival+1]
```

成员访问运算符

- 点运算符，获取类对象的一个成员
    - 成员所属的对象是左值，那么结果是左值
    - 成员所属的对象是右值，那么结果是右值
- 箭头运算符，ptr->mem等价于(*ptr).mem
    - .运算符优先级大于*，所以加括号
    - 作用于一个指针类型的运算对象，结果是左值

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

//不加括号的话
*p.size();  //p是一个指针，没有size成员
```

练习

```
//假设iter的类型是vector<string>::iterator，说明下面的表达式是否合法。
//如果合法，表达式的含义是什么？如果不合法，错在何处？
(a) *iter++;
(b) (*iter)++;
(c) *iter.empty();
(d) iter->empty();
(e) ++*iter;
(f) iter++->empty();
```

- (a)合法。返回迭代器所指向的元素，然后迭代器递增。
- (b)不合法。因为vector元素类型是string，没有++操作。
- (c)不合法。这里应该加括号。
- (d)合法。判断迭代器当前的元素是否为空。
- (e)不合法。string类型没有++操作。
- (f)合法。判断迭代器当前元素是否为空，然后迭代器递增。

条件运算符

- 条件运算符（?:）允许我们把简单的if-else逻辑嵌入到单个表达式中去，按照如下形式：cond? expr1: expr2
- 两个表达式都是左值或能转换成同一种左值，运算结果是左值，否则运算结果是右值。

```
string finalgrade = (grade<60)?"fail":"pass";
```

- 可以嵌套使用，**右结合律**，从右向左顺序组合
    - 可读性下降，不要超过三层

```
finalgrade = (grade > 90) ? "high pass"
    : (grade < 60) ? "fail" : "pass";
//等价于
finalgrade = (grade > 90) ? "high pass"
    :  ((grade < 60) ? "fail" : "pass") ;
```

- 输出表达式使用条件运算符记得加括号，条件运算符优先级太低。

```
cout<< ((grade<60)?"fail":"pass");//pass or fail
cout<< (grade<60)?"fail":"pass";  //1 or 0 !
cout<< grade<60?"fail":"pass";//error cout<60?
```

练习

```
//本节的示例程序将成绩划分为high pass、pass 和 fail 三种，
//扩展该程序使其进一步将 60 分到 75 分之间的成绩设定为`low pass`。
//要求程序包含两个版本：一个版本只使用条件运算符；
//另一个版本使用1个或多个`if`语句。哪个版本的程序更容易理解呢？为什么？
#include <iostream>
using std::cout; using std::cin; using std::endl;
int main()
{   for (unsigned g; cin >> g;)
    {
        auto result = g > 90 ? "high pass" : g < 60 ? "fail" :
                                    g < 75 ? "low pass" : "pass";
        cout << result << endl;
        // -------------------------
        if (g > 90)         cout << "high pass";
        else if (g < 60)    cout << "fail";
        else if (g < 75)    cout << "low pass";
        else                cout << "pass";
        cout << endl;
    }
    return 0;
}
//第二个版本容易理解。当条件运算符嵌套层数变多之后，代码的可读性急剧下降。
//而if else的逻辑很清晰。
```

练习

```
//因为运算符的优先级问题，下面这条表达式无法通过编译。
//指出它的问题在哪里？应该如何修改？
string s = "word";
string pl = s + s[s.size() - 1] == 's' ? "" : "s" ;

//加法运算符的优先级高于条件运算符。因此要改为：
string pl = s + (s[s.size() - 1] == 's' ? "" : "s") ;
```

位运算符

提供检查和设置二进制位的功能。

- 位运算符是作用于**整数类型**的运算对象。
- 二进制位向左移（<<）或者向右移（>>），移出边界外的位就被舍弃掉了。
- 位取反（~）（逐位求反）、与（&）、或（|）、异或（^）

- 位运算符（左结合律）

运算符	功能	用法
~	位求反	~expr
<<	左移	expr<<expr
>>	右移	expr>>expr
&	位与	expr&expr
^	位异或	expr^expr
|	位或	expr|expr

- 运算对象是小整型，则自动提升成较大的整型类型
- 有符号数负值可能移位后变号，所以强烈建议**位运算符仅用于无符号数**。

移位运算符

```
//假设char占8位 int占32位
unsigned char bits=0233; // 八进制0233: 000 0   10 011 011
bits<<8;//bit提升成整型，向左移动八位
// 00000000 00000000 10011011 00000000

bits<<31;//向左移动31位，左边超出边界的位舍弃
// 10000000 00000000 00000000 00000000

bits>>3;//向右移动3位，最右边的3位舍弃
// 00000000 00000000 00000000 00010011
```

位求反运算符

```
//逐位求反,1->0,0->1
unsigned char bits=0227; // 八进制0227: 000 0   10 010 111
~bits;
// 11111111 11111111 11111111 01101000
```

位与、位或、位异或运算符

```
unsigned char b1 = 0145;          //01100101
unsigned char b2 = 0257;          //10101111
b1&b2;//00000000 00000000 00000000 00100101
b1|b2;//00000000 00000000 00000000 11101111
b1^b2;//00000000 00000000 00000000 11001010
```

不要把逻辑运算和位运算搞混

位运算应用：

```
unsigned long quiz1 = 0;    // 每一位代表一个学生是否通过考试
1UL << 12;  // 代表第12个学生通过
quiz1 |= (1UL << 12);   // 将第12个学生置为已通过
quiz1 &= ~(1UL << 12);  // 将第12个学生修改为未通过
bool stu12 = quiz1 & (1UL << 12);   // 判断第12个学生是否通过
```

- 位运算符使用较少，但是重载版本cout、cin用过，
    - 重载运算符的优先级结合律和内置版本一样。
    - 位运算符满足左结合律，优先级介于中间，使用时尽量加括号。

```
cout<<"hi"<<"there"<<endl;
//等同于
((cout<<"hi")<<"there")<<endl;



cout<<42+10;// 正确 +优先级更高
cout<<(10<42);//括号优先级高

cout<<10<42;//错误，试图比较cout和42
//等同于
(cout<<10)<42;
//即
cout<42
```

练习

```
//下列表达式的结果是什么?
unsigned long ul1 = 3, ul2 = 7;
(a) ul1 & ul2
(b) ul1 | ul2
(c) ul1 && ul2
(d) ul1 || ul2
```

- (a) 3
- (b) 7
- (c) true
- (d) true

SIZEOF运算符

- 返回一条表达式或一个类型名字所占的**字节数**。
- 返回的类型是 `size_t`的常量表达式。
- `sizeof`并不实际计算其运算对象的值。
- 两种形式：
    1. `sizeof (type)`，给出类型名
    2. `sizeof expr`，给出表达式

```
// 对char类型或者类型为char的表达式，执行sizeof运算，结果为1
// 对引用类型执行sizeof运算，得到被引用对象所占空间大小
// 对指针执行sizeof运算，得到指针本身所占空间大小
// 对解引用指针执行sizeof运算，得到指针所指对象所占空间大小，指针不需要有效。
// 对数组执行sizeof运算，得到整个数组所占空间大小
// 对string vector对象执行sizeof运算，只返回固定部分的大小，不会计算对象中元素占用多少空间。



int ia[10];
// sizeof(ia)返回整个数组所占空间的大小
// sizeof(ia)/sizeof(*ia)返回数组的大小
constexpr size_t sz = sizeof(ia)/sizeof(*ia);
int arr[sz];//sizeof返回常量
```

练习

```
//推断下面代码的输出结果并说明理由。
//实际运行这段程序，结果和你想象的一样吗？如不一样，为什么？
int x[10];   int *p = x;
cout << sizeof(x)/sizeof(*x) << endl;
cout << sizeof(p)/sizeof(*p) << endl;
```

第一个输出结果是 10。第二个结果1，

```
//在下述表达式的适当位置加上括号，使得加上括号之后的表达式的含义与原来的含义相同。
(a) sizeof x + y
(b) sizeof p->mem[i]
(c) sizeof a < b
(d) sizeof f()

(a) (sizeof x) + y
(b) sizeof(p->mem[i])
(c) sizeof(a) < b
(d) sizeof(f())
```

逗号运算符

- 从左向右依次求值。规定了运算对象的顺序。
- 首先对左侧求值，然后将求值结果丢弃，逗号运算符**结果是右侧表达式**的值。
    - 如果右侧运算对象是左值，那么最终的求值结果也是左值。

```
//逗号表达式经常被用在for语句里
    vector<int>::size_type cnt = ivec.size();
    for(vector<int>::size_type ix=0;
            ix!=ivec.size();++ix,--cnt)
            ivec[ix]=cnt;
```

练习

```
//解释下面这个循环的含义。
constexpr int size = 5;
int ia[size] = { 1, 2, 3, 4, 5 };
for (int *ptr = ia, ix = 0;
    ix != size && ptr != ia+size;
    ++ix, ++ptr) { /* ... */ }

//这个循环在遍历数组ia，指针ptr和整型ix都是起到一个循环计数的功能。
```

类型转换

隐式类型转换

- 自动进行无需介入
- 转换设计为尽可能避免损失精度

下面情况编译器自动转换对象的类型

- 比 int类型小的整数值先提升为较大的整数类型。
- 条件中，非布尔转换成布尔。
- 初始化中，初始值转换成变量的类型。赋值语句，右侧转成左侧。
- 算术运算或者关系运算的运算对象有多种类型，要转换成同一种类型。
- 函数调用时也会有转换。

算术转换

- 运算符的运算对象转成最宽的类型

整型提升

- 把小整数类型转换成较大的整数类型
- 常见的char、bool、short能存在int就会转换成int，否则提升为`unsigned int`
- 较大的char类型，`wchar_t,char16_t,char32_t`提升为整型中`int,long,long long ……`最小的，且能容纳原类型所有可能值的类型。

无符号类型的运算对象

- 整型提升 类型匹配则不进行进一步转换
- 提升后都带符号或都不带符号，小类型运算对象转较大的类型
- 若无符号不小于带符号类型，则带符号运算对象转成无符号的（可能有副作用）。
- 带符号类型大于无符号类型，结果依赖机器。

理解算数转换

- 要理解算数转换，可研究大量的例子。

```
//trans.cpp
#include <iostream>
#include <vector>
#include <string>
using std::cout; using std::endl;
using std::vector; using std::string;
int main()
{
    bool    flag; char               cval;
    short   sval; unsigned short     usval;
    int     ival; unsigned int       uival;
    long    lval; unsigned long      ulval;
    float   fval; double             dval;
```

```
    3.1415926L + 'a';   //'a'提升成int，然后int转成long double
    dval + ival;        //ival->double
    dval + fval;        //fval->double
    ival = dval;        //dval->int,小数部分丢弃
    flag = dval;        //如果dval非0则真
    cval + fval;        //cval提升int，int转float
    sval + cval;        //sval cval ->int
    cval + lval;        //cval -> long
    ival + ulval;       //ival->unsigned int
    usval +ival;        //根据unsigned short he int 所占空间进行提升
    uival +lval;        //根据unsigned int 和long所占空间大小进行转换

    return 0;
}
```

习题

```
//根据本节给出的变量定义，说明在下面的表达式中将发生什么样的类型转换：
(a) if (fval)
(b) dval = fval + ival;
(c) dval + ival * cval;
```

需要注意每种运算符遵循的是左结合律还是右结合律。

(a) fval 转换为 bool 类型
(b) ival 转换为 float ，相加的结果转换为 double
(c) cval 转换为 int，然后相乘的结果转换为 double

习题

```
char cval;
int ival;
unsigned int ui;
float fval;
double dval;
```

请回答在下面的表达式中发生了隐式类型转换吗？如果有，指出来。

```
(a) cval = a + 3;
(b) fval = ui - ival * 1.0;
(c) dval = ui * fval;
(d) cval = ival + fval + dval;
```

- (a) a 转换为 int ，然后与 3 相加的结果转换为 char
- (b) ival 转换为 double，ui 转换为 double，结果转换为 float
- (c) ui 转换为 float，结果转换为 double
- (d) ival 转换为 float，与fval相加后的结果转换为 double，最后的结果转换为char

其它隐式类型转换

- 数组转指针

```
int ia[10];//含有十个整数的指针
int* ip =ia;//ia 转换成指向数组首元素的指针
```

- 指针的转换
    - 常量指针0和字面值nullptr能转换成任意指针类型
    - 指向任意非常量的指针能转化成void*
    - 指向任意对象的指针能转换成const void*
    - 有继承关系的类型间还有另外一种指针转换方式

- 转成bool类型
    - 算数类型 指针类型，非0则真

```
char *cp = get_string();
if(cp)//... //cp非0，则真
while(*p) //...//*cp 非空字符，则真
```

- 转换成常量
    - 允许将指向非常量的指针转换成指向相应常量的指针，引用也如此
    - T是一种类型，我们就能将T的指针或引用，分别转换成指向const T的指针或引用

```
int i;
const int &j =i;//非常量转为const int 的引用
const int *p =&i;//非常量地址转为const 地址
int &r =j,*q=p;//错误，不允许const 转成非常量
//相反的转换不存在，因为试图删除底层const
```

- 类类型的转换

```
string s, t="a value";//字面值转为string
while(cin>>s)//cin 转为bool
//IO库定义了istream到bool类型的转换，cin自动转为bool值
//bool值由输入流的状态决定 最后一次读入成功true 读入失败false
```

显式类型转换（尽量避免）

- 强制类型转换
    - 这种方法本质是危险的，有时不得不使用。

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

命名的强制类型转换

cast-name (expression)

- type 转换目标类型。若是引用，则结果是左值。
- expression 要转换的值
- cast-name是static_cast,dynamic_cast,const_cast,reinterpret_cast中一种。
    - dynamic_cast支持运行时类型识别

static_cast

- 任何明确定义的类型转换，只要不包含底层const，都可以使用。

```
//进行强制类型转换以便执行浮点数除法
double slope = static_cast<double>(j)/i;
//把较大的算数类型赋值给较小的类型时，static_cast非常有用
//告诉编译器不在乎精度损失。使用时警告信息被关闭。


//找回存于void*的指针。编译器无法自动执行的类型转换
void* p = &d;//正确，任何非常量的地址都能存入void*
//正确，将void*转换回初始指针类型
double * dp = static_cast<double>(p);
```

const_cast

- 只能改变运算对象的底层const，一般可用于去除const性质。
- 只有其可以改变常量属性

```
const char *pc;
char *p = const_cast<char*>(pc);//正确，但通过p写值是未定义的


const char *cp;
//错误 static_cast不能转换掉const属性
char *q = static_cast<char*>(cp);
static_cast<string>(cp);//正确，字符串字面值转string
const_cast<string>(cp);//错误，const_cast 只改变常量属性
//const_cast 常用于有函数重载的上下文
```

reinterpret_cast

- 通常为运算对象的位模式提供低层次上的重新解释。

```
int *ip;
char *pc = reinterpret_cast<char*>(ip);
//必须牢记pc所指的真实对象是int而非char
string str(pc)//可能导致异常运行时行为
//类型改了，编译器没有警告出错
//语法上全对，查找问题困难。
```

旧式强制类型转换

- (type) expr
- type (expr)
- 旧式强制转换，表现形式不那么清晰明了，容易看漏，出了问题追踪困难。
- 旧式转换， 如果转换const_cast ,static_cast 合法，行为与对应命名转换一致。
    - 如替换不合法，则与reinterpret_cast 类似的功能。

```
char *pc = (char*)ip;//ip是指向整数的指针
//效果与使用reinterpret_cast 一样。
```

练习

```
//假设 i 是int类型，d 是double类型，
//书写表达式 i*=d 使其执行整数类型的乘法而非浮点类型的乘法。

i *= static_cast<int>(d);
```

```
//用命名的强制类型转换改写下列旧式的转换语句。
int i; double d; const string *ps; char *pc; void *pv;
(a) pv = (void*)ps;
(b) i = int(*pc);
(c) pv = &d;
(d) pc = (char*)pv;

(a) pv = static_cast<void*>(const_cast<string*>(ps));
(b) i = static_cast<int>(*pc);
(c) pv = static_cast<void*>(&d);
(d) pc = static_cast<char*>(pv);
```

练习

说明下面这条表达式的含义。

```
double slope = static_cast<double>(j/i);
```

将`j/i`的结果值转换为`double`然后赋值给`slope`。

运算符优先级表

见4.12

```
::
. -> [] ()
后置++ -- typeid  explicit cast
前置++ -- ~ ! 一元- 一元+ * & () sizeof new new[] delete delete[] noexcept
->* .*
* / %
+ -
<< >>
< <= > >=
== !=
```

运算符优先级表

见4.12

```
位&
位^
位|
&&
||
?:
=
*= /= %= += -= <<= >>= &= |= ^=
throw
,
```

实践课

- 本场景将使用下发的ECS兑换码兑换ECS实例安装vscode界面编写程序,
- 无法兑换服务器的话可直接打开阿里云平台（IDE 界面）做实验
    - 使用vscode网页版或阿里云平台（IDE 界面）或Vim编辑C++代码
    - 使用g++编译运行这段代码
    - 编辑一个 **README.md** 文档,键入本次实验心得.
    - 使用git进行版本控制 可使用之前的gitee代码仓库

```
- 云服务器（Elastic Compute Service,简称ECS）
- Aliyun Linux 2是阿里云推出的 Linux 发行版
- Vim是从vi发展出来的一个文本编辑器.
- g++ 是c++编译器
```

1. 使用兑换码兑换ecs，选择阿里云linux镜像
2. 按照演示，在云服务器上安装vscode。（选做：架设临时博客。）
    - yum install -y git g++ gdb tmux vim
    - git clone https://gitee.com/cpp-njuer-org/book
    - 开放端口
    - bash book/install/cs2.sh
        - vscode网页版，http://ip:8080，密码123456
    - bash book/install/hexo.sh
        - 临时博客地址 http://ip
3. 仿照课本例题，编写位运算程序，设置学生成绩。
4. 仿照课本例题，编程理解算数转换。如dval+ival是什么类型。
5. 仿照课本例题，使用sizeof，以字节为单位，计算常见类型所占内存空间大小。

在服务器上使用vscode网页版或 阿里云平台（IDE 界面）或Vim
编辑c++代码和markdown文档,使用git进行版本控制

```
yum install -y git
yum install -y gcc-c++
```

使用git工具进行版本控制

```
git clone你之前的网络git仓库test(或其它名字)
cd test 进入文件夹test
(clone的仓库,可移动旧文件到目录weekN:  mkdir -p weekN ; mv 文件名 weekN;)
vim test1.cpp
g++ ./test1.cpp 编译
./a.out 执行程序

vim test2.cpp
g++ ./test2.cpp 编译
./a.out 执行程序

vim test3.cpp
g++ ./test3.cpp 编译
./a.out 执行程序
```

```
git add . 加入当前文件夹下所有文件到暂存区
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
git commit –m "test1" 表示提交到本地,备注test1
vim readme.md 键入新内容（实验感想),按ESC 再按：wq退出
git add .
git commit –m "test2" 表示提交到本地,备注test2

git push 到你的git仓库

git log 可看git记录
键入命令并截图,并提交到群作业.
cat test* readme.md
git log
```

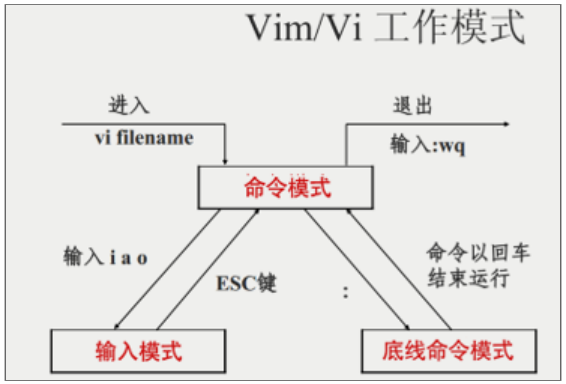
提交

- 截图或复制文字,提交到群作业.
- 填写阿里云平台（IDE 界面）的网页实验报告栏,发布保存.本次报告不需要分享提交
- 填写问卷调查 **https://rnk6jc.aliwork.com/o/cppinfo**

关于使用TMUX

```
sudo yum install -y tmux
cd ~ && wget https://cpp.njuer.org/tmux && mv tmux .tmux.conf
tmux 进入会话 .
前缀按键prefix= ctrl+a,
prefix+c创建新面板,
prefix+"分屏,
prefix+k选上面,prefix+j选下面,
prefix+1选择第一,prefix+n选择第n,
prefix+d脱离会话
tmux attach-session -t 0 回到会话0
```

VIM 共分为三种模式



- 命令模式
  - 刚启动 vim,便进入了命令模式.其它模式下按ESC,可切换回命令模式
    - i 切换到输入模式,以输入字符.
    - x 删除当前光标所在处的字符.
    - : 切换到底线命令模式,可输入命令.
- 输入模式
  - 命令模式下按下i就进入了输入模式.
    - ESC,退出输入模式,切换到命令模式
- 底线命令模式
  - 命令模式下按下:(英文冒号)就进入了底线命令模式.
    - wq 保存退出

VIM 常用按键说明

除了 i, Esc, :wq 之外,其实 vim 还有非常多的按键可以使用.命令模式下:

- 光标移动
  - j下 k上 h左 l右
  - w前进一个词 b后退一个词
  - Ctrl+d 向下半屏 ctrl+u 向上半屏
  - G 移动到最后一行 gg 第一行 ngg 第n行
- 复制粘贴
  - dd 删一行 ndd 删n行
  - yy 复制一行 nyy复制n行
  - p将复制的数据粘贴在下一行 P粘贴到上一行
  - u恢复到前一个动作 ctrl+r重做上一个动作
- 搜索替换
  - /word 向下找word ? word 向上找
  - n重复搜索 N反向搜索
  - :1,$s/word1/word2/g从第一行到最后一行寻找 word1 字符串,并将该字符串取代为 word2

VIM 常用按键说明

底线命令模式下：

- :set nu　显示行号
- :set nonu 取消行号
- :set paste　粘贴代码不乱序

【注：把caps lock按键映射为ctrl,能提高编辑效率.】

MARKDOWN 文档语法

```
# 一级标题
## 二级标题
*斜体* **粗体**
- 列表项
  - 子列表项
> 引用
[超链接](http://asdf.com)
![图片名](http://asdf.com/a.jpg)

|表格标题1|表格标题2|
|-|-|
|内容1|内容2|
```

谢谢