# 一元运算符 和 二元运算符

**一元运算符** ：作用于一个运算对象，如：取地址符（&）、解引用（\*）

**二元运算符** ：作用于两个运算对象，如：加号（+）

一些符号既是一元运算符也是二元运算符，如（\*）

# 左值和右值

**左值（lvalue）** ： 既能够出现在等号左边也能出现在等号右边的变量(或表达式)。

**右值（rvalue）** ： 只能出现在等号右边的变量(或表达式)。

**其它解释：**

左值是一个表达式，它表示一个可被标识的（变量或对象的）内存位置，并且允许使用&操作符来获取这块内存的地址。如果一个表达式不是左值，那它就被定义为右值。

# 为什么括号可以无视优先级和结合律？

因为括号内的部分被当做一个单元来求值。

# m%(-n) 和 (-m)%n 等于多少？

m%(-n) = m%n,

(-m)%n = -(m%n)

# (-m)% (-n)呢？

(-m)% (-n) = (-m)% n = -(m% n)

# && 和 || 的求值有什么特点？

如果&&运算符的第一个操作数是false，就不需要考虑第二个操作数的值了，因为无论第二个操作数的值是什么，其结果都是false。

如果第一个操作数是true，||运算符就返回true，无需考虑第二个操作数的值。

# 如何理解 if (i < j < k) ？

1) 先运行 i < j ，它返回一个布尔值；

2) 将上一步得到的布尔值和k进行比较，所以最终结果是k只要大于 1 就为真

其实正确的写法应该是：

**if** (i < j && j < k)

# 赋值运算的结合律是怎样的？

右结合律

# 赋值运算的返回值是什么？

赋值运算符返回的是其左侧对象的引用：

* 1. **int** sum = 0;  // 这个语句其实是由返回值的，它返回的是 sum的引用

# 为什么赋值运算符的返回值是 左侧对象的引用？

原因有两个：

　　① 允许进行连续赋值

② 防止返回对象（返回对象也可以进行连续赋值（常规的情况，如a = b = c，而不是（a = b） = c））的时候调用拷贝构造函数和析构函数导致不必要的开销，降低赋值运算符的效率。

　　对于第二点原因：如果用”值传递“的方式，虽然功能仍然正确，但由于return语句要把\*this拷贝到保存返回值的外部存储单元之中，增加了不必要的开销，会降低赋值函数的效率。

# 下面的代码的结果是什么？为什么？

* 1. **int** a, b, c;
  2. a = b = c = 15

赋值采用的是右结合律，所以上述连锁赋值被解析为

1. a = (b=( c=15 ));

执行过程如下：

① 这里15先被赋值给c，然后返回 c的引用；

② c的引用 对b进行赋值，即b=c，然后再返回 b的引用

③ 最后 b的引用 对a进行赋值，即 a=b

# 这么写对吗？为什么？int ival, jval; ival = jval = 0;

对，因为 赋值运算 满足 右结合律，jval; ival = jval = 0  的可以拆为：

1) jval = 0

2) ival = jval

# int a = b = c = 1 对吗？

不对，因为前面有int的表示变量定义语句，后面只能是一系列的变量，这些变量可以有初值，但是不能有语句，正确的应该这么写：

1. **int** a, b, c;
2. a = b = c = 1

# i += 1 和 i = i + 1 有什么区别？

i += 1 只求值一次，而 i = i + 1 求值两次，其中包括：

1) tmp = i + 1；

2) i = tmp;

# 对于下面的代码，i和j的值分别是多少？

* 1. **int** i = 1, j=2;
  2. i = j = 10; // j = 1, i = 1:

赋值语句的结合性是右结合的，所以先从最右边的开始，j=10，然后是把j的之赋给i。结果是两个都是10。

# 前置自增运算符 和 后置自增运算符有什么不同？

* 1. **int** i = 0, j;
  2. j = ++i; // j = 1, i = 1:
  3. j = i++; // j = 1, i = 2:

# 为什么前置自增（自减）运算符 效率高？

**前置运算符**，先将自身递增，然后返回自身；

**后置运算法**，先创建自身的一个副本，而后自身递增，然后返回副本。

因此若非必须，尽量使用前置自增（自减）运算符。

# 如何理解 \*pbeg++ ？

后置自增运算符 比 解引用运算符优先级高，因此 \*pbeg++ = \*(pbeg++)：

1) pbeg++ 先自增，然后返回 pbeg 的原值给 解引用运算符；

2) 解引用运算符 获取

# 如何看待如下代码？

* 1. **while** (beg != s.end() && !isspace(\*beg))
  2. \*beg = toupper(\*beg++);

这个结果是未定义的，因为大多数运算符都没有规定运算对象的求值顺序，这在一般情况下是无伤大雅的，但是如果一条子表达式改变了某个运算对象的值，里一条子表达式又要用到该表达式的话，运算对象的求值顺序就很关键了。

在上面的代码中，赋值运算符 左右两端 的运算对象都用到了 beg，并且右侧的运算对象还改变了 beg的值，所以该赋值语句是未定义的，编译器可能按照下面任意一种思路处理该表达式：

1. \*beg = toupper(\*beg);  // 如果先求左侧的值；
2. \*(beg + 1) = toupper(\*beg);  // 如果先求右侧的值；

也可能采取其它什么方式处理它。

# 什么是短路求值？

作为"&&"和"||"操作符的操作数表达式，这些表达式在进行求值时，只要最终的结果已经可以确定是真或假，求值过程便告终止，这称之为短路求值（short-circuit evaluation）:

假如expr1和expr2都是表达式，并且expr1的值为0，在下面这个逻辑表达式的求值过程中：

expr1 && expr2

expr2将不会进行求值，因为整个逻辑表达式的值已经可以确定为0。

类似地，如果expr1的值不是0，那么在下面的这个[逻辑表达式](https://baike.baidu.com/item/%E9%80%BB%E8%BE%91%E8%A1%A8%E8%BE%BE%E5%BC%8F)的求值过程中：

expr1 || expr2

expr2将不会进行求值，因为整个逻辑表达式的值已经确定为1。

# 解引用运算符 和 成员访问运算符 谁的优先级高？

成员访问运算符 的高。

# 成员访问运算符 和 箭头运算符

* 1. string s1 = "a string", \*p = &s1;
  2. auto n = s1.size();  // run the size member of the string s1
  3. n = (\*p).size();  // run size on the object to which p points
  4. n = p->size();  // equivalent to (\*p).size()

# 用条件运算符判断成绩是否及格（是返回pass，否返回fail）

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

# 用条件运算符将成绩分为三档：优秀(大于90)、合格、不及格

* 1. finalgrade = (grade > 90) ? "high pass"  : (grade < 60) ? "fail" : "pass";

上面语句的执行顺序：

先执行 (grade > 90) ? "high pass" ，

如果为真则返回 "high pass"，

如果为假，则执行 (grade < 60) ? "fail" : "pass";

# 使用条件运算符时需要注意什么？

1) 嵌套的条件运算符最好不要超多3条，要不然可读性太差；

2) 条件运算符优先级很低，因此嵌套了条件运算子表达式时，通常需要在它的两端加上括号，看下面的代码：

1. cout << ((grade < 60) ? "fail" : "pass"); // prints pass or fail
2. cout << (grade < 60) ? "fail" : "pass"; // prints 1 or 0!
3. cout << grade < 60 ? "fail" : "pass"; // error: compares cout to 60

对于第二个语句，它相当于：

1. cout << (grade < 60); //输出 0 或 1
2. cout ? "fail" : "pass"; // 根据 cout的值是true还是false产生对应的输出

对于第三个语句，它相当于：

1. cout << grade;  // “<”的优先级比”<<”低，所以先输出 grade
2. cout < 60 ? "fail" : "pass";  // 然后比较 cout和60，决定输出fail还是pass

# sizeof运算符返回的是什么？

返回的是 **字节数**

# sizeof运算符的求值发生在什么阶段？

最开始的c标准规定sizeof只能编译时求值，后来c99又补充规定sizeof可以运行时求值。

# sizeof(类型)、sizeof(引用)返回什么？

**sizeof(类型)** ：该类型所占的字节数，比如char是1字节，int是4字节

**sizeof(引用)** ：该引用所引用的对象所占空间

# sizeof(指针)、sizeof(解引用指针)、sizeof(数组名)返回什么？

**sizeof(指针)** ：指针本身的大小，64位OS是8字节

**sizeof(解引用指针)** ：该指针指向对象所占空间

**sizeof(数组名)** ：整个数组所占空间（**注意！sizeof运算符不会讲数组转换为指针来处理！**）

# sizeof(string对象) 、sizeof(vector对象)返回什么？

只计算固定部分的大小，不会计算对象中的元素占了多少空间

TODO：比较复杂，后面再看

# 既然有了strlen，为什么还要sizeof？

# 逗号表达式 的结果哪一个？

整个逗号表达式的值为 **系列中最后一个表达式的值**。

* 1. #include <iostream>
  2. **using** **namespace** std;
  4. **int** main()
  5. {
  6. **int** i, j;
  8. j = 10;
  9. i = (j++, j+100, 999+j);
  11. cout << i;
  13. **return** 0;
  14. }

结果： 1010，

对于语句 i = (j++, j+100, 999+j);  等号右侧的的一系列运算从左开始按顺序执行，先是j+，此时j为 11 ，因为整个逗号表达式的值为 **系列中最后一个表达式的值**，所以  j+100 并没有起作用，(j++, j+100, 999+j); 的值为 999+j，即 1010

# 什么是 隐式转换？

隐式转换是系统跟据程序的需要而自动转换的，无需程序员介入，有时甚至不需要程序员了解。

# OS是怎么执行 int ival = 3.541 + 3; 的？

1) 3被转换成了double类型，即 3.0

2) 3.541 和 3.0 相加，即 6.541

3) 执行 ival的初始化，此时ival的类型无法改变，只能将 6.541 转为 int，即 6 赋给ival。

# 什么时候会发生隐式转换？

1) 在条件中，非布尔转换为 布尔类型；

2) 在初始化过程中，初始值 转换为 变量的类型；

3) 在赋值语句中，右侧对象 转换为 左侧运算对象 的类型；

4) 算数运算 或 关系运算时，若有多种类型，需要转换为 同一种类型。

5) 将数组名赋给一个指针时

1. **int** ia[10];   // array of ten ints
2. **int**\* ip = ia;  // ia被转换为指向数组首元素的指针

6) 将指针、算数类型作为条件时：

1. **char** \*cp = get\_string();
2. **if** (cp) /\* ... \*/ // true if the pointer cp is not zero
3. **while** (\*cp) /\* ... \*/ // true if \*cp is not the null character

7) 将非常量类型的指针赋给常量指针时

1. **int** i;
2. **const** **int** &j = i;  // 非常量转换为 const int
3. **const** **int** \*p = &i;  // 非常量的地址转换为 const地址
4. **int** &r = j, \*q = p;  // 错误: 不允许const转换成非常量

值得注意的是，算术类型之间的转换被设计得尽可能避免损失精度，如：

auto ival = 3.541 + 3 // ival为double或float

# 整形提升

整形提升负责把 小整数类型 转换为较大的整数类型，对于bool, char, signed char, unsigned char, short, and unsigned short 来说，只要它们所有的值都能存在int里，它们就会转换为int类型。

对于较大的char 类型(wchar\_t, char16\_t, and char32\_t)将提升成 unsigned int, long, unsigned long, long long和unsigned long long中 最小的一种类型，前提是转换后的类型要能容纳原类型的所有值。

# 无符号类型的算数转换

有符号数 和 无符号数进行算术运算的时候，有无符号数会被转换为无符号数。

# 类型转换的例子

* 1. **bool** flag; **char** cval;
  2. **short** sval; unsigned **short** usval;
  3. **int** ival; unsigned **int** uival;
  4. **long** lval; unsigned **long** ulval;
  5. **float** fval; **double** dval;
  6. 3.14159L + 'a';  // 'a' 提升为int, 然后该 int转换为 long double
  7. dval + ival;  // ival converted to double
  8. dval + fval;  // fval converted to double
  9. ival = dval;  // dval converted (by truncation) to int
  10. flag = dval;  // if dval is 0, then flag is false, otherwise true
  11. cval + fval;  // cval promoted to int, then that int converted to float
  12. sval + cval;  // sval and cval promoted to int
  13. cval + lval;  // cval converted to long
  14. ival + ulval;  // ival converted to unsigned long
  15. usval + ival;  // promotion depends on the size of unsigned short and int
  16. uival + lval;  // conversion depends on the size of unsigned int and long

# 两个char类型进行运算会发生什么？

来看《C和指针》里面的一个例子：

1. **char** a, b, c;
2. **char** a = '1';
3. **char** b = '2'
4. c = a + b; // a和b的值被提升为普通整型， 然后再执行加法运算,加法运算的结果将被截短， 然后再存储于c中

# char字符变量可以和int整型数值加减么？ 需要强制类型转换吗？

可以进行加减，因为char类型是可以转换为int类型的（计算过程中自定进行转换，不需要强制转换的）。

# 显示类型转换

C语言的类型转换比较自由，但也带来了一些问题，这些问题大多由程序员自行控制和解决。对于庞大的C++语言机制而言，这种简单粗暴的类型转换方式显然是个巨大的负担，因此C++引入4种类型转换运算符，更加严格的限制允许的类型转换，使转换过程更加规范。

**static\_cast**

1 ) 除了 包含底层const，都可以使用static\_cast

2 ) 对于较大算术类型转换为较小的算术类型，static\_cast可以关闭警告信息。

3 ) static\_cast对于编译器无法自动执行的类型转换也非常有用，例如可以找回存于void\*指针中的值:

1. **void**\* p = &d; // 正确: 任何非常量对象的地址都能存入 void\*
2. // 正确: 将void\* 转换回初始的指针类型
3. **double** \*dp = **static\_cast**<**double**\*>(p);

4 ) static\_cast它能在内置的数据类型间互相转换，对于类只能在有联系的指针类型间进行转换。可以在继承体系中把指针转换来、转换去，但是不能转换成继承体系外的一种类型

**dynamic\_cast**

① 转换类型必须是一个指针、引用或者void\*，用于将基类的指针或引用安全地转换成派生类的指针或引用；

　　② dynamic\_cast在运行期间强制转换，运行时进行类型转换检查；

　　③ 对指针进行转换，失败返回null，成功返回type类型的对象指针，对于引用的转换，失败抛出一个bad\_cast ，成功返回type类型的引用；

　　④ dynamic\_cast不能用于内置类型的转换；

　　⑤ 用于类的转换，基类中一定要有virtual定义的虚函数（保证多态性），不然会编译错误。

**const\_cast**

const\_cast 唯一能做的 就是 改变**底层const**、**volatile**属性（也只有它能做到）；

1. **const** **char** \*pc;
2. **char** \*p = **const\_cast**<**char**\*>(pc); // 正确: 但是通过p写值是未定义的行为

const\_cast 还常用于函数重载的上下文中。

**reinterpret\_cast.**

reinterpret\_cast为运算对象的位模式提供较低层次上的重新解释，即它会产生一个新的值，这个值会有与原始表达式有完全相同的比特位。使用reinterpret\_cast非常危险，本质上依赖于机器。

# C++提供的几个显示类型转换运算符分别发生在什么阶段？

其他三种都是编译时完成的，dynamic\_cast是运行时处理的。

|  |  |
| --- | --- |
| static\_cast | 编译期 |
| dynamic\_cast | 运行时 |
| const\_cast | 编译期 |
| reinterpret\_cast | 编译期 |

# 旧式的强制类型转换

# 为什么在 C++ 中不提倡 C 风格的强制类型转换？

1 ) C++ 具有继承，**static\_cast**  和  **dynamic\_cast**可表示向下转型。使用多个关键字来做不同的 casting，能减少歧义，令代码更清晰易理解。

2) 表现形式不够清晰，追踪起来比较困难

# 为什么要尽量避免 强制类型转换

因为强制类型转换干扰了 正常的类型检查。