**1 程序的编写编译和运行**

**1.1 C++标准(版本)**

C++98/03/11/14/17/20/23/26....

从实际工作出发，建议初学者先基于C++11/14系统学习一遍C++语法(相比之前的老版本，C++11引入了大量的新特性，C++14只是对C++11做了细微的改进和补充)。然后对于更新的C++版本，只需要学对应版本引入的新特性即可, 并且等用到时再学。

**1.2 编写、编译和运行**

1. #include <iostream>
3. **int** get\_max(**int** a, **int** b)
4. {
5. std::cout << "the function is called" << std::endl;
7. **return** a > b ? a : b;
8. }
10. **int** main()
11. {
12. **int** a = 10, b = 8, max = 0;
14. max = get\_max(a, b);
15. std::cout << max << std::endl;
17. **return** 0;
18. }

**2 程序的其他几个组成部分**

**2.1 #include预处理器编译指令**

整个编译包括：预处理(对源代码进行一些替换或添加文本的工作)、编译(将预处理后的源代码文件转换成汇编指令文件)、汇编(将汇编指令文件转换为机器指令文件)、链接(将机器指令文件和其依赖的一些其他库文件，一起合并生成可执行文件)。

例如#include <iostream> 该编译指令将在编译的预处理阶段将iostream文件的内容添加到程序中。实际上，iostream文件的内容将取代程序中的代码行#include <iostream>， 而是将源代码文件和iostream文件组合成一个复合文件。

**2.2 namespace**

C++标准库头文件没有扩展名；不过使用时需要声明namespace，使头文件中定义的名称对程序可用。C++ 标准函数库的所有元素都被声明在一名为std(读作standard)的namesapce中。

在使用时可以声明：

1. std::cout；//直接使用
2. **using** **namespace** std；//使得std这个namespace中的所有名称都可以用，这是偷懒的做法。
3. **using** std::cout；//只声明使用的名称，其中::为作用域限定符

C++为什么要引入这个nasmpace? 给代码指定名称，有助于降低命名的冲突，比如通过使用std::cout，编译器就会到命名空间std中去找cout，而不会到别的地方去找。

**2.3 程序的多文件结构**

模块文件(.cpp)：函数定义等

头文件(.h)：函数声明等

主程序文件(.cpp)：main函数

**3 变量和常量**

**3.1 内存访问模型和变量的引入**

可以将物理内存DDR看成一个个大小为8bit(1Byte)的空间排列而成，每个空间都有编号，即地址。

那么如何使用内存？使用变量！变量的定义格式为如下。

1. /\*其中变量的类型是给编译器看的，编译器根据类型来为变量预留相应大小的空间，并将变量关联到所预留的内存单元。变量名用来指代这段内存空间。\*/
2. VariableType varName;
3. /\*初始化是一个很好的编程习惯，若不初始化相应内存单元所存储的值还是之前的遗留值。\*/
4. VariableType varName = InitialValue;
5. /\*C++11引入的初始化方法，称为列表初始化。这种初始化方法可以避免缩窄转换错误。\*/
6. VariableType varName {InitialValue};

这里需要注意：

1. 如果需要在一条语句中定义多个同一类型的变量，中间用逗号隔开。
2. 不要使用已有关键字作为变量名。变量名可包含数字、字母、下划线，但不能以数字打头(尽量不要适用单个字母；尽量按照一定的规范来命名，比如采用第一个单词首字母小写，第二个单词开始首字母大写的骆驼命名法)。
3. 可以使用sizeof确定变量的长度。

**3.2 局部变量和全局变量**

可以根据是否在函数内部定义来区分，定义在函数内部的叫局部变量，定义在函数外部的叫全部变量。

全局变量分为普通全局变量和静态(static)全局变量，它们在程序运行期间一直存在(有效)，程序结束后才会被释放回收。其中对于普通全局变量，不仅可以在它所定义的文件里访问它，在别的文件中也可以访问，不过需要使用extern关键字再次声明它。对于静态全局变量，只能在它所应以的文件中访问它，在别的文件中不能访问。

局部变量分为普通局部变量和静态(static)局部变量，都在它们所定义的函数运行期间有效。其中对于普通局部变量，函数每次调用结束后，会被释放回收。对于静态局部变量，函数调用结束后，不会别释放，直到程序结束后才会被释放。

**3.3 使用const定义常量**

const是一个限定符，它用来限定一个变量不允许改变，它将一个对象转换成一个常量。

C语言中const修饰的局部变量示例：

1. **int** main()
2. {   /\*C语言中const修饰的局部变量保存在内存中的栈区，初始化该
3. 常量后就不能再通过定义时的该变量名去修改内存中的内容。\*/
4. **const** **int** a = 10;
5. //a = 100; 报错，const限制了不能通过变量名a修改内存中的内容
6. **int**\* p = &a;
7. \*p = 100;  //但是可以通过变量a的地址来修改
9. printf("a = %d\n", a);  // a = 100
10. **return** 0;
11. }

C语言中const修饰全局变量示例：

1. /\*C语言中const修饰的全局变量在内存中的常量区(只读的不能被更改)\*/
2. **const** **int** b = 10;
3. **int** main()
4. {
5. //b = 100；报错，const限定不能通过变量名b去修改
6. **int**\* p = &b;
7. //\*p = 100; 报错，也不能通过地址去修改，在Linux系统中会报段错误
9. printf("b = %d\n", b);  // a = 100
10. **return** 0;
11. }

C语言中const全局变量的跨文件访问示例：

1. //模块文件.c
2. **const** **int** num = 1;  //const修饰的全局变量默认是外部链接属性
3. //主文件.c
4. **int** main()
5. {
6. **extern** **const** **int** num;
7. print("num = %d",  num);
8. }

C++中const修饰局部变量(赋值为常量时)示例：

1. **int** main()
2. {
3. /\*在C++中const修饰的局部变量存在符号表(变量名作为key，初始值作为value)里面\*/
4. **const** **int** a = 10;
5. //a = 10;报错，const限制了不能通过变量名修改
6. /\*对a取地址，编译器会产生一个临时变量来保存a的值：
7. int tmp = a; int \*p = &tmp.所以p实际指向的是栈区tmp这个
8. 临时变量的地址，所以\*p实际修改的还是tmp\*/
9. **int**\* p = (**int** \*)&a;//C++类型转换比较严格，a类型为const int\*，所以需要强转
10. \*p = 100;  //指针p指向a的地址，通过指针来修改
11. cout << a << endl;   //还是10，从符号表中取值，永远是10
13. **return** 0;
14. }

C++中const修饰局部变量(赋值为变量)示例：

1. **int** main()
2. {
3. **int** b = 2;
4. /\*const修饰的局部变量赋值为变量时，a存在内存中的栈区(可以通过指针修改)\*/
5. **const** **int** a = b;
6. **int**\* p = (**int**\*)&a;
7. \*p = 100;   //可以通过指针修改
9. cout << a << endl;
11. **return** 0;
12. }

另外需要注意，cosnt修饰的自定义类型变量也是存在栈区。

C++中修饰全局变量示例：

1. /\*const修饰的全局变量存在常量区\*/
2. **const** **int** b = 1;
3. **int** main()
4. {
5. //b = 10;报错，const限制了不能通过变量名修改
6. **int**\* p = (**int**\*)&b;//C++类型转换比较严格，b类型为const int\*，所以需要强转
7. \*p = 100;  //报错，修改常量区的内容会出错,在Linux系统会出现段错误。
8. cout << b << endl;
10. **return** 0;
11. }

C++中const全局变量的跨文件访问示例：

1. //模块文件.cpp
2. /\*C++中const修饰的全局变量默认为内部链接属性，只能在当前文件使用，不能被其他文件链接；
3. 所以若外部文件访问，可以将其改为外部链接属性(在前面加上extern)\*/
4. //const int num = 1;
5. **extern** **const** **int** num = 1;
6. //主文件.cpp
7. **int** main()
8. {
9. **extern** **const** **int** num;
10. print("num = %d",  num);
11. }

**3.4 使用#define定义常量(不建议使用)**

**3.5. 使用constexpr定义常量(C++11)**

**3.6 字面常量：**

可以是任意类型，比如bool、整型、字符串等等。

**3.7 使用constexpr定义常量表达式(C++11引入)**

常量表达式(函数表达式)必须包含简单的实现，并返回简单类型，如int、double等。

1. constexpr **int** square(**int** x)
2. {
3. **return** x \* x；
4. }
5. float a[square(9)];

同等条件下在编译阶段进行解析，最终程序的执行速度会比将这些计算放在函数中时更快。

注意和常规函数的区别，函数是在运行阶段进行解析；而常量表达式是在编译阶段进行解析，只要编译器能够进行计算解析都会进行计算来优化，否则就忽略这个constexpr进行将常量表达式当作常规函数来处理。

**4 数据类型介绍**

**4.1 内置类型**

bool/char/float/dobule/int/short int/long/unsigned int/......

C++11: char16\_t/char32\_t/long long

**4.2 自定义类型**

数组/enum/union/struct/class......

**4.3 类型操作**

可以使用auto自动推断类型(C++11)

使用typedef替换变量的类型

**5 运算符介绍**

是C++提供的工具，让你能够使对数据进行处理。

**5.1 算术运算符**

加(+)、减( -)、乘( \*)、除( / )、求模(%)、自增(++)、自减(--)

其中前置++/--：操作数先加/减1，然后赋给变量。后置++/--，与前置相反

1. **int** a = 5, b = 2;
2. **int** c = a % b;

**5.2 关系运算符**

不等(!=)、相等(==)、小于(<)、大于(>)、小于等于(<=)、大于等于(>=)

1. **int** c = 0, d = 0;
2. **bool** my\_bool = c != d;

**5.3 逻辑运算符**

与(&&)：两边都为true则总体才为true，否则为false

或(||)：两边都为false时总体才为false，否则为ture

非(!): 取反

1. **int** c = 0, d = 0, e = 1;
2. //对于单个变量名的bool判断，若其值为0，则为false，非零则为true
3. **bool** my\_bool = c > 0 || d < 0 || e;
4. **bool** my\_bool = !e;

**5.4 按位运算符**

与(&)：两边都为1，则为1，否则为0

或(|)：两边都为0，则为0，否则为1

取反(~)：0变1，1变0

异或(^)：两边相等则为0，不等则为1

左移(<<)、右移(>>)：对一个数乘以(左移)或者除以(右移)2的n次方

1. 5 & 15 = 0101（5）
2. 5 | 15 = 1111
3. ~5 = 1010
4. 5 ^ 15 = 1010
5. 16 \* 2的2次方 = 10000 << 2 = 1000000 = 64

**5.5 赋值运算符**

直接赋值(=)、加法赋值(+=)、减法赋值(-=)、乘法赋值(\*=)、除法赋值(/=)、求模赋值(%=)、按位左移赋值(<<=)、按位右移赋值(>>=)、按位与赋值(&=)、按位或赋值(|=)、按位异或赋值(^=)

1. a += b; //a = a + b;
2. a \*= b; //a = a \* b;

**5.6 其他地一些运算符**

作用域解析运算符(::)、sizeof运算符、条件运算符(? :)、逗号运算符(,)、成员运算符(.和->)、取地址运算符(&)、解引用运算符(\*)、内存分配和释放运算符(new/delete/new[]/delete[])、类型转换运算符

**5 enum类型**

**5.1 自定义枚举类型**

枚举是由一组称为enumerator(枚举量)的常量组成。编译器会将枚举量转换为整数，每个枚举量都比前一个大1。你可以指的起始值，如果没有指定，编译器认为起始值为0。当然还可以显式地给每个枚举量指定值。

1. **enum** four\_directions {
2. East,
3. West,
4. South,
5. North
6. };

**5.2 定义变量**

可以使用枚举来定义变量，这样变量只能取指定的值。

1. for\_directions my\_diret = West;
2. cout << my\_direct << endl;

**5.3 限定作用域的定义和使用方式(C++11)**

定义：enum class/struct 自定义类型名 {枚举量}；

使用：自定义类型名 变量名 = 自定义类型名::枚举量成员；

**6 数组**

**6.1 一维数组的定义、初始化和索引操作**

1. **int** my\_array[5] = {1，2，3，4，5};  //初始化5个元素
2. **int** my\_array[5] = {};  //都初始化为0
3. **int** my\_array[5] = {1, 2};   //只初始化前2个元素，后3个为0
4. **int** my\_array[] = {1, 2, 3}; //创建一个数组，包含3个int类型的元素
5. **int** my\_array[ARRAY\_SIZE] = {1, 2, 3, 4, 5}； //constexpr int ARRAY\_SIZE = 5；
6. 访问数组时索引从0开始，最后一个索引为ARRAY\_SIZE-1。

b.使用索引访问(读写)数组时，编译器并不会检查索引是否在数组的范围之内，你可以在最大索引ARRAY\_SIZE-1的数组中取回索引超过ARRAY\_SIZE-1的元素；但是这样超越边界的话结果将是无法预料的，很多情况下会导致程序崩溃，所以程序员需要确保访问数组时不超越边界。

**6.2 多维数组的定义、初始化和索引操作**

多维数组可以看成是”数组”的数组，C++可以模拟多维数组，由于内存是一维的，所以编译器将多维数组映射到内存，内存只沿一个方向延伸。

1. **int** my\_arrays[2][3] = {{0, 1, 2}, {3, 4, 5}};   //2行3列，推荐使用这种，更直观
2. **int** my\_arrays[2][3] = {0, 1, 2, 3, 4, 5}

每一维的索引操作都和一样数组一样。

**6.3 动态数组：std::vector**

以上讲到的数组是静态数组，它们有一个特点：长度(元素个数)在编译阶段就已经确定，也就是说包含的元素以及所占用的内存量都是固定的，比如对于一维数组，其内存为sizeof(元素类型) \* ARRAY\_SIZE。在这种情况下，如果预留的内存有部分没使用，就有点浪费了。

针对这种情况，有没有办法得到在编译阶段长度不固定的数组呢？这就有了动态数组。顾名思义就是可以在运行时候可以根据需要动态调节数组的长度。

动态数组可以让程序员无需在编译阶段考虑其最大长度，使用动态数组可更好地管理内存，以避免分配更多内存，而又不使用它们。

**7 字符串**

**7.1 C语言风格字符串**

用双引号括起来，它是由一个个的字符(单引号括起来)组成。这种C风格的字符串在内存中存储的时候，编译器总是在会最后一个字符后面加上空字符’\0’，这种C风格字符串与下面使用char数组的方式等价：

1. /\* 字符数组的最后一个字符为空字符'\0' \*/
2. **char** my\_carray[] = {'H', 'e', 'l', 'l', 'o', ' ', 'C++', '\0'};  //"Hello C++"

‘\0’这个字符只是用来标识字符串结束，比如my\_carray[5] = ‘\0’, 然后再次输出发现只会输出Hello，也就是说cout看到’\0’后就停止输出。

字符串长度(使用strlen函数计算)是不包含末尾这个空字符串。这里需要与用数组存储字符串的区别，使用数组存储时，数组的长度应该比字符串长度大1。

**7.2 C++风格字符串：std::string**

不同于字符数组(长度固定)，string是动态的，需要存储更多数据时其容量将增大。

字符串的拼接：“+”

字符串的复制：“="

字符串的长度：length()

**8 条件结构**

**8.1 if-else条件结构**

1. /\*注意点：
2. 1） 若条件表达式为false时不执行任何操作，可以省略else部分。
3. 2）大括号内的多条语句被称为语句块(复合语句)，当只有单条语句时可以不使用大括号。
4. \*/
5. **if** (条件表达式)
6. {
7. //当条件表达式的值为true(表达式的结果为不为0，则可视为true)则执行此部分;\*/
8. }
9. **else**
10. {
11. //当条件表达式的值为false(表达式的结果为0)则执行此部分;
12. }

也可以组合使用多个if...else结构

1. **int** main()
2. {
3. **enum** four\_directions {
4. East,
5. West,
6. South,
7. North
8. };
10. **int** direct\_input = East;
11. cin >> direct\_input;
13. **if** (direct\_input == East)
14. cout << "East is called" << endl;
15. **else** **if** (direct\_input == West)
16. cout << "West is called" << endl;
17. **else** **if** (direct\_input == South)
18. cout << "South is called" << endl;
19. **else** **if** (direct\_input == North  )
20. cout << "North is called" << endl;
21. **else**
22. cout << "Wrong Input" << endl;
24. **return** 0;
25. }

**8.2 switch-case条件结构**

计算表达式的值，并将其与每个case标签(常量, 表)进行比较。如果计算出

的表达式的值与case标签相等，就执行标签后面的代码，break将导致程序

退出当前代码块。

1. /\*注意点：
2. 1) 如果省略break，无论标签是否满足都将继续执行直到遇到break(若没遇到则直
3. 到switch-case结束)。
4. 2) default也是可选的，它用于执行表达式不与任何标签匹配时的操作。
5. \*/
6. **switch**(表达式)
7. {
8. **case** 标签1：
9. 执行操作；
10. **break**;
11. **case** 标签2：
12. 执行操作；
13. **break**;
14. ......
15. **default**:
16. 执行操作；
17. **break**;
18. }
19. **switch** (direct\_input)
20. {
21. **case** East:
22. cout << "East is called" << endl;
23. **break**;
24. **case** West:
25. cout << "West is called" << endl;
26. **break**;
27. **case** South:
28. cout << "South is called" << endl;
29. **break**;
30. **case** North:
31. cout << "North is called" << endl;
32. **break**;
33. **default**:
34. cout << "Wrong Input" << endl;
35. **break**;
36. }

**9 循环结构**

**9.1 while循环(循环也叫迭代)结构**

1. /\*1)只要条件表达式为true就一直执行语句块，因此确保条件表达式在特定条件
2. 下将为false，否则while将永不停止
3. 2) 若在while(条件表达式)后面加分号也是合法的，不过这将导致循环执行一个空语句(因为只有分号的话相当于是个空语句)。此时后面的语句块就与while不相干了\*/
4. **while** (条件表达式)
5. {
6. //执行操作；
7. }

示例如下：

1. **int** main()
2. {
3. **int** my\_val = 3;
4. **while** (my\_val)
5. {
6. cout << "continue......" << endl;
7. --my\_val;
8. }
9. **return** 0;
10. }

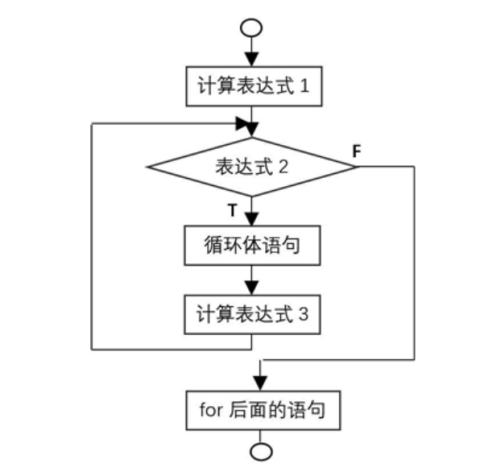
**9.2 do...while循环(迭代)结构**

1. /\* 1）循环会至少执行一次
2. 2）注意while(条件表达式)后面有一个分号，这不同于前面的while循环。\*/
3. **do**
4. {
5. //执行操作；
6. } **while**(条件表达式)；

示例如下：

1. **int** main()
2. {
3. **int** my\_val = 3;
4. **do**
5. {
6. cout << "continue......" << endl;
7. --my\_val;
8. } **while** (my\_val);
9. **return** 0;
10. }

**9.3 for循环(迭代)结构**



1. **for** (表达式1; 表达式2; 表达式3) {  //可以初始化多个变量 ,for (;;) {}是死循环
2. // 执行循环体
3. }

**9.4 range-based for循环(迭代)结构，C++11引入**

1. /\*declaration定义一个变量，sequence表示一个序列\*/
2. **for**(declaration : sequence)
3. {
4. //执行循环体
5. }
6. vector<**int**> vec{1,2,3}; //C++11 only，大括号初始化
7. **for** (**int** i : vec )
8. cout << i;

**9.5 循环结构中的break和continue**

break可以终止当前所在的循环过程。

continue可以终止本次循环并进入下一次循环。

示例一：

1. **int** main()
2. {
3. **for** (**int** i : { 2, 3, 5, 7, 9 })
4. {
5. **if** (i == 7)
6. //continue;
7. **break**;
8. cout << i << endl;
9. }
10. **return** 0;
11. }

示例二：

1. **int** main()
2. {
3. **int** my\_val = 0;
4. **while** (my\_val < 3)
5. {
6. ++my\_val;
7. **if** (my\_val == 1)
8. //continue;
9. **break**;
10. cout << my\_val << endl;
11. }
12. **return** 0;
13. }

**10 函数**

**10.1 auto作为返回类型**

从C++14起，可以使用auto让编译器根据您返回的值来推断函数的返回类型，而不直接指定类型。

**10.2 void作为返回类型和形参**

void为“无类型”，不可以用void来声明变量。如：void my\_val = 10；如果出现这样语句[编译器](https://so.csdn.net/so/search?q=%E7%BC%96%E8%AF%91%E5%99%A8&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/hankern/article/details/_blank)会报错。

void一般用作函数的返回类型限定和对函数参数的限定：当将一个函数的返回类型限定为void后，该函数就不能再有返回值；当将一个函数的形参限定为void后，就不能再有参数(此时void可省略)。

1. **void** sys\_hello()
2. {
3. cout << "Hello C++" << endl;
4. return; //仅仅只有一个return表示结束方法的执行，返回到被调用处
5. }

**10.3 默认值参数**

可以在函数声明中给多个参数指定默认值，但是这些参数必须位于形参列表的末尾，对于调用者来说这个默认参数就是可选的。

**10.4 将数组作为实参传递给函数示例**

1. **void** show\_array(**int** the\_array[], **int** length);
3. **int** main()
4. {
5. **int** my\_array[4] = { 2, 3, 4, 5 };
6. show\_array(my\_array, 4);
7. **return** 0;
8. }
10. **void** show\_array(**int** the\_array[], **int** length)
11. {
12. **for** (**int** i = 0; i < length; ++i)
13. cout << the\_array[i] << " ";
14. cout << endl;
15. }

**10.5 inline函数**

常规函数调用时，CPU需要跳转到被调用函数处处理，处理完成后再返回到最初离开的地方(幕后发生了不少事)，CPU执行这个调用需要多出跳转时间开销。所以如果被调函数非常短的话并且使用很频繁的情况下就可以将其前面加一个inline(编译器通常将关键字inline视为请求，将函数的内容直接放到调用它的地方），这样的话CPU就不用再去跳转了，以提高代码的指行速度。

**10.6 函数重载**

函数名相同，但形参列表不同的函数称为函数重载，如果需要使用不同的参数调用具有特定名称和返回类型的函数，函数重载将很有用。

注意函数重载的依据只有形参列表不同。函数的返回类型不同不可以作为函数重载的依据。

**10.7 函数模板**

**11 指针**

**11.1 指针的定义和解引用**

1. /\*1）定义一个指向int类型的指针my\_ptr，也就是说my\_ptr中存的地址对 应的内存单元存了
2. 一个整数。
3. 2）与大多数变量一样，除非对指针进行初始化，否则包含的值将是随机的(对于指针来说这非常危险，因为指针包含的值被视为地址，程序中对未初始化的指针进行误操作可能导致程序访问非法内存单元，导致崩溃，因此定义时初始化。如果不能初始化为一个有效地址，可将其初始化为NULL)。\*/
4. **int** my\_var = 10;
5. **int**\* pmy\_var  = &my\_var；

有了包含合法地址的指针后，便可以通过解引用运算符（\*）来访问这个地址所在的内存。

1. /\*解引用运算符作用域指针pmy\_var，以打印pmy\_var所保存内存地址所对应内存的值\*/
2. \*pmy\_var

指针也是变量，与所有的变量一样，指针也占用内存空间。与普通变量(比如int变量存的值一个整数)不同的是指针里面存的是内存地址。因此也可以说指针是一种指向内存单元的特殊变量。

1. /\*因为指针存的是地址，所以无论指针是那种类型，所以在特定系统中其值是固定的，不同的系统不同\*/
2. cout << **sizeof**(**int**\*) << endl;   //sizeof(my\_ptr)

**11.2 void指针**

void\*为无类型指针，void\*可以用来定义指针。如void\* a;

void\*指针可以接收任意类型的赋值。

1. **void** \*a = NULL；
2. **int** \* b = NULL；
3. a  =  b；//a是void \* 型指针，任何类型的指针都可以直接赋值给它，无需进行强制类型转换

但void\*赋值给其他类型时需要进行强制类型转换。

1. **int** \* a = NULL ；
2. **void** \* b ；
3. a  =  （**int** \*）b；

但是有意思的是：void\* 在转换为其他数据类型时，赋值给void\* 的类型 和目标类型必须保持一致。简单点来说：void\* 类型接受了int \* 的赋值后 这个void \* 不能转化为其他类型，必须转换为int \*类型；

1. **int** main ()
2. {
3. **int** a= 10;
4. **void** \*b = &a;
5. printf("int a = %d\n",a);
6. printf("void (int \*)b =%d \n",\*(**int** \*)b);
7. printf("void (double \*)b =%d \n",\*(**double**\*)b);  //编译器并不会报错但是其结果却有点出人意料
9. **return** 0;
10. }

**11.3 多级指针:**

int my\_val = 10;

int\* pmy\_val = &my\_val;

int\*\* ppmy\_val = &pmy\_val;

int\*\*\* pppmy\_val = &ppmy\_val;

...

**11.4 动态分配和释放内存**

除了编译器去自动分配内存(静态内存)外，程序员自己也是可以手动来分配(new)和释放(delete)内存(动态内存)的。这里需要注意不再使用分配的内存后需要释放，如果不释放将导致系统可用内存越来越少，这被称为内存泄露。

运算符new和delete分配和释放自由存储区(堆区)中的内存。堆区是一种内存抽象，表现为一个内存池，应用程序可以分配(预留)和释放其中的内存。

使用delete释放堆内存之后，任何有效指针都将无效。为避免这种问题，很多程序员在初始化指针或释放指针后将其设置为NULL，并在使用运算符\*对指针解引用前检查它是否有效(将其与NULL比较)。

1. /\*使用new时，需要指定为哪种类型分配内存,如果成功则将返回该内存的
2. 地址，使用new分配的内存，都需要使用对应的delete释放\*/
3. **int**\* pmy\_val = **new** **int**;
4. **delete** pmy\_val;
6. /\*需要为多个元素分配内存时，还可以指定元素的个数\*/
7. **int**\* pmy\_vals = **new** **int**[10];
8. **delete**[] pmy\_vals;

非常请求分配的内存量特大，或系统处于临界状态，可提供的内存很少，否则一般new都可以成功。对于分配不成功时，C++提供了两种确认指针有效的方法。

默认方法是使用异常，即如果内存内存分配失败将发生std::bad\_alloc异常，这导致应用程序中断，除非你提供了异常处理程序，否则应用程序将崩溃：

1. /\*只执行如下语句，会导致应用程序中断、崩溃\*/
2. **int**\* p\_myarray = **new** **int**[0x1fffffffffff];

可以提供以下处理程序，让程序不崩溃，正常退出：

1. **int** main()
2. {
3. **try** {
4. //request a lot of memory!
5. **int**\* p\_myarray = **new** **int**[0x1fffffffffff];
6. //use the allocated memory
8. **delete**[] p\_myarray;
9. }
10. /\*如果没有编写异常处理程序(catch块)，程序将以非常讨厌的方式结束(崩溃)。由此可见当
11. 内存分配导致程序无法正常执行时，可以使用try-catch异常处理结构让程序能够向用户指出
12. 这一点，再正常退出\*/
13. **catch** (bad\_alloc)
14. {
15. cout << "Memory allocation failed." << endl;
16. }
17. **return** 0;
18. }

如果不想依赖于异常，可以使用另一种方法：new变种new(nothrow)，这个变种再内存分配失败时不引发异常，而是返回NULL。这样的话就能够可以再使用指针前检查其有效性

1. **int** main()
2. {
3. **int**\* p\_myarray = **new**(**nothrow**) **int**[0x1fffffffffff];
5. **if** (p\_myarray) //check p\_myarray != NULL
6. {   //use the allocated memory
8. **delete**[] p\_myarray;
9. }
10. **else**
11. cout << "Memory allocation failed." << endl;
13. **return** 0;
14. }

**11.5 指针的自增和自减**

定义一个指针：varType\* my\_ptr = Address；//Address为my\_ptr指向的内存的地址

那么对指针my\_ptr执行一次++/--运算后的值为：Address +/- sizeof(varType)

**11.6 数组和指针**

数组变量(本质是指针常量)就是指针(指向其第一个元素)，因此:

可以将指针解引用运算符(\*)用于数组，同样，可将数组运算符([])用于指针。

1. **int** main()
2. {
3. **const** **int** ARRAY\_SIZE = 5;
4. **int** my\_array[ARRAY\_SIZE] = {24, -1, 365, -999, 2011};
5. /\*这里需要注意，下面若反过来就不行了。因为数组名本质上是一个指针常
6. 量，因此不可修改。\*/
7. **int**\* pmy\_array = my\_array;
9. **for** (**int** i = 0; i < ARRAY\_SIZE; ++i)
10. cout << i << "=" << \*(my\_array + i) << endl;
12. cout << "......................................" << endl;
14. **for** (**int** i = 0; i < ARRAY\_SIZE; ++i)
15. cout << i << "=" << pmy\_array[i] << endl;
17. **return** 0;
18. }

**11.7 将关键字const用于指针和指针间赋值**

通过将变量声明为const，可确保变量的取值在整个生命周期内都固定为初始值，这种变量的值不能修改(常量)。指针也是变量，因此可将const用于指针。

然而指针是特殊变量，既可以用于存储内存地址，还可用于修改修改其指向的内存中的数据。因此const指针有如下三种：

1)指针中所存储的内存地址不可修改；但内存地址中的内容(指针指向的内存中的数据)可修改。即指针本身是常量。

1. **int** my\_var = 10;
2. **int**\* **const** pmy\_var = &my\_var;
3. \*pmy\_var = 11;  //OK
4. **int** my\_var1 = 20;
5. pmy\_var = &my\_var1; //Error,Cannot change address

2)指针中所存储的内存地址可以修改；但指针指向的内存中的数据不可修改。即指向常量的指针。

1. **int** my\_var = 10;
2. **const** **int**\* pmy\_var = &my\_var;
3. \*pmy\_var = 11;  //Error, Cannot change data beging pointed to
4. **int** my\_var1 = 20;
5. pmy\_var = &my\_var1; //OK
6. /\*int \* 表示，指向非常量int类型的指针，指针这块内存的内容可修改，指针
7. 本身可修改。而const int\* 已经限制此地址内容，不可修改。这时却让 int \* 指针
8. 指向这块地址，而使用 int \* 指针，表示此地址内容可修改。那么到底可不可以改？
9. 所以编译器会报错\*/
10. **int**\* newPoint = pmy\_var; //Error

3)指针中所存储的内存地址以及它指向的内存中的数据都为常量，都不可修改。即指向const数据的const指针，这种组合最严格，一般将指针传递给函数时，函数形参应该声明为这种最严格的const指针，以确保函数内部不会修改指针及其指向的值。

1. **int** my\_var = 10;
2. **const** **int**\* **const** pmy\_var = &my\_var;
3. \*pmy\_var = 11;  //Error, Cannot change data beging pointed to
4. **int** my\_var1 = 20;
5. pmy\_var = &my\_var1; //Error, Cannot change address

另外注意一点在指针间赋值时必须满足：

a.两个操作数都是指向有限定符或无限定符的相容类型的指针；

b.左边指针所指向的类型必须具有右边指针所指向类型的全部限定符。

**11.8 按指针传递函数参数示例**

1. /\*将指针作为函数参数时，确保函数只修改你希望它修改的参数很重要。例如下面如果函数根据
2. 以指针方式传入的半径计算圆的面积，就不应该修改半径。可以通过const来控制。
3. ptr\_area用于存储输出的参数，因此不能修改修改指针的值(地址)，但可以修改它指向的数据\*/
4. **void** get\_area(**const** **double**\* **const** ptr\_pi,
5. **const** **double**\* **const** ptr\_radius,
6. **double**\* **const** ptr\_area)
7. {
8. **if** (ptr\_pi && ptr\_radius && ptr\_area)
9. \*ptr\_area = (\*ptr\_pi) \* (\*ptr\_radius) \* (\*ptr\_radius);
10. }
11. **int** main()
12. {
13. **const** **double** pi = 3.1416;
15. **double** radius = 10, area;
17. get\_area(&pi, &radius, &area);
19. cout << "Area is: " << area << endl;
20. }

**12 引用**

**12.1 定义**

引用是变量的别名，所以定义引用时，需要将其初始化为一个变量，因为引用只是另一种访问相应变量存储的数据的方式。可以使用引用运算符&来定义一个引用。

1. /\*无论将引用初始化为变量还是其他引用，它都指向相应变量所在的内存单元。
2. 因此引用是真正的别名，即相应变量的另一个名字\*/
3. **int** my\_val = 10;
4. **int**& rmy\_val = my\_val;  //int\* const rmy\_val = &my\_val
5. **int**& rmy\_val1 = rmy\_val;

**12.2 将const用于引用**

const修饰的是引用，所以不能通过引用取修改引用的这块空间的内容。

如果想禁止通过引用修改它指向的变量的值，为此可在声明引用时使用关键字const。

1. **int** my\_val = 10;
2. **const** **int**& rmy\_val = my\_val;
3. rmy\_val = 11; //not allowed, rmy\_val cannot change value
4. /\*const类型的引用(rmy\_val)只能赋值给const类型的引用(rmy\_val1)\*/
5. **int**& rmy\_val1 = rmy\_val;  //not allowed, rmy\_val1 is not const
6. **const** **int**& rmy\_val1 = rmy\_val;  //Ok
7. /\*非const类型的引用既可以赋值给const类型的引用，也可以赋值给非const类型的引用\*/

**12.3 按引用传递参数示例**

引用让你能够访问相应变量变量所在的内存单元，这使得编写函数时引用很有用。

1. /\*下述代码将导致实参argument的值直接复制传递给形参parameter，再被DoSomething()使用。
2. 如果argument占用的内存比较大，那这个复制过程的开销就比较大。同样当DoSomething()返回
3. 时，这个值被复制给Result。如果能避免这些步骤，让函数直接使用调用者栈中的数据就好了。为此 可使用引用。\*/
4. ReturnType DoSomething(Type parameter);
5. ReturnType Result = Dosomething(argument);
6. /\*可避免复制步骤的被调函数的代码类似下面这样。\*/
7. ReturnType DoSomething(Type& parameter);//parameter不再是argument的拷贝而是别名
8. ReturnType& DoSomething(Type parameter);

示例：

1. **void** get\_square(**int**& num)//通过使用引用可确对main中的number所在内存单元进行操作
2. {
3. num \*= num;
4. }
5. **int** main()
6. {
7. **int** number = 10;
8. get\_square(number);
9. cout << "Square is: " << number << endl;
10. }

引用的有点之一是可避免将实参直接复制给形参，提高性能。然而，让被调用的函数直接使用调用函数栈时，确保被调用函数不能修改调用函数中的变量很重要。为此，可将引用声明为const的。

1. **void** get\_square(**const** **int**& num, **int**& result)  //禁止修改num的值
2. {
3. result = num \* num;
4. }
5. **int** main()
6. {
7. **int** number = 10, my\_square;
8. get\_square(number, my\_square);
10. cout << "Square is: " << my\_square << endl;
11. }

**12.4 指针的引用示例**

1. **void** get\_memory(**int** \*\*q)
2. {
3. \*q = (**int** \*)malloc(**sizeof**(**int**));
4. }
5. **void** get\_memory(**int**\* &q)
6. {
7. q = (**int**\*)malloc(**sizeof**(**int**));
8. }
9. **int** main()
10. {
11. **int** \*p = NULL;
12. get\_memory(&p);
13. get\_memory(p);
14. }

**12.5 右值引用(C++11引入)**

1. /\*左值：在内存中是有明确的内存地址，其可以被修改，比如变量、数组元素等。
2. 右值：在内存中没有明确的内存地址。其通常不可修改，像字面值常量、表达式的结果等都是右值。\*/
3. **int** array[3] = {1, 2, 3};
4. array[1] = 10;  //数组元素array[1]是左值。
5. **int** a = 10;  //a为左值，
6. **int** b = a;  //b为左值，a为左值，
7. **int** sum = a + b;   //sum为左值，a + b为右值。右值只能出现在赋值语句的右边；

引用和指针类似，无论是左值引用(Type&)还是右值引用(Type&&, C++11引入的)，要有引用就必须确保被引用的值是有内存地址的。

左值引用用来引用左值(有一个比较特殊：const左值引用可以引用右值):

1. **int** a = 10;
2. **int**& x = a;
3. **int**& x = 10;  //错误！10是一个右值，没有地址，所以用来不能初始化一个左值引用

右值引用只能用来引用右值:

1. /\*这里10没有地址，为什么却可以取其引用呢？这是因为编译器在取右值引用时，会先用一个"临时变量"存储
2. 右值10，然后再用这个”临时变量"对这个右值引用进行初始化。而临时变量是有地址的，可以对其取引用“\*/
3. **int**&& i = 10;  //注意虽然i引用的是一个右值，但是它本身仍是个左值。
5. /\*虽然右值引用不能绑定左值对象，但可以利用标准库提供的move函数将一个左值转换为右值，例如将左值r1转换为右值：\*/
6. **int** &&r2 = r1; //错误
7. **int** &&r2 = std::move(r1);

**12.6 通用引用和完美转发**

引用折叠规则：& & ，& && 和&& &都会被折叠为&。&& &&被折叠为&&

通用引用(有的地方也叫万能引用)：通常情况下右值引用&&只能与右值绑定，不能绑定左值，但在以下二种需要推断类型的场合时，它既可以与左值绑定也可以与右值绑定，此时它变成一种通用引用(universal reference)。

1. a. 当右值引用&&与atuo结合在一起时：
2. **int** i = 10；
3. auto r0 = 10; //对于该赋值，编译器将auto推导为int
4. auto r0 = i; //对于该赋值，编译器将auto推导为int
5. auto& r0 = i;  //对于该赋值，编译器将auto推导为int
6. auto&& r1 = i; //对于该赋值，编译器将auto推导为int&类型，这样的话r1类型为int& &&即int&
7. auto&& r2 = 10;  //对于该赋值，编译器将auto推导为int&类型，这样的话r2的类型为int&&
9. b. 当右值引用&&与模板中的类型参数T(泛型)结合在一起时:
10. **template**<**typename** T>
11. **void** forwardValue(T&& val)  //形参为左值
12. {
13. ...
14. }

完美转发：

1. /\*将输入的参数原封不动地传递到下一个函数中，这个“原封不动”指的是：
2. a.如果输入的参数是左值，那么传递给下一个函数的参数的也是左值；
3. b.如果输入的参数是右值，那么传递给下一个函数的参数的也是右值。\*/
4. std::forward<T>(参数);
5. /\*
6. 其本质是一个强制类型转换：
7. 如果传入T = int，那么param将被强转为int&&类型，最终保证了实参的右值属性。转发正确！
8. 如果传入T = int&，那么param将被强转为int& &&即int&，最终保证了实参的左值属性。转发正确！
9. \*/
10. **return** **static\_cast**<T&&>(val);

代码示例：

1. **void** process(**int**& val) {
2. cout << "process(int&)" << endl;
3. }
4. **void** process(**int**&& val) {
5. cout << "process(int&&)" << endl;
6. }
7. **template**<**typename** T>
8. **void** forwardValue(T&& val) {
9. process(forward<T>(val));
10. }
12. **int** main()
13. {
14. **int** my\_val = 42;
15. forwardValue(42);   //T -> int
16. forwardValue(my\_val);   //T -> int&
17. **return** 0;
18. }

**13 union类型**

在使用union自定义的类型中，所有成员公用同一块内存空间，union变量所占用的内存长度等于其最长的那个成员的内存长度。

每次只有一个成员处于活动状态，也就是说union限制我们每次只能使用它的一个成员。

1. **union** Test {
2. **char** my\_char;
3. **long** my\_long;
4. **int** my\_array[4];
5. }
7. Test my\_test;
8. my\_test.my\_long = 1001;    //选择第二个成员处于活动状态
9. cout << my\_test.my\_long << endl;

需要注意，在C++中，union也可以像class和struct那样在其中定义成员函数，比如构造函数等，但是工程项目上这种用法很少见，所以暂不必掌握。

**14 类和对象**

**14.1 类和对象的引入**

将数据(又叫属性)以及使用这些数据的函数(在类中，又称为方法)封装在一块，这种结构就是类，属性和方法都是类的成员。这种封装是面向对象编程的重要特征。

要使用类的功能，通常要创建其实例，即对象(使用类定义的变量，实例化一个对象指的就是用自定义的类定义一个变量)，并通过对象访问成员方法和属性。通过对象来访问类成员时可以使用运算符.和->

若将类的属性和方法设置为public则可以在类外进行访问；若设置为private则在类的外部不能直接访问，除非使用友元。

1. **class** Person
2. {
3. **private**:
4. /\*成员属性\*/
5. string name;
6. **int** age;
7. **public**:
8. /\*成员方法\*/
9. **void** set\_name(string PersonName)
10. {
11. name = PersonName;
12. }
13. **void** introduce\_self()
14. {
15. cout << "My name is " << name << endl;
16. }
17. };
19. **int** main()
20. {
21. Person firstman;
22. firstman.set\_name("Luca");
23. firstman.introduce\_self();
25. Person\* secondman = **new** Person;
26. secondman->set\_name("Lisa");
27. secondman->introduce\_self();  //等价于（\*secondman).introduce\_self()
28. **delete** secondman;
30. **return** 0;
31. }

**14.2 struct与class的区别**

在C++编译器看来，结构(struct)与类(class)极其相似，差别在于成员的访问权限未指定时，默认访问权限不同。

a.除非显式指定了，否则结构中的成员默认为public的(而类成员默认为private的)；

b.除非显式指定了，否则结构默认以public方式继承基结构(而类为private继承方式继承基类)。

**14.3 构造函数地引入和默认构造函数**

构造函数在实例化对象时被编译器自动调用，可以用来初始化对象。

1. Person(string PersonName = "Luca", **int** PersonAge = 18)  //Person()
2. {
3. name = PersonName;       //构造函数的函数体中进行初始化
4. age = PersonAge ;
5. ......
6. }
7. /\*使用初始化列表进行初始化：由冒号标识，冒号后面列出了各个变量及其初始值；初始值可以是参数或者固定值\*/
8. Person(string PersonName = "Luca", **int** PersonAge = "18") : name(PersonName) , age(PersonAge)
9. {
10. ......
11. }

实例化对象时可不提供初始值，所调用的构造函数(默认构造函数可以在实例化对象时不传参数，而其他的构造在实例化对象时必须传参数)。默认构造函数有二种：a.构造函数不带参数；b.构造函数带参数但参数都有默认值(在一个类中默认构造函数只能有一个)。例如：

1. Person() {**do** something};//默认构造函数：
2. Person(**int** PersonAge=18, string PersonName ="Luca") {**do** something};//默认构造函数
3. Person(string PersonName) {**do** something};//普通构造函数

注意：如果在定义的类中一个构造函数都没写的话，编译器会提供一个空的"隐式默认构造函数"。它不会对成员属性进行初始化。但是一旦在类中提供了任意构造函数，编译器将不提供该"隐式默认构造函数"。

1. Person() {};

另外注意可使用constexpr关键字将构造函数定义为常量表达式。在有助于提高性能的情况下，可在构造函数的声明中使用这个关键字。

**14.4 析构函数**

析构函数在对象销毁时自动被调用。

析构函数不能重载，每个类都只能有一个析构函数。如果不自己实现析构函数，编译器将自动创建一个隐式的默认析构函数并调用它:~Person() {};

需要注意的是编译器创建的这个隐式默认析构函数为空，因此若申请了动态内存，则不会自己释放。

**14.5 拷贝构造函数、浅拷贝和深拷贝**

1. /\*拷贝构造函数唯一的一个参数是本类型的一个引用变量，该参数是const类型，不可变的，使用时注意：
2. a. 编写复制构造函数时务必将接受源对象的参数声明为const，可确保复制构造函数不会修改指向的源对象。另外复制构造函数必须将指向源对象的引用作为参数，这是编译器的要求，其原因是如果按值接受源对象，复制构造函数将调用自己，导致没完没了的复制循环，直到耗尽系统的内存为止。
3. b.声明构造函数时务必考虑使用关键字explicit，以避免隐式转换。
4. c. 除非万不得已，不要类成员声明为原始指针，如果你编写类时需要包含字符串成员，用于存储姓名等。应使用std::string，而不是char\*。在没有使用原始指针的情况下，你都不需要编写复制构造函数。这是因为编译器添加的默认复制构造函数将调用成员对象(如std::string)的复制构造函数。 在设计类时，属性成员应尽可能使用智能指针、string和STL容器来减少工作量。
5. \*/
6. Test(**const** Test& copy\_src)
7. {
8. **do** something;
9. }

当用一个已初始化过了的自定义类类型对象去初始化另一个新构造的对象的时候，拷贝构造函数就会被自动调用。也就是说，当类的对象需要拷贝时，拷贝构造函数将会被调用。以下情况都会调用拷贝构造函数：

1. a.一个对象需要另一个对象进行初始化。
2. Test my\_test(100);
3. Test my\_test1(my\_test);  //或Test my\_test1 = my\_test;
4. b.当对象直接作为参数传给函数时，函数将建立对象的临时拷贝。
5. Test my\_test(100);
6. use\_test(my\_test);
8. **void** use\_test(Test src) {   //Test src = my\_test;
9. **do** something;
10. }
11. c.当函数中的局部对象作为返回值返回给函数调用者时，也将建立此局部对象的一个临时拷贝。
12. Test use\_test()
13. {
14. Test my\_test(100);
15. **return** my\_test;
16. }
17. Test my\_test1 = use\_test();   //Test my\_test1 = my\_test;

拷贝构造函数的二种类型：浅拷贝和深拷贝。浅拷贝将一个对象的成员属性复制到另一个对象中，不包括动态分配的内存(堆区分配)。深拷贝除了将一个对象的成员属性复制到另一个对象中外，还要将堆区内存一块复制。

如果在没有显式地定义一个拷贝构造函数，那么编译器将生成一个默认的拷贝构造函数，只不过这个默认拷贝构造函数是浅拷贝类型。如果成员属性中没有指向堆区内存的指针，可以不显式定义，直接使用这个默认的拷贝构造函数即可。如果成员属性中有指向堆区内存的指针，那么将导致二个对象的指针都指向同一块堆区内存，因此这个时候就必须要显式定义一个拷贝构造函数来完成深拷贝。

**14.6 移动构造函数(C++11引入)**

1. /\*源对象资源的控制权全部交给目标对\*/
2. ClassName(ClassName&& other)
3. {
4. **do** something;
5. }

需要注意若有指针成员，移动后需要将原对象的指针置为空(若不置为空的话，就会导致二个指针

指向同一块内存，析构函数中delete二次会出问题)。

强制类型转换：static\_cast<Test&&>(my\_test);//将my\_test强制转换为Test&&右值引用类型。也可以使用std::move()强制转换为右值

右值有一个特点那就是基本上只用来使用一次，用完就不想再用它了，我们有时候不想拷贝它，只想移动它里面的资源(比如对于10这种字面量还没什么问题，因为它可能是存在寄存器中的效率比较高，但是对于字符串或者自定义的对象这些需要在堆上开辟空间的，就不想再去复制，而是想将其资源直接移动到需要构造的对象里)。

**14.7 类的static成员**

类的static成员是属于类本身的，而不是属于类的任何实例对象。static成员在内存中只有一份，在所有实例对象之间共享。

静态数据成员为在一个类范围的数据共享提供了技术支持，而不必使用全局变量。静态数据成员的主要用途是定义类的各个对象所公用的数据。

成员函数也可以定义为静态的。与静态数据成员不同，静态成员函数的作用不是为了对象之间的沟通，而是为了能处理静态数据成员。静态成员函数可以访问静态数据成员， 而不能访问非静态成员。

1. **class** className {
2. **private**:
3. **int** my\_val;
4. **static** **int** my\_val1;
5. **public**：
6. **void** my\_func() {
7. //do something;
8. }
9. **static** **void** my\_func1() {
10. //do something;
11. }
12. }
13. /\*second\_obj和second\_obj分别都有自己的my\_val和my\_func()成员；
14. 而my\_val1和my\_func1()即不属于first\_obj也不属于second\_obj。\*/
15. className first\_obj, second\_obj;

static属性成员不能在类中进行初始化，需要在类的外面进行初始化，初始化时候不需要再加static。

static方法成员可以访问本类中的static属性成员，不能直接访问非static属性成员。

static成员的访问可以用“类名::属性名”，但是非static成员不可以。

虽然static成员不属于具体的类对象，但是可以通过类对象来访问static成员。

**14.8 this指针**

使用自定义的类创建对象时，比如创建n的对象，那么：static成员属性在内存中只有一份，被n个对象共享；非static成员属性在内存中共有n份，每个对象在内存中都各自有一份；成员方法在内存中只有一份，被n个对象共享。

当调用成员方法时，C++如何才能保证成员方法中对成员属性的处理是针对自己的属性成员，而不是其他

对象的属性成员呢？

当使用对象调用非static成员方法时或构造函数被自动调用时，有一个隐式的指针类型形参this接受了调用对象的地址(&object)，this指针是一个指针常量。也就是说编译器将隐式地将当前所在对象的地址传递给成员方法，使this指针得到当前对象的地址。

调用static成员方法时，编译器不会生成this指针。

**14.9 类的const成员**

const修饰的成员属性，要么定义时直接初始化，要么使用构造函数的初始化列表进行初始化(不能在构造函数的函数体中进行，因为在函数体中属于再次赋值)。否则会错过初始化时机，无法完成初始化。

const修饰的成员方法(格式为：类型说明符 函数名（参数表）const)，本质修饰的还是this指针，此时this指针指向的当前对象就不能修改了。这里需要注意const是函数类型的一个组成部分，因此在声明函数和定义函数时都要有关键字const，在调用时不必加const。

const对象只能调用const成员方法，不能调用非const成员方法。(根据指针的赋值原则，不能将指针的权限放大！)

const对象可以访问非const成员属性，但是不能修改，因为const对象内的所有成员都是不允许修改的。但是加上mutable关键字就可以修改了。

const成员方法不能调用非const成员方法，但反之可以。

无论是const成员方法还是非const成员方法，都是可以读取const/非const成员属性，都不能修改const成员属性(非const的成员方法可以修改非const的成员变量； const的成员方法对于非const成员属性加上mutable后就可以修改了)。

**14.10 constexpr和编译期常量对象**

编译期常量对象(使用constexpr来修饰对象)的任何计算都在编译阶段完成。

定义编译期常量对象有以下几点要求：a. 构造函数使用constexpr修饰，必须使用初始化列表对成员进行初始化(保证对象属性成员在编译期间确定其值)。b. 对象调用的成员函数必须使用constexpr修饰(修饰后就成了一个常量表达式，调用该函数则会在编译阶段计算出结果)，另外如果该成员函数有返回值则必须使用一个定义的编译期常量来保存其结果，而不能使用变量，否则该函数仍然会在运行期执行。

1. **class** Test {
2. **private**:
3. **int** my\_val;
4. **public**:
5. constexpr Test(**int** input\_val): my\_val(input\_val) {}
6. constexpr **int** get\_val() **const** { **return** my\_val\*my\_val; }
7. };
9. **int** main()
10. {
11. constexpr Test my\_test(10);
12. **constexpr** **int** the\_val = my\_test.get\_val();
13. cout << the\_val << endl;
15. **return** 0;
16. }

**14.11 友元**

不能从外部访问类的私有成员(包括属性和方法)，但这条规则不适用于友元(包括友元类和友元函数)。要声明友元可使用关键字friend。需要注意友元并不是类的成员。

将本类外部定义的函数指定为可信赖的朋友的话，外部函数就能访问本类的私有成员。

1. **class** Test {
2. **private**:
3. **int** my\_val;
4. **friend** **void** my\_print(**const** Test& obj);
5. **public**:
6. Test(**int** input\_val): my\_val(input\_val)
7. {
8. ++my\_val;
9. }
10. };
12. **void** my\_print(**const** Test& obj)
13. {
14. cout << obj.my\_val << endl;
15. }
17. **int** main()
18. {
19. Test my\_test(10);
20. my\_print(my\_test);
22. **return** 0;
23. }

将本类外部定义的类指定为可信赖的朋友的话，外部类中的所有方法就能访问本类的私有成员。

1. **class** Test {
2. **private**:
3. **int** my\_val;
4. **friend** **class** Utility;
5. **public**:
6. Test(**int** input\_val): my\_val(input\_val)
7. {
8. ++my\_val;
9. }
10. };
12. **class** Utility {
13. **public**:
14. **void** my\_print(**const** Test& obj)
15. {
16. cout << obj.my\_val << endl;
17. }
18. };
20. **int** main()
21. {
22. Test my\_test(10);
23. Utility my\_utility;
24. my\_utility.my\_print(my\_test);
26. **return** 0;
27. }

**14.12 运算符重载的引入**

在默认情况下，C++是不支持自定义类型对象使用运算符，为了能让自定义类型能够像内置类型一样去使用运算符，就引入了运算符重载(不允许重新定义新的运算符)。

1. /\*1)作为类成员(非static成员方法)的单目运算符没有参数，因为它们使用的唯一参数是当前类实例(\*this)，运算符的操作量就是对象本身。如果重载的是双目运算符则参数只需设置一个作为右侧操作量，而左侧操作量就是对象本身(运算符函数的第一个运算对象与隐含的this指针绑定)。
2. 2)实现为全局函数或static成员方法的话左侧操作对象就不再是类对象了。\*/
3. return\_type operator operator\_type(parameter...)
4. {
5. //...implementation
6. }

在众多运算符中其中主要有5个不能重载：成员访问(.)、成员指针访问(.\*和->\*)、域操作(::)、条件(?:)、空间计算(sizeof)。

为了类的安全，属性成员的访问权限往往设置为private，对于private成员的访问可以在友元、或类的成员函数中进行访问，所以同一个运算符的重载一般有两种形式：要么重载为类的成员函数，要么友元函数。

**14.13 输入(>>)和输出(<<)运算符重载**

用户自己定义的类型的数据，是不能直接用cin(是istream类的对象用于处理标准输入)和cout(是ostream类的对象用于处理标准输出)来输出和输入的, 这是因为在istream类和ostream类里并没有针对我们自定义的类型将>>和<<进行重载。

由于不能直接修改istream类和ostream类，所以只能在其外面再定义一个重载函数来处理我们自定义的这个类型。

这二个运算符一般重载为自定义类的友元，因为>>和<<运算符的左侧对象分别是cin和cout，而不是自定类对象本身。这样的话可以让输入输出运算符也能使用到自定义类的私有成员。

1. /\*os是cout的引用, 这里假设自定义class的类型为Test\*/
2. **friend** **void** operator<< (ostream&os, **const** Test&tt);
3. **friend** ostream& operator<< (ostream&os, **const** Test&tt); //可以连续输出
4. /\*is是cin的引用，这里假设自定义class的类型为Test\*/
5. **friend** **void** operator>> (istream&is, Test&tt);
6. **friend** istream& operator>> (istream&is, Test&tt);

如果重载为类的成员函数的话就变成如下这种形式了：

1. **class** Test {
2. ....
3. **public**:
4. ostream& operator<< (ostream&os)
5. {
6. //do something;
7. }
8. };
10. Test my\_test;
11. my\_test.operator<<(cout) //显式调用
12. my\_test << cout   //隐式调用，左侧对象是my\_test, 右侧对象为cout，这显然是错的。

代码示例：

1. **class** Test {
2. **private**:
3. **int** my\_val;
4. **friend** istream& operator>>(istream& is, Test& tt);
5. **friend** ostream& operator<<(ostream& os, **const** Test& tt);
6. **public**:
7. Test(**int** input\_val) :my\_val(input\_val) {}
8. };
9. istream& operator>>(istream& is, Test& tt)
10. {
11. is >> tt.my\_val;
12. **return** is;
13. }
14. ostream& operator<< (ostream& os, **const** Test& tt)
15. {
16. os << tt.my\_val;
17. **return** os;
18. }
20. **int** main()
21. {
22. Test my\_test(10);
23. cout << my\_test << my\_test;
24. **return** 0;
25. }

**14.14 类型转换运算符重载**

1. /\*将自定义类型强制转换为某一数据类型T，比如说const char\*，float、int等\*/
2. operator T **const**()
3. {
4. //do something;
5. **return** result;
6. }

**14.15 拷贝赋值运算符重载**

用来将一个类实例对象的内容赋给另一个类实例，如果不显式定义赋值运算符函数，那么编译器将会生成一个默认的拷贝赋值运算符函数。

和默认的拷贝构造函数一样，默认的拷贝赋值运算符函数仍然是浅拷贝，所以如果类有原始指针成员的话，就必须要自定义一个拷贝赋值运算符函数，来保证完成深拷贝。

赋值运算符重载只能是类的成员函数,不允许类外实现重载为友元函数。因为若重载为友元的话，就和编译器在类中生成的默认赋值运算符函数冲突了(另外还需要注意，->, []、()这三个运算符也不能重载为友元，如果在类中不显示定义，编译器也会添加一个默

认的运算符重载成员函数)。

1. classType& operator= (**const** classType& other)
2. {
3. **if**(**this** != &other)  //protection against copy into self
4. {
5. //do something;
6. }
7. **return** \***this**;
8. }
10. classType obj1(obj1的初始化值);
11. classType obj2(obj2的初始化值);
12. obj2 = obj1 = obj0;   //拷贝赋值, 注意若classType obj2 = obj1为拷贝构造

**14.15 移动赋值运算符重载(C++11引入)**

1. classType& operator= (classType&& other)
2. {
3. **if**(**this** != &other)  //protection against move into self
4. {
5. //do something;
6. }
7. **return** \***this**;
8. }

和移动构造一样，并没有复制对象，将源对象(other)中的资源控制权移交给当前目标对象(\*this)。

和移动构造一样，当源对象资源给目标对象后不准备再用了，这个时候就没必要进行深拷贝了，直接就可以使用移动赋值运算符函数来进行优化。

**14.16 函数运算符重载**

重载函数调用运算符()的类，其实例化的对象称为函数对象(functor)。

当函数对象使用重载的这个()时，行为非常类似于函数调用，也叫仿函数(主要用作STL算法的参数)。

**14.17 类成员的protected访问限定**

类中的protected也是一个访问限定符。

当不考虑继承时，它和private访问限定符一样在类外的话只允许通过友元访问。

考虑到继承时，它除了可以让类外的友元访问外，还可以允许在其派生类中被直接访问。

**15 类的继承**

**15.1 继承中的访问权限**

继承是指在已有类的基础上创建(派生)新类，这个已有类称为基类(或父类)，新建类称为派生类(或子类)。继承分为单继承(派生类继承单个基类)和多继承(派生类继承多个基类)。

1. /\*1)其中继承方式分为public、protected和private，如果不指定默认为private（注意struct的话默认为public继承）。
2. 2)继承之后基类成员在派生类中的访问属性：public > protected > private > 不可直接访问(注意也被继承了，只是不可见而已，在派生类中可通过基类的public或protect成员函数访问)。
3. 3）需要注意的是如果派生类实现了和基类同名的方法，就会隐藏基类的这个方法。此时直接调用该方法，调用的是在派生类中实现的这个；如果要调用基类被隐藏的那个就需要使用“基类名::方法”。\*/
4. **class** 派生类 ：继承方式 基类
5. {
6. //派生类成员声明
7. }

private继承方式：基类中的所有public成员在派生类中均为private属性；所有protected成员在派生类中均为private属性；所有private成员在派生类中均不可直接访问。

protected继承方式：基类中的所有public成员在派生类中为protected属性；所有protected成员在派生类中为protected属性；所有private成员在派生类中仍然不可直接访问。

public继承方式：基类中所有public成员在派生类中为public属性；所有protected成员在派生类中为protected属性；所有private成员在派生类中不可直接访问(注意也被继承了，只是不可见而已，只能通过基类的public或protect成员函数访问)。

**15.2 继承中的构造函数和析构函数**

基类的构造函数和析构函数是不能被继承的，因此派生类必须使用基类的构造/析构函数来初始化/销毁从基类继承的成员。

1. /\*如果派生类构造函数的初始化列表没有显式调用其直接基类的构造函数，那么编译器将调用其直接基类的默认构造函数\*/
2. 派生类名(全参数列表):基类名(基类参数),派生类子类对象名1(子类对象1参数)，派生类子类对象名2(子类对象2参数)...
3. {
4. //派生类构造函数体
5. }

派生类构造函数执行顺序为：

a.基类的构造函数。

b.派生类中子类成员的构造函数(如果有的话)，有多个时按照声明的顺序。

c.派生类的构造函数。

派生类析构函数执行顺序：

a.派生类的析构函数。

b.派生类中子类成员的析构函数(如果有的话)，有多个时按照声明的顺序。

c.基类的析构函数。

**15.3 多继承**

1. /\*继承方式：public、protected、private之一，如果缺省，则是private\*/
2. **class** 派生类名:继承方式1 基类名1,继承方式2,基类名2...
3. {
4. //派生类成员声明
5. }
7. /\*在多继承的情况下，派生类的构造函数格式如下\*/
8. 派生类名(全参数列表): 基类名1(基类1参数),基类名2(基类2参数)...派生类子类对象名1(子类对象1参数)，
9. 派生类子类对象名2(子类对象2参数)...
10. {
11. //派生类构造函数体
12. }

**15.4 虚继承**

由于C++支持多继承，因此当Final\_Derived继承多个从同一个类Base派生而来的基类(Derived1, Derived2 Derived3 ...)时，如果这些基类没有采用虚继承将导致二义性。这种二义性被称为菱形问题(Final\_Derived对象中存在Base成员的多个副本,编译器不知道要操作哪个)。

解决方法便是采用虚继承，即从Base派生出Derived1, Derived2 Derived3 ...时使用关键字virtual

1. /\*1) virtual也可以放在继承方式的后面。
2. 2) 这个基类可称为"虚基类"。
3. \*/
4. **class** 派生类名: **virtual** 继承方式 基类名
5. {
6. //派生类成员声明
7. }；

这样的话再从Derived1, Derived2 Derived3 ...派生出Final\_Derived时，每个Final\_Derived实例对象只包含一个Base实例。

**15.5 使用关键字final禁止继承(C++11引入)**

从C++11 开始，被声明为final的类不能用作基类，从而禁止继承它。

除了用于类之外，还可将final用于成员函数来控制多态行为。

**15.6 赋值兼容规则**

派生类和基类对象之间的赋值：如果派生类通过public方式继承了基类，那么

a.派生类对象可以赋值给基类对象

b.基类指针可以指向派生类对象

c.基类引用可以指向派生类对象

**16 多态**

多态就是允许用一个函数名的调用来实现不同的功能。按照实现的时机，多态可以分为二类：

1) 编译时多态：在程序编译时就能确定调用哪一个函数，比如函数重载(包括运算符重载)、模板实例化。

2) 运行时多态：在程序运行时才能决定调用的函数，实现运行时多态的条件：a.有虚函数；b.符合赋值兼容规则；c.由指针或引用去调用虚函数。

**16.1 虚函数**

通过基类指针或引用调用虚方法时，如果它指向的是派生类对象，将调用派生类的方法实现。

虚函数的定义格式如下：

1. **class** 类名
2. {
3. ...
4. /\*
5. 1)虚函数必须参在于类的继承环境下才有意义
6. 2)在派生类中实现了同样的函数后，就会覆盖基类中的。在派生类中写或者不写这个virtual都可以
7. \*/
8. **virtual** 返回类型 函数名(形参列表);
9. };

函数隐藏(同名隐藏)：派生类的实现了与基类同名的成员方法(参数列表和返回值不做要求)，并且要求基类的这个方法不能是虚函数。

函数覆盖(虚函数覆盖/重写)：派生类中实现了与基类同名、同参数列表、同返回值的方法。并且要求基类的这个方法是虚函数。也就是说覆盖针对的是多态，函数覆盖就是函数重写，准确地叫做虚函数覆盖和虚函数重写。

**16.2 使用override和final来声明虚函数**

运行时多态(基类指针或引用与派生类对象的动态绑定)是通过同名虚函数的覆盖来实现的，该实现有严格的语法要求(函数名、返回类型、形参列表都要一致)。若不一致就会导致派生类版本的函数没有覆盖掉基类的版本，也不会产生多态的行为。为了避免这类错误可以使用override关键字以显式说明派生类的函数要覆盖基类的虚函数，这样的话编译器就可以帮我们做检查。

被声明为final的类不能作为基类，同样对于被声明为final的虚函数，不能在派生类中被覆盖。

**16.3 虚析构函数**

构造函数不能声明为虚函数: 因为构造函数是编译阶段执行的，而虚函数作为运行过程中多态的基础，是在运行阶段执行的。

对于使用new在堆区中实例化的派生类对象，如果将其赋给基类指针并通过该指针调用delete，将不会调用派生类的析构函数。为避免这种问题，可将析构函数声明为虚函数；这样的话可以确保delete运算符作用于基类指针时，编译器不仅会执行基类的析构函数，还会指针派生类的析构函数。

声明为虚析构函数后，无论派生类对象是使用new在堆区实例化的，还是以局部变量的方式在栈区实例化的，都可以保证程序正常。因此工程中一般情况下都习惯将基类包含一个虚析构函数，即使基类不需要析构函数，也显式地定义一个函数体为空的析构函数。

需要注意将基类的析构函数声明为虚函数时，由该基类所派生的所有派生类的析构函数也都自动声明为虚函数(派生类不加virtual也可以)。

**16.4 多态中基类到派生类的转换**

1. **int** main()
2. {
3. Derived derv;
4. Base& base = derv;
5. **try** {
6. Derived& derived = **dynamic\_cast**<Derived&>(base);
7. derived.my\_func();
8. } **catch**(std::bad\_cast) {
9. cout << "cast failed" << endl;
10. }
11. /\*
12. Base\* base = &derv;
13. Derived\* derived = dynamic\_cast<Derived\*>(base);
14. if(derived) {
15. derived->my\_func();
16. }
17. else {
18. cout << "cast error" << endl;
19. }
20. \*/
21. **return** 0;
22. }

**16.5 纯虚函数和抽象类**

包含有纯虚函数的类是抽象类，抽象类不能实例化对象。这样的类一般只有一个用途，那就是从它派生出其他的类，其中包含的纯虚函数，指定了派生类必须实现哪些函数。

1. **class** AbstractBase {
2. **public**:
3. /\*纯虚函数，告诉编译器AbstractBase的派生类必须要提供该函数的实现。如果派生
4. 类不提供该实现那么派生类仍是一个抽象类\*/
5. **virtual** 返回类型 函数名(形参列表) = 0;
6. }

**17 模板**

模板通过把类型定义为参数，从而实现真正的代码重用。模板分为函数模板和类模板。

使用模板的话需要先实例化模板。实例化指的是为模板指定具体类型，当指定了具体的类型后，就相当于命令编译器来创建一个模板实例(有具体类型的函数或类)。

**17.1 模板的定义和隐式实例化**

函数模板的定义：

1. /\*
2. a. template是一个声明模板的关键字，它表示要声明一个模板。
3. b. typename表示任何类型的意思, 也可以使用class替代typename。
4. c. T是一个类型占位符，在实例化模板时，将使用对象(变量)的具体类型替换它(泛型)；
5. 另外也可在定义模板时为其指定默认类型。
6. d. 当有多个模板参数T的话用逗号隔开
7. \*/
8. **template**<**typename** T1, **typename** T2, ...>
9. 返回类型 函数名(模板形参列表)
10. {
11. //函数体
12. }

函数模板的实例化：

1. /\*对于函数模板来说，其中<具体类型>可以省略，即可以不显式的指定具体类型，
2. 因为编译器可以根据实参的类型自动推断出类型\*/
3. 函数名<具体类型>(实参列表);

类模板的定义:

1. /\*a. 类模板体的成员方法其实都是函数模板。
2. b.与普通类的成员方法类似，类模板的成员方法的定义可以放在类模板的内部，
3. 也可以放在类模板的外部。\*/
4. **template**<**typename** T1, **typename** T2, ...>
5. **class** 类模板名
6. {
7. //类模板体
8. }

类模板的实例化格式：

1. /\*其中<具体类型>不可以省略。
2. 当指定了具体类型之后，就相当于命令编译器使用模板来创建一个类(实例化了一个类模板)。
3. \*/
4. 类模板名<具体类型> 对象名;

**17.2 模板显式实例化(了解即可)**

通过实例化使泛型的函数/类模板，生成一个有具体类型的真正的函数/类。模板实例化分为隐式实例化和显式实例化。

在实际工程项目中，类模板的成员方法可以在类模板内实现，也可以在类模板外实现(类模板内声明类模板外定义)，并且一般情况下都不做分文件编写，都是将其声明和实现都放到同一个文件中(约定俗成.hpp文件)中。但是有的时候个别项目会将其分开(非常不好！)，这个时候就需要用到显式实例化了。

1. /\*类模板的显式实例化。\*/
2. **template** **class** **class**-name<具体类型列表>  //例如：template class Test<short, const char\*>;

同理对于函数模板，如果将其声明和实现分开，也需要用到显式实例化。

1. /\*对于函数模板，和隐式实例化一样，<具体类型列表>可以省略不写\*/
2. **template** **return**-type function-name<具体类型列表>(形参列表); //例如：template void Add(int a, int b);

**17.3 模板具体化(特化)**

当定义的通用模板不能在所有类型实例下正常工作时，需要实现一个针对这种情况的特殊模板，称为模板的特化(具体化)。模板特化分为全特化(限定全部类型参数)、偏特化(半特化, 限定部分类型参数)。

1. 函数模板特化：
2. **template**<**typename** T1, **typename** T2> **return**-type function-name(形参列表) {...}  //通用函数模板
3. **template**<**typename** T1> **return**-type function-name(形参列表) {...}  //偏特化函数模板， 将T2设置为int
4. **template**<> **return**-type function-name(形参列表) {...}  //全特化函数模板
5. 实例化函数模板时，如果有多个模板供选择，编译器将使用特化程度高的模板：
6. function-name(实参列表);
8. 类模板特化：
9. **template**<**typename** T1, **typename** T2> **class** **class**-name {...};  //通用类模板
10. **template**<**typename** T1> **class** **class**-name<T1, **int**> {...};  //偏特化类模板， 将T2设置为int
11. **template**<> **class** **class**-name<**int**, **int**> {...}; //全特化类模板， 将T1设置为int, 将T2设置为int
12. 实例化类模板时，如果有多个模板供选择，编译器将使用特化程度高的模板：
13. **class**-name<**double**, **double**> p1;  //使用通用类模板
14. **class**-name<**double**, **int**> p2;  //使用偏特化类模板
15. **class**-name<**int**, **int**> p3;  //使用全特化类模板

也可以通过为指针提供特殊版本来特化现有模板：

1. **template**<**typename** T> **class** **class**-name {...};  //通用模板
2. **template**<**typename** T\*> **class** **class**-name {...}; //指针特化
3. 如果提供的类型不是指针，则编译器将使用通用版本；如果提供的是指针，则编译器将使用指针特化版本：
4. **class**-name<**char**> p1;   //使用通用class-name模板，T为char
5. **class**-name<**char** \*> p2;  //使用class-name T\*特化，T为char。如果没有进行特化，则将使用通用版本，将T转为char\*

**17.4 智能指针**

智能指针类模板内部重载了解引用运算符(\*)和成员选择运算符(->)，所以可以让程序员像使用普通指针那样使用它们。

如果将new返回的地址赋给这智能指针对象，则无需记住稍后释放这些内存，因为当智能指针对象过期时，其析构函数将使用delete来自动释放内存。

unique\_ptr：它不支持直接拷贝和赋值, 但它支持move语义(移动拷贝和移动赋值)。可以通过relese或者reset这二个成员函数将一个动态内存的所有权从一个unique\_ptr转移给另外一个unique\_ptr。

1. /\*release将p1置为nullptr并返回p1原来的值；这样的话p2就指向了堆区内存。\*/
2. unique\_ptr<**int**> p1(**new** **int**(207));
3. unique\_ptr<**int**> p2(p1.release());
4. /\*reset销毁p3指向的堆区内存，重新指向p2的堆区内存。也就是说p2指向的动态内存转移给了p3\*/
5. unique\_ptr<**int**> p3(**new** **int**(105));
6. p3.reset(p2.release());

其他的几个：

1. shared\_ptr和weak\_ptr
2. 在编写多线程程序时，可考虑使用这二个。它们可帮助实现线程安全和引用计数对象共享。
3. auto\_ptr
4. 从C++11开始已被摒弃。

**17.5 STL引入**

STL(标准模板库)是C++标准库提供的一系列针对常用数据结构操作的类模板和函数模板，所以也称为是一个泛型化的数据结构和算法库。向程序员提供了：

STL容器：包括顺序容器和关联容器，另外STL还提供了被称为容器适配器的类，它们是顺序容器和关联容器的变种，包含的功能有限，用于满足特殊的需求。

1. 1)顺序容器包括：vector、deque、list、forward\_list
2. 2)关联容器：set、unordered\_set、map、unordered\_map、multiset、unordered\_multiset、multimap、unordered\_multimap
3. 3)容器适配器：stack、queue、priority\_queue

STL算法：用于操作容器，比如find、find\_if、reverse、remove\_if、transform等

STL迭代器：将STL中的算法和容器关联起来，STL提供的算法几乎都是通过迭代器实现对容器中元素的操作。

STL字符串类：STL提供了一个专门操纵字符串而设计的模板类：std::basic\_string<T>，该模板类的二个常用具体化如下：

1. 1)std::string, 其为基于**char**的std::basic\_string具体化。
2. 2)std::wstring, 其为基于**wchar\_t**的std::base\_string具体化。