C++关键字

> https://blog.csdn.net/qq_35671135/article/details/88092382
>
> ## 1.关键字分类及简介
>
> https://www.runoob.com/w3cnote/cpp-keyword-intro.html
> https://blog.csdn.net/qq_35671135/article/details/88092382
> https://www.cnblogs.com/zxj9487/p/10964968.html

数据类型：void，int，char，float，double，bool，w_char
类型定义：struct，union，enum，class，typedef
常量值：true，false
类型修饰符：long，short，singed，unsigned
类型限定符：const，volatile，restrict
存储说明符：auto，register，static，extern，thread_local，mutable
其它修饰符：inline，asm
循环控制：for，while，do
跳转控制：break，continue，return，goto
分支结构： if，else，switch，case，default
内存管理：new, delete
运算符：sizeof，and，and_eq，bitand，bitor，compl，not，not_eq，or，or_eq，xor，xor_eq
访问限定符：this，friend，virtual，mutable，explicit，operator
类访问修饰符：private，protected，public
模板：template，typename
命名空间：namespace，using
异常处理：throw，try，catch

1.1 数据类型相关

> https://www.runoob.com/cplusplus/cpp-data-types.html
> https://blog.csdn.net/qq_35671135/article/details/88092382
>
> ---
>
> ### (1)bool、true、false
>
> bool是布尔类型，属于基本类型中的整数类型，取值分为true和false。true和false是具有布尔类型的字面量，是右值（https://zhuanlan.zhihu.com/p/240833006）
> tips:
> 1)左值可在等号左右两边，右值只能在等号右边；
> 2)左值可以寻址，有持久性；
> 3)右值一般是表达式求值过程中创建的无名临时对象，短暂性的。
> 4)左值可以被修改而右值不能
> 5)左值引用：引用一个对象
> 6)右值引用：必须绑定到右值的引用，C++11中右值引用可以实现“移动语义”
> 7)右值引用和相关的移动语义可以避免无谓的复制，提高程序性能
> https://www.jianshu.com/p/94b0221f64a5

```
int x = 6;
int &y = x;
int &z1 = x*6; //不能将一个非常量左值引用与一个右值绑定
const int &z2 = x * 6; //常量左值引用可以与右值绑定
int &&z3 = x * 5;       //正确的右值引用
cout << z2 << endl;
int &&z4 = x; //不能将一个右值引用与一个左值绑定
```

---

(2)char、wchar_t

char表示单字节字符，wchar_t表示多字节字符，char、signed char、unsigned char表示了有符号字符和无符号字符两种类型，char不同编译器可能不同；宽字符的定义是为了可以简化使用国际通用字符集进行的编程(typedef unsigned short wchar_t)

---

(3)int、double、float、short、long、signed、unsigned

signed和unsigned作为前缀修饰整数类型，分别表示有符号和无符号。signed和unsigned修饰char类型，构成unsigned char和signed char，和char都不是相同的类型；不可修饰wchar_t、char16_t和char32_t。其它整数类型的signed省略或不省略，含义不变。signed或unsigned可单独作为类型，相当于signed int和unsigned int。
double和float专用于浮点数，double表示双精度，精度不小于float表示的浮点数。long double则是C++11指定的精度不小于double的浮点数。

---

(4)explicit

explicit在C++中一般只用来修饰构造函数，可以禁止编译器自动调用拷贝初始化(对应直接初始化)，还可以禁止编译器对拷贝函数的参数进行隐式转换
1)explicit只能在类内部使用，不能在外面声明
2)声明了explicit之后，就不允许进行隐式转换了
3)explicit对于多个参数的构造函数没有任何意义

```
#include<iostream>
using namespace std;
class test
{
public:
    explicit test(int x)
    {
        cout << "the int is called" << endl;
    }
    test(char ch)
    {
        cout << "the char is called" << endl;
    }
    test()
    {
        cout << "the none is called" << endl;
    }
    test(int x, int y)
    {
        cout << "the two is called" << endl;
    }
```

```
    test(const test& a)
    {
        cout << "the copy is called" << endl;
    }
};
int main()
{
    test A = 10;
    test A1 = 'c';
    return 0;
}
```

(5)auto decltype

1)auto
将表达式的值赋给变量，需要清楚地知道表达式的类型，auto关键字会根据初始值自动推断变量的数据类型,不是每个编译器都支持auto。
auto x = 7;//推断出x类型为int
auto i=0,j=3.14;//出错，一条声明语句只能有一个数据类型
2)decltype
希望从表达式的类型推断出要定义的变量的类型，但是不想用该表达式的值初始化变量，此时可以使用decltype
decltype(i+j) x = 0;

```
#include<iostream>
using namespace std;
int main()
{
    int x = 3;
    double y = 3.14;
    cout << x+y << endl;
    decltype(x+y) res = x+y;
    cout << res << endl;
    return 0;
}
```

1.2 定义、初始化相关

const、volatile、mutabale、const_cast
1)const
const 是 constant 的缩写，本意是不变的，不易改变的意思。在 C++ 中是用来修饰内置类型变量，自定义对象，成员函数，返回值，函数参数。C++ const 允许指定一个语义约束，编译器会强制实施这个约束，允许程序员告诉编译器某值是保持不变的。如果在编程中确实有某个值保持不变，就应该明确使用const，这样可以获得编译器的帮助。
2)volatile

- 当读取一个变量时，为提高存取速度，编译器优化时有时会先把变量读取到一个寄存器中，以后再取变量值时，就直接从寄存器中取值
- 优化器在用到volatile变量时必须每次都小心地重新读取这个变量的值，而不是使用保存到寄存器里的备份。
- volatile适用于多线程应用中被几个任务共享的变量。

(1)const修饰普通类型变量

(1)不能对一个常量赋值

```
const int a = 7;
int b = a; //可以用一个常量给变量赋值
a = 8;  //错误
```

尝试修改一个const变量，调试窗口中可以看到a的值已经变成了8，但输出结果仍然是7.

```
#include<iostream>
using namespace std;
int main()
{
    const int a = 7;
    int *p = (int*)&a;
    *p = 8;
    cout << a << endl;
    return 0;
}
```

在const之前加上volatile就能成功改变

```
#include<iostream>
using namespace std;
int main()
{
    volatile const int a = 7;
    int *p = (int*)&a;
    *p = 8;
    cout << a << endl;
    return 0;
}
```

const修饰指针变量
(1)const修饰指针所指的对象，对象不可改变
const int* p = 8;
(2)const修饰指针本身，指针本身不能变,内容可变
int *const int
(3)两者都修饰
const int *const int

---

const参数传递和函数返回值
修饰函数参数有三种情况：
(1)值传递的const修饰，*一般而言这种情况不需要const修饰，因为函数会自动产生临时变量复制实参*

```
#include<iostream>
using namespace std;
void print(int a)
{
    a++;
    cout << a << endl;
}
```

```
int main()
{
    int num = 1;
    print(num);
    cout << num << endl;
    return 0;
}
```

指针传递的const修饰，可以防止指针被恶意篡改

```
#include<iostream>
using namespace std;
void print(int *const a)
{
    cout << *a << endl;
    *a = 9;
    //int b = 5;
    //a = &b;//常量指针不能修改
}
int main()
{
    int num = 1;
    print(&num);
    cout << num << endl;
    return 0;
}
```

自定义类型的参数传递，需要临时对象复制参数，对于临时对象的构造，需要调用构造函数，比较浪费时间，因此采用const加引用传递的方式，对于内置类型一般不需要采用引用传递的方式。

---

const修饰函数返回值
(1)内置类型返回值不需要const修饰，即使加了也没效果

```
#include<iostream>
using namespace std;
const int test()
{
    return 1;
}
int main()
{
    int m = test();
    cout << m << endl;
    m++;
    cout << m << endl;
    return 0;
}
```

(2)const 修饰自定义类型的作为返回值，此时返回的值不能作为左值使用，既不能被赋值，也不能被修改。
(3)const 修饰返回的指针或者引用，是否返回一个指向 const 的指针，取决于我们想让用户干什么。

```
#include<iostream>
using namespace std;
```

```
int a = 1;
const int* test()
{
    return &a;
}
int main()
{
    const int* p = test();
    //(*p)++;//出错，不能修改
    a++;//输出2
    cout << *p << endl;
    return 0;
}
```

---

const修饰类成员函数
const 修饰类成员函数，其目的是防止成员函数修改被调用对象的值，如果我们不想修改一个调用对象的值，所有的成员函数都应当声明为const成员函数。
注意：const 关键字不能与 static 关键字同时使用，因为 static 关键字修饰静态成员函数，静态成员函数不含有 this 指针，即不能实例化，const 成员函数必须具体到某一实例。

```
#include<iostream>
using namespace std;
class Test
{
    int a;
    mutable int b;
public:
    Test(int x, int y):a(x),b(y){}
    void test() const
    {
        //a++; //错误
        b++;
    }
};
int main()
{
    Test test(8,7);
    return 0;
}
```

const相关知识点
(1)const数据成员只在某个对象的生存期内是常量，对于整个类而言是可变的。因为类可以创建多个对象，不同的对象其const数据成员的值可以不同。所以不能在类的声明中初始化const数据成员，因为类的对象没被创建时，编译器不知道const数据成员的值是什么。const数据成员的初始化只能在类的构造函数的初始化列表中进行。要想建立在整个类中都恒定的常量，应该用类中的枚举常量来实现，或者static const。
(2)C++11之后允许const成员变量直接初始化
(3)非常量对象可以调用常量成员函数与非常量成员函数，若量函数重载，非常量对象自动调用非常量成员函数版本
(4)常量对象只能调用常量成员函数，不能调用非常量成员函数

```
#include<iostream>
using namespace std;
class Test
```

```
{
    const int a=5;//C++11之后支持
    int b;
public:
    Test(int x, int y):a(),b(y){} //初始化列表初始const数据成员
    void test() const
    {
        cout << "const_fun is called" << endl;
        cout << a << ' ' << b << endl;
    }
    void test()
    {
        //b++; //非常量成员函数可以修改成员变量
        cout << "fun is called" << endl;
        cout << a << ' ' << b << endl;
    }
};
int main()
{
    Test test(8,7);
    test.test();//优先调用nonconst函数
    const Test test1(8,7);
    test1.test();
    return 0;
}
```

(5)const成员函数，不能调用非const成员函数，不能修改成员变量
(6)const成员变量可以通过初始化列表或者类外初始化(C++11之前)
(7)const对象默认为文件局部变量，想在其他文件中访问const变量，必须在文件中显示的指出extern
(8)const 与 define

> https://light-city.club/sc/basic_content/const/

- 定义常量：const int a = 100;
- 类型检查：const定义常量可以进行类型检查
- 节省空间：从汇编角度看，const定义的常量给出了地址，而define给出的是立即数，const定义的常量在程序运行过程中只有一份拷贝，而define可能有多份
- const定义的变量只有类型为整数或者枚举，且以常量表达式初始化才能作为常量表达式。其他情况下只是const限定的变量。
  (9)提高效率：编译器通常不为普通const常量分配存储空间，而是将他们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作。第一次使用时为其分配内存，在程序结束的时候释放；const全局变量存储在只读数据段中，const局部变量存储在栈中。
  (10)const_cast(expression)可修改const属性。
  (11)整个类的常量的实现(希望某些常量只在类中有效)

```
方法一：
class A
{
    enum{SIZE1 = 100, SIZE2 = 200};         //枚举常量
    int arrayA[SIZE1];
    int arrayB[SIZE2];
};
方法二：
class Test
{
public:
```

```
      Test():a(0){}
      enum {size1=100,size2=200};
private:
      const int a;//只能在构造函数初始化列表中初始化
       static int b;//在类的实现文件中定义并初始化
      const static int c;//与 static const int c;相同。
};

int Test::b=0;//static成员变量不能在构造函数初始化列表中初始化，因为它不属于某个对象。
cosnt int Test::c=0;//注意：给静态成员变量赋值时，不需要加static修饰符。但要加cosnt
```

(12)const变量必须进行初始化
(13)一点有意思的语法

```
#include<iostream>
using namespace std;
int main()
{
    double dval = 3.14;
    const int &ri = dval; //int &ri = dval;错误
    cout << ri << endl;
    return 0;
}
```

(14)常量表达式是指值不会改变并且在编译过程中就能得到计算结果的表达式

---

(2)enum关键字的应用

> https://www.runoob.com/w3cnote/cpp-enum-intro.html
> enum 类型名 {枚举常量表};
> enum fruit_set {apple, orange, banana=1, peach, grape}//枚举常量apple=0,orange=1,
> banana=1,peach=2,grape=3。
> tips:
> (1)枚举常量不会占用对象空间，他们在编译时被全部求值
> (2)隐含的数据类型是整数，最大值有限而且不能表示浮点数

---

(3)export

使用该关键字可实现模板函数的外部调用。对模板类型，可以在头文件中声明模板类和模板函数；在代码文件中，使用关键字export来定义具体的模板类对象和模板函数；然后在其他用户代码文件中，包含声明头文件后，就可以使用该这些对象和函数

---

(4)extern

1)extern介绍
extern（外部的）声明变量或函数为外部链接，即该变量或函数名在其它文件中可见。被其修饰的变量（外部变量）是静态分配空间的，即程序开始时分配，结束时释放。用其声明的变量或函数应该在别的文件或同一文件的其它地方定义（实现）。在文件内声明一个变量或函数默认为可被外部使用。在 C++ 中，还可用来指定使用另一语言进行链接，这时需要与特定的转换符一起使用。目前仅支持 C 转换标记，来支持 C 编译器链接。使用这种情况有两种形式：
extern "C"
{

}
extern const改变const变量可见性
2)C++与C编译的区别
在C++中常在头文件见到extern "C"修饰函数，那有什么作用呢？ 是用于C++链接在C语言模块中定义的函数。

C++虽然兼容C，但C++文件中函数编译后生成的符号与C语言生成的不同。因为C++支持函数重载，C++函数编译后生成的符号带有函数参数类型的信息，而C则没有。

例如int add(int a, int b)函数经过C++编译器生成.o文件后，add会变成形如add_int_int之类的, 而C的话则会是形如_add, 就是说：相同的函数，在C和C++中，编译后生成的符号不同。

这就导致一个问题：如果C++中使用C语言实现的函数，在编译链接的时候，会出错，提示找不到对应的符号。此时extern "C"就起作用了：告诉链接器去寻找_add这类的C语言符号，而不是经过C++修饰的符号。
3)C++和C互调方法

```
C++调用C
//xx.h
extern int add(...)

//xx.c
int add(){

}

//xx.cpp
extern "C" {
    #include "xx.h"
}
C调用C++
//xx.h
extern "C"{
    int add();
}
//xx.cpp
int add(){

}
//xx.c
extern int add();
```

(5)public、protected、private

1)这三个都为权限修饰符。public为公有的，访问不受限制；protected为保护的，只能在本类和友元中访问；private为私有的，只能在本类、派生类和友元中访问。
2)class默认访问权限是private的，struct默认访问权限是public的。
3)public继承，派生类成员函数可以访问public、protected成员变量，派生类实例只能访问public成员变量；protected继承，派生类成员函数可以访问public、protected成员变量，派生类实例无法访问任何基类成员；private继承，成员函数可以访问public、protected成员，派生类实例无法访问任何基类成员。

(6)template

声明一个模板，模板函数，模板类等。模板的特化。

(7)static

> https://www.runoob.com/w3cnote/cpp-static-usage.html
> https://light-city.club/sc/basic_content/static/
> 可修饰变量、也可以修饰函数和类中的成员函数。static修饰的变量的周期是整个函数的生命周期。具有静态生存期的变量，只在函数第一次调用时进行初始化，在没有显示初始化的情况下，系统将他们初始化为0.
> 1)函数中的静态变量

```
#include<iostream>
using namespace std;
void test()
{
    static int num = 0;
    cout << num << ' ';
    num++;
}
int main()
{
    for(int i=0; i<10; ++i)
    {
        test();
    }
    return 0;
}
```

2)类中的静态变量
由于声明为static的变量只被初始化一次，因为它们在单独的静态存储中分配了空间，因此类中的静态变量由对象共享。对于不同的对象，不能有相同静态变量的多个副本。也是因为这个原因，静态变量不能使用构造函数初始化。

```
#include<iostream>
using namespace std;
class Test
{
public:
    static int num;
    Test(){};
};
int Test::num = 5;
int main()
{
    Test obj1,obj2;
    obj1.num = 1;
    obj2.num = 3;
    cout << obj1.num << ' ' << obj2.num << endl;
    return 0;
}
```

3)类对象为静态

```
```

```
//非静态对象
#include<iostream>
using namespace std;
class Test
{
public:
    static int num;
    Test()
    {
        cout << "constructed is called" << endl;
    }
    ~Test()
    {
        cout << "deconstructed is called" << endl;
    }
};
int Test::num = 5;
int main()
{
    int x=0;
    if(x==0)
    {
        Test obj;
    }
    cout << "end of main \n";
}
constructed is called
deconstructed is called
end of main
//静态对象
#include<iostream>
using namespace std;
class Test
{
public:
    static int num;
    Test()
    {
        cout << "constructed is called" << endl;
    }
    ~Test()
    {
        cout << "deconstructed is called" << endl;
    }
};
int Test::num = 5;
int main()
{
    int x=0;
    if(x==0)
    {
        static Test obj;
    }
    cout << "end of main \n";
}
constructed is called
end of main
deconstructed is called
```

4)类中的静态函数
就像类中的静态数据成员或静态变量一样，静态成员函数也不依赖于类的对象。我们被允许使用对象和'.'来调用静态成员函数。但建议使用类名和范围解析运算符调用静态成员。静态成员函数只能访问静态数据成员和其他静态成员函数，他们无法访问类的非静态成员或非静态成员函数。
5)static的作用

- static 是 C/C++ 中很常用的修饰符，它被用来控制变量的存储方式和可见性
- 在修饰变量的时候，static 修饰的静态局部变量只执行初始化一次，而且延长了局部变量的生命周期，直到程序运行结束以后才释放
- static 修饰全局变量的时候，这个全局变量只能在本文件中访问，不能在其它文件中访问，即便是 extern 外部声明也不可以。
- static 修饰一个函数，则这个函数的只能在本文件中调用，不能被其他文件调用。static 修饰的变量存放在全局数据区的静态变量区，包括全局静态变量和局部静态变量，都在全局数据区分配内存。初始化的时候自动初始化为 0。
- 不想被释放的时候，可以使用static修饰。比如修饰函数中存放在栈空间的数组。如果不想让这个数组在函数调用结束释放可以使用 static 修饰
- 考虑到数据安全性（当程序想要使用全局变量的时候应该先考虑使用 static）
  6)为什么需要static
- 想将函数中的的变量保存到下一次调用(比如记录某个函数被调用了几次)：全局变量会破坏变量的访问范围
- 让某一个数据成员为整个类而不是某个对象服务，却不想破坏封装性，要求成员隐藏在类的内部，对外不可见，可以将其定义为静态数据
  7)静态数据的存储
  全局（静态）存储区：分为 DATA 段和 BSS 段。DATA 段（全局初始化区）存放初始化的全局变量和静态变量；BSS 段（全局未初始化区）存放未初始化的全局变量和静态变量。程序运行结束时自动释放。其中BBS段在程序执行之前会被系统自动清0，所以未初始化的全局变量和静态变量在程序执行之前已经为0。存储在静态数据区的变量会在程序刚开始运行时就完成初始化，也是唯一的一次初始化。

在 C++ 中 static 的内部实现机制：静态数据成员要在程序一开始运行时就必须存在。因为函数在程序运行中被调用，所以静态数据成员不能在任何函数内分配空间和初始化。

这样，它的空间分配有三个可能的地方，一是作为类的外部接口的头文件，那里有类声明；二是类定义的内部实现，那里有类的成员函数定义；三是应用程序的 main() 函数前的全局数据声明和定义处。

静态数据成员要实际地分配空间，故不能在类的声明中定义（只能声明数据成员）。类声明只声明一个类的"尺寸和规格"，并不进行实际的内存分配，所以在类声明中写成定义是错误的。它也不能在头文件中类声明的外部定义，因为那会造成在多个使用该类的源文件中，对其重复定义。

static 被引入以告知编译器，将变量存储在程序的静态存储区而非栈上空间，静态数据成员按定义出现的先后顺序依次初始化，注意静态成员嵌套时，要保证所嵌套的成员已经初始化了。消除时的顺序是初始化的反顺序。

优势：可以节省内存，因为它是所有对象所公有的，因此，对多个对象来说，静态数据成员只存储一处，供所有对象共用。静态数据成员的值对每个对象都是一样，但它的值是可以更新的。只要对静态数据成员的值更新一次，保证所有对象存取更新后的相同的值，这样可以提高时间效率。
8)C++中static用法
类名：：变量名直接引用
类名：：方法名直接引用
对象.变量名
对象.类名
9)为什么静态成员函数不能调用非静态成员和非静态成员函数
静态成员函数在类实例化对象之前就已经分配了空间，而类的非静态成员在实例化之后才有内存空间，相当于使用了一个还没有声明的变量

10)不能通过类名来调用非静态成员函数，可以通过对象类调用静态成员函数和非静态成员函数
11)类的非静态成员函数可以调用静态成员函数
12)类中的静态成员变量在使用前必须初始化

---

(8)struct、class、union

用于类型声明，class有的功能struct基本都有，二者默认访问权限不同，class默认private,struct默认public

---

1.C中的struct

- 单纯用作数据复合类型，只能将数据成员放在里面，而不能将函数放在里面
- C结构体中不能使用C++访问修饰符
- 定义时必须使用struct
- C的结构体没有继承
- 结构体的名字和函数名相同可以正常运行且调用

```
#include<stdio.h>

struct Base {            // public
    int v1;
//    public:      //error
        int v2;
    //private:
        int v3;
    //void print(){       // c中不能在结构体中嵌入函数
    //    printf("%s\n","hello world");
    //};    //error!
};

void Base(){
    printf("%s\n","I am Base func");
}
//struct Base base1;  //ok
//Base base2; //error
int main() {
    struct Base base;
    base.v1=1;
    //base.print();
    printf("%d\n",base.v1);
    Base();
    return 0;
}
```

- C中结构体主要作用是封装，封装的好处是重复利用简单
  2.C++中的struct
- C++结构体中不仅可以定义数据，还可以定义函数。
- C++结构体中可以使用访问修饰符，如：public、protected、private
- C++结构体使用可以直接使用不带struct
- C++继承
- 若结构体的名字与函数名相同，可以正常运行且正常的调用！但是定义结构体变量时候只用用带struct的！
  3.可以使用typedef定义结构体别名

4.结构体大小与内存对齐方式
经过内存对齐之后，CPU访问内存的速度可以大为提升

对齐规则：

- 第一个成员在与结构体变量偏移为0的地址处
- 其他成员变量要对齐到某个数字（对齐数）的整数倍的地址处。(编译器默认的对齐数与成员大小的较小值)
- 结构体总大小是最大对齐数的整数倍(结构体的每个成员变量都有一个对齐数)，取最大
- 如果嵌套了结构体的情况，嵌套的结构体对齐到自己成员最大对齐数的整数倍，结构体的整体大小就是所有最大对齐数（含嵌套结构体的对齐数）的整数倍。
- 使用结构体时，占用空间较小的元素几种在一起可以节省空间
- 结构体的总大小sizeof()的结果，min(#pargram pack(), 内部长度最长的数据成员)
  修改编译器默认字节对齐方法：

> https://blog.csdn.net/m0_37433111/article/details/110882008
> 5.union简介
> 联合（union）是一种节省空间的特殊的类，一个 union 可以有多个数据成员，但是在任意时刻只有一个数据成员可以有值。当某个成员被赋值后其他成员变为未定义状态。联合有如下特点：

- 默认访问控制符为 public
- 可以含有构造函数、析构函数
- 不能含有引用类型的成员
- 不能继承自其他类，不能作为基类
- 不能含有虚函数
- 匿名 union 在定义所在作用域可直接访问 union 成员
- 匿名 union 不能包含 protected 成员或 private 成员
- 全局匿名联合必须是静态（static）的
  union的应用：
  (1)测试大小端CPU大小端：

> https://blog.csdn.net/ybhuangfugui/article/details/99669954
> https://www.cnblogs.com/jeakeven/p/5113508.html

```
#include <iostream>
using namespace std;

void checkCPU()
{
    union MyUnion{
        int a;
        char c;
    }test;
    test.a = 1;
    if (test.c == 1) //低地址存放低字节是小端模式
        cout << "little endian" <<endl;
    else cout << "big endian" <<endl;
}

int main()
{
    checkCPU();
    return 0;
}
```

(9)mutable

mutable也是为了突破const的限制而设置的。被mutable修饰的变量，将永远处于可变的状态，即使在一个const函数中。

```
#include<iostream>
using namespace std;
class Test
{
    mutable int num = 1;
public:
    void test() const
    {
        num++;
        cout << num << endl;
    }
};
int main()
{
    Test test;
    test.test();
    return 0;
}
```

应用：基于mutex的线程安全的队列

(10)virtual

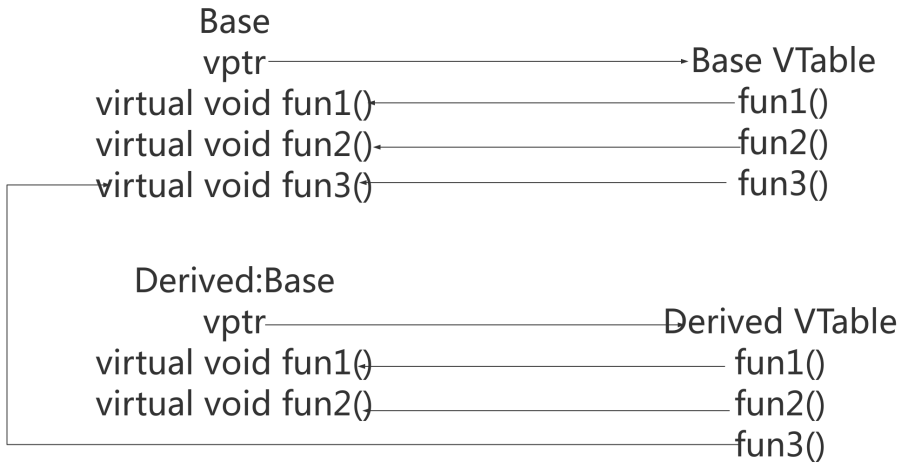
1.纯虚函数
C++中的纯虚函数(或抽象函数)是我们没有实现的虚函数！我们只需声明它!通过声明中赋值0来声明纯虚函数！包含纯虚函数的类叫作抽象类，抽象类只能作为基类来派生新类来使用，不能实例化，抽象类的指针和应用->派生类。

2.实现抽象类
抽象类中：在成员函数内可以调用纯虚函数，在构造函数/析构函数内部不能使用纯虚函数。如果一个类从抽象类派生而来，它必须实现了基类中的所有纯虚函数，才能成为非抽象类。

- 抽象类可以有构造函数
- 构造函数不能是虚函数，而析构函数一般是虚函数当基类指针指向派生类对象并删除对象时，我们可能希望调用适当的析构函数。如果析构函数不是虚拟的，则只能调用基类析构函数。
- 如果抽象类没有数据成员，可以不写，如果有，最好写

3.C++虚函数的vptr和vtable
为了实现虚函数，C ++使用一种称为虚拟表的特殊形式的后期绑定。该虚拟表是用于解决在动态/后期绑定方式的函数调用函数的查找表。虚拟表有时会使用其他名称，例如“vtable”，“虚函数表”，“虚方法表”或“调度表”
虚拟表实际上非常简单，虽然用文字描述有点复杂。首先，每个使用虚函数的类（或者从使用虚函数的类派生）都有自己的虚拟表。该表只是编译器在编译时设置的静态数组。虚拟表包含可由类的对象调用的每个虚函数的一个条目。此表中的每个条目只是一个函数指针，指向该类可访问的派生函数。
其次，编译器还会添加一个隐藏指向基类的指针，我们称之为vptr。vptr在创建类实例时自动设置，以便指向该类的虚拟表。与this指针不同，this指针实际上是编译器用来解析自引用的函数参数，vptr是一

个真正的指针。因此，它使每个类对象的分配大一个指针的大小。这也意味着vptr由派生类继承，这很重要。
创建类实例(构造函数)->添加vptr(隐藏指向基类的指针)->指向此类的虚拟表



我们发现C++的动态多态性是通过虚函数来实现的。简单的说，通过virtual函数，指向子类的基类指针可以调用子类的函数

4.虚函数相关问题

- 静态函数不可以声明为虚函数，同时也不能被const 和 volatile关键字修饰
- static成员函数不属于任何类对象或类实例，所以即使给此函数加上virutal也是没有任何意义
- 虚函数依靠vptr和vtable来处理。vptr是一个指针，在类的构造函数中创建生成，并且只能用this指针来访问它，静态成员函数没有this指针，所以无法访问vptr(this->vptr->vtable)
- 构造函数不可以声明为虚函数。同时除了inline|explicit之外，构造函数不允许使用其它任何关键字。
- 尽管虚函数表vtable是在编译阶段就已经建立的，但指向虚函数表的指针vptr是在运行阶段实例化对象时才产生的。 如果类含有虚函数，编译器会在构造函数中添加代码来创建vptr。 问题来了，如果构造函数是虚的，那么它需要vptr来访问vtable，可这个时候vptr还没产生。 因此，构造函数不可以为虚函数。我们之所以使用虚函数，是因为需要在信息不全的情况下进行多态运行。而构造函数是用来初始化实例的，实例的类型必须是明确的。 因此，构造函数没有必要被声明为虚函数。
- 析构函数可以声明为虚函数。如果我们需要删除一个指向派生类的基类指针时，应该把析构函数声明为虚函数。 事实上，只要一个类有可能会被其它类所继承， 就应该声明虚析构函数(哪怕该析构函数不执行任何操作)。
- 通常类成员函数都会被编译器考虑是否进行内联。 但通过基类指针或者引用调用的虚函数必定不能被内联。 当然，实体对象调用虚函数或者静态调用时可以被内联，虚析构函数的静态调用也一定会被内联展开。
- 虚函数可以是内联函数，内联是可以修饰虚函数的，但是当虚函数表现多态性的时候不能内联。
- 内联是在编译器建议编译器内联，而虚函数的多态性在运行期，编译器无法知道运行期调用哪个代码，因此虚函数表现为多态性时（运行期）不可以内联。
- inline virtual 唯一可以内联的时候是：编译器知道所调用的对象是哪个类（如 Base::who()），这只有在编译器具有实际对象而不是对象的指针或引用时才会发生
- RTTI（Run-Time Type Identification)，通过运行时类型信息程序能够使用基类的指针或引用来检查这些指针或引用所指的对象的实际派生类型。

1.3系统操作相关

(1)catch、throw、try

用于异常处理。try指定try块的起始，try块后的catch可以捕获异常。异常由throw抛出。throw在函数中还表示动态异常规范

(2)new、delete

new、delete属于操作符，可以被重载。new表示向内存申请一段新的空间，申请失败会抛出异常。new会先调用operator new函数，再在operator new函数里调用malloc函数分配空间，然后再调构造函数。delete不仅会清理资源，还会释放空间。delete县调用析构函数，其次调用operator delete函数，最后在operator delete函数里面调用free函数。malloc申请内存失败会返回空。free只是清理了资源，并没有释放空间

(3)friend

友元。使其不受访问权限控制的限制。例如，在1个类中，私有变量外部是不能直接访问的。可是假如另外1个类或函数要访问本类的1个私有变量时，可以把这个函数或类声明为本类的友元函数或友元类。这样他们就可以直接访问本类的私有变量

- 提供了一种 普通函数或者类成员函数 访问另一个类中的私有或保护成员的机制
- 提高了程序的运行效率
- 破坏了类的封装性和数据的透明性
- 能访问私有成员 - 破坏封装性 - 友元关系不可传递 - 友元关系的单向性 - 友元声明的形式及数量不受限制
- 友元没有继承性和传递性

(4)inline

内联函数，在编译时将所调用的函数代码直接嵌入到主调函数中。各个编译器的实现方式可能不同。

- 类内定义为隐式内联
- 内联能提高函数效率，但并不是所有的函数都定义成内联函数！内联是以代码膨胀(复制)为代价，仅仅省去了函数调用的开销，从而提高函数的执行效率。
- 如果执行函数体内代码的时间相比于函数调用的开销较大，那么效率的收货会更少！
- 另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。
  以下情况不宜用内联：
  如果函数体内的代码比较长，使得内联将导致内存消耗代价比较高。
  如果函数体内出现循环，那么执行函数体内代码的时间要比函数调用的开销大。

---

(5)operator

和操作符连用，指定一个重载了的操作符函数，比如，operator+

(6)register

提示编译器尽可能把变量存入到CPU内部寄存器中

(7)typename

typename关键字告诉编译器把一个特殊的名字解释为一个类型

1.4 命名相关

(1)using

- 在当前文件引入命名空间，例using namespace std
- 在子类中使用，using声明引入基类成员名称
- 在继承过程中，派生类可以覆盖重载函数的0个或多个实例，一旦定义了一个重载版本，那么其他的重载版本都会变为不可见。如果对于基类的重载函数，我们需要在派生类中修改一个，又要让其他的保持可见，必须要重载所有版本，这样十分的繁琐
- C中常用typedef A B这样的语法，将B定义为A类型，也就是给A类型一个别名B,对应typedef A B，使用using B=A可以进行同样的操作。

---

(2)namespace

C++标准程序库中的所有标识符都被定义于一个名为std的namespace中。命名空间除了系统定义的名字空间之外，还可以自己定义，定义命名空间用关键字“namespace”，使用命名空间时用符号“::”指定。

(3)typedef

typedef声明，为现有数据类型创建一个新的名字。便于程序的阅读和编写

1.5 函数和返回值相关

(1)sizeof

返回类型名或表达式具有的类型对应的大小

- 空类的大小为1字节
- 一个类中，虚函数本身、成员函数（包括静态与非静态）和静态数据成员都是不占用类对象的存储空间。
- 对于包含虚函数的类，不管有多少个虚函数，只有一个虚指针,vptr的大小。
- 普通继承，派生类继承了所有基类的函数与成员，要按照字节对齐来计算大小
- 虚函数继承，不管是单继承还是多继承，都是继承了基类的vptr。(32位操作系统4字节，64位操作系统 8字节)!

(2)typeid

typeid是一个操作符，返回结果为标准库种类型的引用
运行时类型识别(RTTI)

- dynamic_cast允许运行时刻进行类型转换，从而使程序能够在一个类层次结构中安全地转化类型，与之相对应的还有一个非安全的转换操作符static_cast
- typeid是C++的关键字之一，等同于sizeof这类的操作符。typeid操作符的返回结果是名为type_info的标准库类型的对象的引用

---

(3)this

每个类成员函数都隐含了一个this指针，用来指向类本身。this指针一般可以省略。但在赋值运算符重载的时候要显示使用。静态成员函数没有this指针

---

(4)*_cast

C++类型风格来性转换。const_cast删除const变量的属性，方便赋值；dynamic_cast用于将一个父类对象的指针转换为子类对象的指针或引用；reinterpret_cast将一种类型转换为另一种不同的类型；static_cast用于静态转换，任何转换都可以用它，但他不能用于两个不相关的类型转换
