C++强制类型转换：

<https://blog.csdn.net/u014624623/article/details/79837849>

1. static\_cast:用于静态类型转换（非多态类型）

static\_cast< new\_type >(expression)

1. 基本数据类型的转换，与C中的隐式类型类似，但不能用于不相关类型的转换，如整型转换为整型指针。
2. static\_cast不能转换掉const,volatile
3. 基类与派生类的转换

例：程序gaode\_test

classBase

{

public:

virtualvoidfunc()

{

*cout*<<"Print Base"<<*endl*;

}

};

classDerived : publicBase

{

public:

voidfunc()

{

*cout*<<"Print Derived"<<*endl*;

}

};

inti = 10;

doubled1 = i; // 隐式类型转换b

doubled2 = static\_cast<double>(i);

doubled3 = 4.5;

double\* pd = &d3;

void\* d = static\_cast<void\*>(pd); // 指针转换为void指针

double\* pd1 = static\_cast<double\*>(d); // void指针转换为其它指针

*cout*<< \*pd1<<*endl*;

inta = 10;

constintb = static\_cast<constint>(a);

constintc = 5;

// int d = static\_cast<int>(c); // error，static\_cast不能转换掉const

// int\* p = static\_cast<int\*>(a); // error，类型转换无效，不能从int转换为int\*

//void p; // error，不允许不完整类型

// void\* p = static\_cast<int\*>(a); // error，类型转换无效

void\* p = static\_cast<int\*>(&a);

Basebb;

Deriveddd;

Base&bb1 = static\_cast<Base&>(dd); // 上行，安全

Base\* pb = static\_cast<Base\*>(&dd); // 上行，安全

pb->func(); // Print Derived

Derived&dd1 = static\_cast<Derived&>(bb); // 下行，不安全

Derived\* pdd = static\_cast<Derived\*>(&bb); // 下行，不安全

pdd->func(); // 在VS上为Print Base，但其它编译器不一定

1. dynamic\_cast:动态类型转换，只能用于完整类类型或void指针和引用的转换，并且相互转换的类类型必须构成多态。

在下行转换时，dynamic\_cast相比static\_cast要安全，转换的结果会有所不同

例：程序gaode\_test1

inta = 9;

// double b = dynamic\_cast<double>(a); // error，dynamic\_cast只能用于转换指针和引用

// double\* pd = dynamic\_cast<double\*>(&a); // error，dynamic\_cast只能用于完整类类型或void的指针和引用

Baseb;

Derivedd;

Base&bb = dynamic\_cast<Base&>(d); // 上行，安全与static\_cast相同

Base\* pb = dynamic\_cast<Base\*>(&d); // 上行，与static\_cast相同

pb->func(); // Print Derived

Derived\* pd = dynamic\_cast<Derived\*>(&b); // 下行，pd为nullptr

Derived&d1 = dynamic\_cast<Derived&>(b); // 下行，会抛出异常

从上面的程序可以看出，在基类与派生类的上行转换中，派生类的对象被强制转换为基类的指针或引用，但调用的虚函数仍然是派生类的，这是由于虚函数指针的缘故，派生类对象地址中存储的是派生类虚函数的函数指针。

1. const\_cast:去除类型的const或volatile属性

例：程序gaode\_test2

const int i = 10;

// int& r = i; // error，左值引用无法绑定常量

int& r = const\_cast<int&>(i); // ok

r = 100;

// 通过引用修改了r后，i并未发生变化

// 这是由于对于const变量，编译器会进行优化，

// 将其放入寄存器中，每次使用直接从寄存器取值

// 即使在内存中修改了它的值，编译器也无法知晓

*cout* << "r = " << r << *endl*; // 100

*cout* << "i = " << i << *endl*; // 10

// 使用volatile关键字，保证编译器

// 每次使用都从内存中取值

const volatile int j = 10;

int& r1 = const\_cast<int&>(j);

r1 = 100;

*cout* << "r1 = " << r1 << *endl*; // 100

*cout* << "j = " << j << *endl*; // 100

Qt信号与槽

1. 类型安全

信号参数类型和参数个数与槽函数的参数类型和个数相同，不过槽的参数可以少于信号参数个数，但缺少的参数必须是信号参数的最后一个或多个。

1. 松散耦合

激发信号的Qt对象不需要知道哪个对象的哪个槽关联到信号，同样，对象的槽也不知道哪个信号关联了自己。这类似观察者模式。

Qt中的信号与槽机制其实是按名称查表，这个表称为元数据表。

模板实现单例模式

例：程序gaode\_test3

class noncopyable {

protected:

noncopyable() {}

~noncopyable() {}

private:

noncopyable(const noncopyable&);

const noncopyable& operator=(const noncopyable&);

};

template<typename T>

class Singleton : public noncopyable

{

public:

static T& GetInstance()

{

static T instance;

return instance;

}

};

class Test

{

friend class Singleton<Test>;

private:

Test()

{

++count\_;s

}

~Test()

{}

public:

void Print()

{

*cout* << "instance count " << count\_ << *endl*;

}

public:

static int count\_;

};

int Test::count\_ = 0;

int main(int argc, char\* argv[])

{

Test& s1 = Singleton<Test>::GetInstance();

s1.Print();

Test& s2 = Singleton<Test>::GetInstance();

s2.Print();

return 0;

}

内联函数与宏定义的区别

函数调用过程：调用某个函数实际上将程序执行顺序转移到该函数所存放在内存中某个地址，将函数的程序内容执行完后，再返回到转去执行该函数前的地方。这种转移操作要求在转去前要保护现场并记忆执行的地址，转回后先要恢复现场，并按原来保存地址继续执行，这就是所谓的压栈和出栈。因此，函数调用要有一定的时间和空间方面的开销。

内联函数：在程序编译时，编译器将程序中出现的内联函数的调用表达式用内联函数的函数体直接进行替换，这样就不会产生转去转回的问题，节省了时间开销，但会增加空间开销。

在类中声明同时定义的函数，自动转化为内联函数，但不一定就是按内联方式调用。

内联函数中一般不包含for，switch，while等复杂的循环语句，函数体也不长，否则即使被声明为inline，编译也会自动放弃内联方式，采用普通函数调用方式。

内联函数与宏定义的区别：

1. 内联函数是函数，由编译器处理，会进行参数类型检查；宏定义由预处理器处理，不会进行参数类型检查，其相对来说不安全。
2. 内联函数被编译器将代码直接插入调用处，而宏定义则是由预处理器进行直接替代（类似字符串的替换）。