C++ vs C

一、命名空间

1. 为什么要使用命名空间?

一个大型的工程往往是由若干个人独立完成的，不同的人分别完成不同的部分，最后再组合成一个完整的程序。由于各个头文件是由不同的人设计的，有可能在不同的头文件中用了相同的名字来命名所定义的类或函数，这样在程序中就会出现名字冲突。不仅如此，有可能我们自己定义的名字会与C++库中的名字发生冲突。

名字冲突就是在同一个作用域中有两个或多个同名的实体，为了解决命名冲突，C++中引入了**命名空间**，所谓命名空间就是一个可以由用户自己定义的作用域，在不同的作用域中可以定义相同名字的变量，互不干扰，系统能够区分它们。

2. 什么是命名空间?

命名空间又称为名字空间，是程序员命名的内存区域，程序员根据需要指定一些有名字的空间域，把一些全局实体分别存放到各个命名空间中，从而与其他全局实体分隔开。通俗的说，每个名字空间都是一个名字空间域，存放在名字空间域中的全局实体只在本空间域内有效。名字空间对全局实体加以域的限制，从而合理的解决命名冲突。

C++中定义命名空间的基本格式如下：

```
namespace wd
{
int val1 = 0;
char val2;
}// end of namespace wd
```

在声明一个命名空间时，大括号内不仅可以存放变量，还可以存放以下类型：

- 变量
- 常量
- 函数，可以是定义或声明
- 结构体
- 类
- 模板
- 命名空间，可以嵌套定义

```
  namespace wd
  {
  int number = 0;
  struct Foo
  {
      char ch;
      int val;
  };
  void display();
  }// end of namespace wd
```

定义在名称空间中的变量或者函数都称为实体，名称空间中的实体作用域是全局的，并不意味着其可见域是全局的。

如果不使用作用域限定符和using机制，抛开名称空间嵌套和内部屏蔽的情况，实体的可见域是从实体创建到该名称空间结束。

在名称空间外，该实体是不可见的。

3.命名空间的使用方式

命名空间一共有三种使用方式，分别是using编译指令、作用域限定符、using声明机制。

3.1、using编译指令

我们接触的第一个C++程序基本上都是这样的，其中std代表的是标准命名空间。

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    cout << "hell,world" << endl;
    return 0;
}
```

其中第二行就使用了using编译指令。如果一个名称空间中有多个实体，使用using编译指令，就会把该空间中的所有实体一次性引入到程序之中；对于初学者来说，如果对一个命名空间中的实体并不熟悉时，直接使用这种方式，有可能还是会造成名字冲突的问题，而且出现错误之后，还不好查找错误的原因，比如下面的程序就会报错，当然该错误是人为造成的。

```
#include <iostream>
using namespace std;
double cout()
{
    return 1.1;
}

int main(void)
{
    cout();
    return 0;
}
```

3.2、作用域限定符

第二种方式就是直接使用作用域限定符::啦。每次要使用某个名称空间中的实体时，都直接加上，例如：

```
namespace wd
{
int number = 10;
void display()
{
    //cout,endl都是std空间中的实体，所以都加上'std::'命名空间
    std::cout << "wd::display()" << std::endl;
}
}//end of namespace wd

```

```
int main(void)
{
    std::cout << "wd::number = " << wd::number << endl;
    wd::display();
}
```

这种方式会显得比较冗余，所以还可以采用第三种使用方式。

3.3、using声明机制

using声明机制的作用域是从using语句开始，到using所在的作用域结束。要注意，在同一作用域内用using声明的不同的命名空间的成员不能有同名的成员，否则会发生重定义。

```
#include <iostream>
using std::cout;
using std::endl;
namespace wd
{
int number = 10;
void display()
{
    cout << "wd::display()" << endl;
}
}//end of namespace wd

using wd::number;
using wd::display;
int main(void)
{
    cout << "wd::number = " << number << endl;
    wd::display();
}
```

在这三种方式之中，我们推荐使用的就是第三种，需要哪个实体的时候就引入到程序中，不需要的实体就不引入，尽可能减小犯错误的概率。

4. 匿名命名空间

命名空间还可以不定义名字，不定义名字的命名空间称为匿名命名空间。由于没有名字，该空间中的实体，其它文件无法引用，它只能在本文件的作用域内有效，它的作用域是从匿名命名空间声明开始到本文件结束。在本文件使用无名命名空间成员时不必用命名空间限定。其实匿名命名空间和static是同样的道理，都是只在本文件内有效，无法被其它文件引用。

```
namespace {
int val1 = 10;
void func();
}//end of anonymous namespace
```

在匿名空间中创建的全局变量，具有全局生存期，却只能被本空间内的函数等访问，是static变量的有效替代手段。

5. 命名空间的嵌套及覆盖

```
int number = 1;
namespace wd
{
int number = 10;

namespace wh
{
int number = 100;
void display()
{
    cout << "wd::wh::display()" << endl;
}
}//end of namespace wh

void display(int number)
{
    cout << "形参number = " << number << endl;
    cout << "wd命名空间中的number = " << wd::number << endl;
    cout << "wh命名空间中的number = " << wd::wh::number << endl;
}

}//end of namespace wd

int main(void)
{
    using wd::display;
    display();
    return 0;
}
```

6. 对命名空间的思考和总结

下面引用当前流行的名称空间使用指导原则：

- [ ] 提倡在已命名的名称空间中定义变量，而不是直接定义外部全局变量或者静态全局变量。
- [ ] 如果开发了一个函数库或者类库，提倡将其放在一个名称空间中。
- [ ] 对于using 声明，首先将其作用域设置为局部而不是全局
- [ ] 不要在头文件中使用using编译指令，这样，使得可用名称变得模糊，容易出现二义性，
- [ ] 包含头文件的顺序可能会影响程序的行为，如果非要使用using编译指令，建议放在所有#include预编译指令后。

二、const关键字的用法

1. const关键字修饰变量

```
const int number1 = 10;//const关键字修饰的变量称为常量
int const number2 = 20;

const int val;//error  常量必须要进行初始化
```

除了这种方式可以创建常量外，还可以使用宏定义的方式创建常量

```
#define NUMBER 1024
```

常考题：const常量与宏定义的区别是什么?

1）编译器处理方式不同。宏定义是在预处理阶段展开，做字符串的替换；而const常量是在编译时。

2）类型和安全检查不同。宏定义没有类型，不做任何类型检查；const常量有具体的类型，在编译期会执行类型检查。

在使用中，应尽量以const替换宏定义，可以减小犯错误的概率。

2. const关键字修饰指针

```
int number1 = 10;
int number2 = 20;
const int * p1 = &number1;//常量指针，Pointer to const
*p1 = 100;//error    通过p1指针无法修改其所指内容的值
p1 = &numbers;//ok   可以改变p1指针的指向

int const * p2 = &number1; //常量指针的第二种写法

int * const p3 = &number1;//指针常量，const pointer
*p3 = 100;//ok           通过p3指针可以修改其所指内容的值
p3 = &number2;//error    不可以改变p1指针的指向

const int * const p4 = &number1;//两者皆不能进行修改
```

3. const关键字修饰成员函数

4. const关键字修饰对象

三、new/delete表达式

在C中用来开辟和回收堆空间的方式是采用malloc/free库函数，在C++中提供了新的开辟和回收堆空间的方式，即采用new/delete表达式。

1. 开辟一个元素的空间

```
int *p = new int(1);
cout << *p << endl;
delete p;
```

2. 开辟一个数组的空间

```
int *p = new int[10]();//开辟数组时，要记得采用[]
for(int idx = 0; idx != 10; ++idx)
{
    p[idx] = idx;
}
delete []p;//回收时，也要采用[]
```

常考题：new/delete表达式与malloc/free的区别是？

1）malloc/free是C/C++语言的标准库函数，new/delete是C++的运算符或表达式；

2）new能够自动分配空间大小，malloc需要传入参数；

3）new开辟空间的同时还对空间做了初始化的操作，而malloc不行；

4）new/delete能对对象进行构造和析构函数的调用，进而对内存进行更加详细的工作，而malloc/free不能。

既然new/delete的功能完全覆盖了malloc/free，为什么C++还保留malloc/free呢？因为C++程序经常要调用C函数，而C程序只能用malloc/free管理动态内存。

四、引用

1. 什么是引用?

在理解引用概念前，先回顾一下变量名。 变量名实质就是一段连续内存空间的别名。那一段连续的内存空间只能取一个别名吗？ 显然不是，引用的概念油然而生。在C++中，引用是一个已定义变量的别名。其语法是：

```
类型 &引用名 = 目标变量名;

void test0()
{
    int a = 1;
    int &ref1 = a;
    int &ref2;
}
```

在使用引用的过程中，要注意以下几点：

1. &在这里不再是取地址符号，而是引用符号，相当于&有了第二种用法
2. 引用的类型必须和其绑定的变量的类型相同
3. 声明引用的同时，必须对引用进行初始化；否则编译时报错
4. 一旦绑定到某个变量之后，就不会再改变其指向

2. 引用的本质

C++中的引用本质上是一种被限制的指针。类似于线性表和栈的关系，栈是被限制的线性表，底层实现相同，只不过逻辑上的用法不同而已。

由于引用是被限制的指针，所以引用是占据内存的，占据的大小就是一个指针的大小。有很多的说法，都说引用不会占据存储空间，其只是一个变量的别名，但这种说法并不准确。引用变量会占据存储空间，存放的是一个地址，但是编译器阻止对它本身的任何访问，从一而终总是指向初始的目标单元。在汇编里， 引用的本质就是“间接寻址”。在后面学习了类之后，我们也能看到相关的用法。

3. 引用作为函数参数

在没有引用之前，如果我们想通过形参改变实参的值，只有使用指针才能到达目的。但使用指针的过程中，不好操作，很容易犯错。 而引用既然可以作为其他变量的别人而存在，那在很多场合下就可以用引用代替指针，因而也具有更好的可读性和实用性。这就是引用存在的意义。

一个经典的例子就是交换两个变量的值。

```
//用指针作为参数
void swap(int *pa, int *pb)
{
    int temp = *pa;
    *pa = *pb;
    *pb = temp;
}
//引用作为参数
void swap(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

参数传递的方式除了上面的指针传递和引用传递两种外，还有值传递。采用值传递时，系统会在内存中开辟空间用来存储形参变量，并将实参变量的值拷贝给形参变量，即形参变量只是实参变量的副本而已；如果函数传递的是类对象，系统还会调用类中的拷贝构造函数来构造形参对象，假如对象占据的存储空间比较大，那就很不划算了。这种情况下，强烈建议使用引用作为函数的形参，这样会大大提高函数的时空效率。

当用引用作为函数的参数时，其效果和用指针作为函数参数的效果相当。当调用函数时，函数中的形参就会被当成实参变量或对象的一个别名来使用，也就是说此时函数中对形参的各种操作实际上是对实参本身进行操作，而非简单的将实参变量或对象的值拷贝给形参。

使用指针作为函数的形参虽然达到的效果和使用引用一样，但当调用函数时仍需要为形参指针变量在内存中分配空间，而引用则不需要这样，故在C++中推荐使用引用而非指针作为函数的参数

4. 引用作为函数的返回值

```
//语法：
类型 &函数名(形参列表)
{
    //函数体
}
```

当以引用作为函数的返回值时，返回的变量其生命周期一定是要大于函数的生命周期的，即当函数执行完毕时，返回的变量还存在。

```
int gNumber;//全局变量

int func1() // 当函数返回时，会对temp进行复制
{
    temp = 100;
    return temp;//此处会进行复制操作
}

int &func2()//当函数返回时，不会对temp进行复制，因为返回的是引用
{
    temp = 1000;
    return temp;
}
```

当引用作为函数的返回值时，必须遵守以下规则：

1. 不能返回局部变量的引用。主要原因是局部变量会在函数返回后被销毁，因此被返回的引用就成为了"无所指"的引用，程序会进入未知状态。
2. 不能在函数内部返回new分配的堆空间变量的引用。如果返回的引用只是作为一个临时变量出现，而没有被赋予一个实际的变量，那么该引用所在的空间就无法释放，会造成内存泄漏。

```
int &func3()
{
    int number = 1;
    return number;
}

int &func4()
{
    int *pInt = new int(1);
    return *pInt;
}

void test()
{
    int a = 3, b = 4;
    int c = a + func4() + b;//内存泄漏
}
```

引用总结：

1. 在引用的使用中，单纯给某个变量取个别名是毫无意义的，引用的目的主要用于在函数参数传递中，解决大块数据或对象的传递效率和空间不如意的问题。
2. 用引用传递函数的参数，能保证参数传递中不产生副本，提高传递的效率，且通过const的使用，保证了引用传递的安全性。
3. 引用与指针的区别是，指针通过某个指针变量指向一个变量后，对它所指向的变量间接操作。程序中使用指针，程序的可读性差；而引用本身就是目标变量的别名，对引用的操作就是对目标变量的操作。

五、C++强制转换

类型转换有c风格的，当然还有c++风格的。c风格的转换的格式很简单

```
TYPE a = （TYPE）EXPRESSION;
```

但是c风格的类型转换有不少的缺点，有的时候用c风格的转换是不合适的，因为它可以在任意类型之间转换，比如你可以把一个指向const对象的指针转换成指向非const对象的指针，把一个指向基类对象的指针转换成指向一个派生类对象的指针，这两种转换之间的差别是巨大的，但是传统的c语言风格的类型转换没有区分这些。

另一个缺点就是，c风格的转换不容易查找，它由一个括号加上一个标识符组成，而这样的东西在c++程序里一大堆。

c++为了克服这些缺点，引进了4个新的类型转换操作符，他们是static_cast，const_cast，dynamic_cast，reinterpret_cast

1. static_cast

最常用的类型转换符，在正常状况下的类型转换, 用于将一种数据类型转换成另一种数据类型，如把int转换为float

```
int iNumber = 100;
float fNumber = 0;
fNumber = (float) iNumber; //C风格
fNumber = static_cast<float>(iNumber);
```

也可以完成指针之间的转换，例如可以将void*指针转换成其他类型的指针

```
void *pVoid = malloc(sizeof(int));
int *pInt = static_cast<int*>(pVoid);
*pInt = 1;
```

但不能完成任意两个指针类型间的转换

```
int iNumber = 1;
int *pInt = &iNumber;
float *pFloat = static_cast<float *>(pInt);//error
```

总结，static_cast的用法主要有以下几种：

1）用于基本数据类型之间的转换，如把int转成char，把int转成enum。这种转换的安全性也要开发人员来保证。

2）把void指针转换成目标类型的指针，但不安全。

3）把任何类型的表达式转换成void类型。

4）用于类层次结构中基类和派生类之间指针或引用的转换。进行上行转换（把派生类的指针或引用转换成基类指针或引用）是安全的；进行下行转换（把基类指针或引用转换成派生类指针或引用）时，由于没有动态类型检查，所以是不安全的。

2. const_cast

该运算符用来修改类型的const属性。常量指针被转化成非常量指针，并且仍然指向原来的对象；常量引用被转换成非常量引用，并且仍然指向原来的对象；常量对象被转换成非常量对象。

```
const int number = 100;
int *pInt = &number;//error
int *pInt2 = const_cast<int *>(&number);
```

3. dynamic_cast

该运算符主要用于基类和派生类间的转换，尤其是向下转型的用法中。

4. reinterpret_cast

该运算符可以用来处理无关类型之间的转换，即用在任意指针（或引用）类型之间的转换，以及指针与足够大的整数类型之间的转换。由此可以看出，reinterpret_cast的效果很强大，但错误的使用reinterpret_cast很容易导致程序的不安全，只有将转换后的类型值转换回到其原始类型，这样才是正确使用reinterpret_cast方式。

六、函数重载

在实际开发中，有时候需要实现几个功能类似的函数，只是细节有所不同。如交换两个变量的值，但这两种变量可以有多种类型，short, int, float等。在C语言中，必须要设计出不同名的函数，其原型类似于：

```
void swap1(short *, short *);
void swap2(int *, int *);
void swap3(float *, float *);
```

但在C++中，这完全没有必要。C++ 允许多个函数拥有相同的名字，只要它们的参数列表不同就可以，这就是函数重载（Function Overloading）。借助重载，一个函数名可以有多种用途。

函数重载是指在同一作用域内，可以有一组具有相同函数名，不同参数列表的函数，这组函数被称为重载函数。重载函数通常用来命名一组功能相似的函数，这样做减少了函数名的数量，避免了名字空间的污染，对于程序的可读性有很大的好处。

C++进行函数重载的实现原理叫名字改编（name mangling），具体的规则是：

1. 函数名称必须相同 。
2. 参数列表必须不同（参数的类型不同、个数不同、顺序不同）。
3. 函数的返回类型可以相同也可以不相同。
4. 仅仅返回类型不同不足以成为函数的重载。

七、默认参数

1. 默认参数的目的

C++可以给函数定义默认参数值。通常，调用函数时，要为函数的每个参数给定对应的实参。

```
void func1(int x, int y);
void func1(int x, int y)
{
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
}
```

无论何时调用func1函数，都必须要给其传递两个参数。但C++可以给参数定义默认值，如果将func1函数参数中的x定义成默认值0， y定义成默认值0，只需简单的将函数声明改成

```
void func1(int x = 0, int y = 0);
```

这样调用时，若不给参数传递实参，则func1函数会按指定的默认值进行工作。允许函数设置默认参数值，是为了让编程简单，让编译器做更多的检查错误工作。

2. 默认参数的声明

一般默认参数在函数声明中提供。当一个函数既有声明又有定义时，只需要在其中一个中设置默认值即可。若在定义时而不是在声明时置默认值，那么函数定义一定要在函数的调用之前。因为声明时已经给编译器一个该函数的向导，所以只在定义时设默认值时，编译器只有检查到定义时才知道函数使用了默认值。若先调用后定义，在调用时编译器并不知道哪个参数设了默认值。所以我们通常是将默认值的设置放在声明中而不是定义中。

3. 默认参数的顺序规定

如果一个函数中有多个默认参数，则形参分布中，默认参数应从右至左逐渐定义。当调用函数时，只能向左匹配参数。如：

```
void func2(int a = 1, int b, int c = 0, int d);//error
void func2(int a, int b, int c = 0, int d = 0);//ok
```

若给某一参数设置了默认值，那么在参数表中其后所有的参数都必须也设置默认值，否则，由于函数调用时可不列出已设置默认值的参数，编译器无法判断在调用时是否有参数遗漏。

4. 默认参数与函数重载

默认参数可将一系列简单的重载函数合成为一个。例如：

```
void func3();
void func3(int x);
void func3(int x, int y);
//上面三个函数可以合成下面这一个
void func3(int x = 0, int y = 0);
```

如果一组重载函数（可能带有默认参数）都允许相同实参个数的调用，将会引起调用的二义性。

```
void func4(int);
void func4(int x, int y = 0);
void func4(int x = 0, int y = 0);
```

所以在函数重载时，要谨慎使用默认参数。

八、bool类型

在C++中，还添加了一种基本类型，就是bool类型，用来表示true和false。true和false是字面值，可以通过转换变为int类型，true为1，false为0.

```
int x = true;// 1
int y = false;// 0
```

任何数字或指针值都可以隐式转换为bool值。

任何非零值都将转换为true，而零值转换为false.

```
bool b1 = -100;
bool b2 = 100;
bool b3 = 0;
bool b4 = 1;
bool b5 = true;
bool b6 = false;
int x = sizeof(bool);
```

一个bool类型的数据占据的内存空间大小为1.

九、inline函数

在C++中，通常定义以下函数来求取两个整数的最大值

```
int max(int x, int y)
{
    return x > y ? x : y;
}
```

为这么一个小的操作定义一个函数的好处有：

- [ ] 阅读和理解函数 max 的调用，要比读一条等价的条件表达式并解释它的含义要容易得多;
- [ ] 如果需要做任何修改，修改函数要比找出并修改每一处等价表达式容易得多;
- [ ] 使用函数可以确保统一的行为，每个测试都保证以相同的方式实现;
- [ ] 函数可以重用，不必为其他应用程序重写代码。

虽然有这么多好处，但是写成函数有一个潜在的缺点：调用函数比求解等价表达式要慢得多。在大多数的机器上，调用函数都要做很多工作：调用前要先保存寄存器，并在返回时恢复，复制实参，程序还必须转向一个新位置执行。即对于这种简短的语句使用函数开销太大。

在C语言中，我们使用带参数的宏定义这种借助编译器的优化技术来减少程序的执行时间，那么在C++中有没有相同的技术或者更好的实现方法呢？答案是有的，那就是内联(inline)函数。内联函数作为编译器优化手段的一种技术，在降低运行时间上非常有用。

1. 什么是内联函数?

内联函数是C++的增强特性之一，用来降低程序的运行时间。当内联函数收到编译器的指示时，即可发生内联：编译器将使用函数的定义体来替代函数调用语句，这种替代行为发生在编译阶段而非程序运行阶段。

定义函数时，在函数的最前面以关键字“inline”声明函数，即可使函数称为内联声明函数。

```
inline int max(int x, y)
{
    return x > y ? x : y;
}
```

2. 内联函数和带参数的宏定义

无论是《Effective C++》中的 “Prefer consts，enums，and inlines to #defines” 条款，还是《高质量程序设计指南——C++/C语言》中的“用函数内联取代宏”。在书《高质量程序设计指南——C++/c语言》中这样解释到：

在 C 程序中，可以用宏代码提高执行效率。宏代码本身不是函数，但使用起来像函数。编译预处理器用拷贝宏代码的方式取代函数调用，省去了参数压栈、生成汇编语言的 CALL 调用、返回参数、执行 return 等过程，从而提高了速度。使用宏代码最大的缺点是容易出错，预处理器在拷贝宏代码时常常产生意想不到的边际效应。例如：

```
#define MAX(a, b)     (a) > (b) ? (a) : (b)
```

语句：

```
result = MAX(i, j) + 2 ;
```

将被预处理器扩展为：

```
result = (i) > (j) ? (i) : (j) + 2 ;
```

由于运算符“+”比运算符“?:”的优先级高，所以上述语句并不等价于期望的：

```
result = ( (i) > (j) ? (i) : (j) ) + 2 ;
```

如果把宏代码改写为：

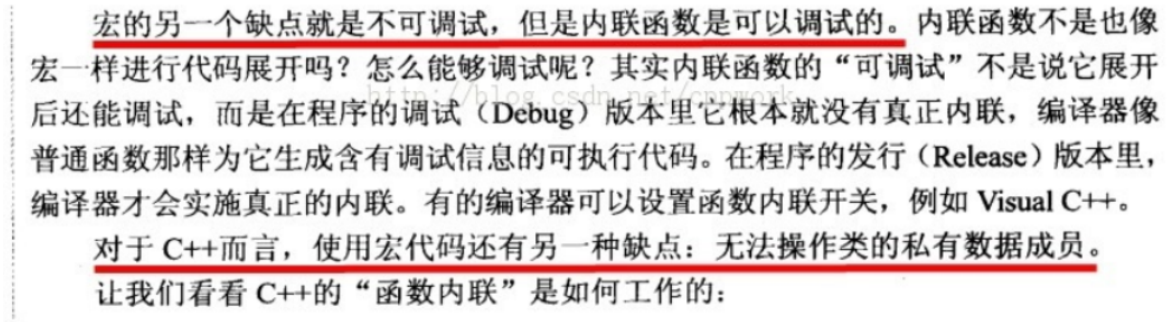
```
#define MAX(a, b)     ((a) > (b) ? (a) : (b))
```

则可以解决由优先级引起的错误。但是即使使用修改后的宏代码也不是万无一失的，例如语句：

```
result = MAX(i++, j);
```

将被预处理器解释为：

```
result = (i++) > (j) ? (i++) : (j);   // 在同一个表达式中 i 被两次求值
```

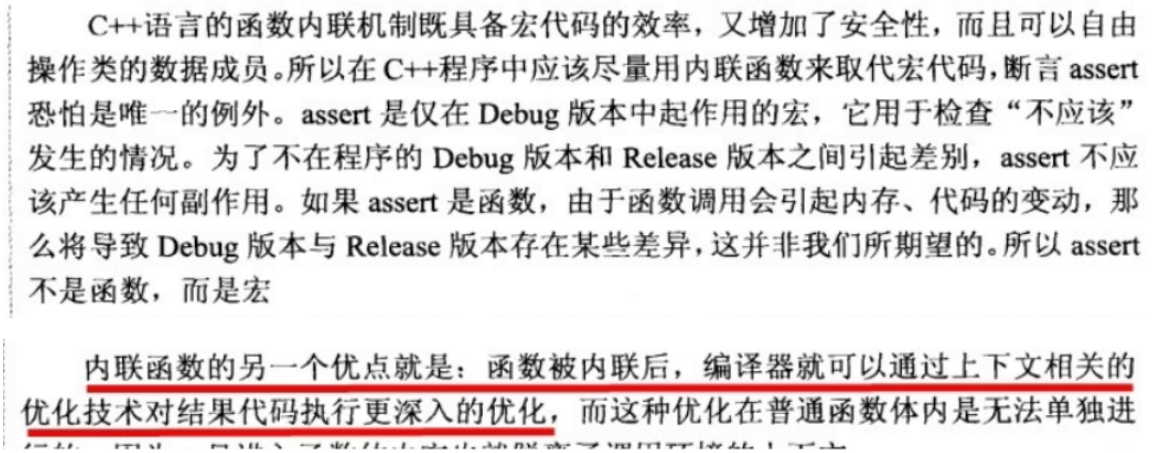


宏的另一个缺点就是不可调试，但是内联函数是可以调试的。内联函数不是也像宏一样进行代码展开吗？怎么能够调试呢？其实内联函数的“可调试”不是说它展开后还能调试，而是在程序的调试（Debug）版本里它根本就没有真正内联，编译器像普通函数那样为它生成含有调试信息的可执行代码。在程序的发行（Release）版本里，编译器才会实施真正的内联。有的编译器可以设置函数内联开关，例如 Visual C++。

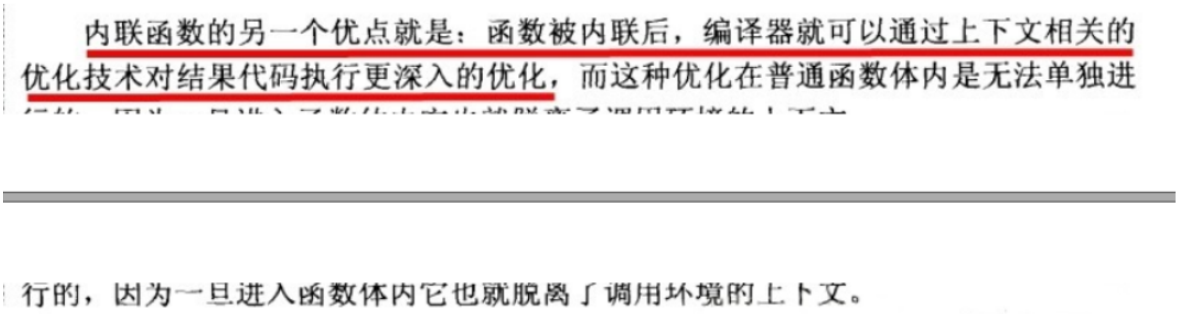
对于 C++而言，使用宏代码还有另一种缺点：无法操作类的私有数据成员。

让我们看看 C++的“函数内联”是如何工作的：

对任何内联函数，编译器在符号表里放入函数的声明，包括名字、参数类型、返回值类型（符号表是编译器用来收集和保存字面常量和某些符号常量的地方）。如果编译器没有发现内联函数存在错误，那么该函数的代码也被放入符号表里。在调用一个内联函数时，编译器首先检查调用是否正确（进行类型安全检查，或者进行自动类型转换，当然对所有的函数都一样）。如果正确，内联函数的代码就会直接替换函数调用语句，于是省去了函数调用的开销。这个过程与预处理器有显著的不同，因为预处理器不能进行类型安全检查和自动类型转换。假如内联函数是成员函数，对象的地址（this）会被放在合适的地方，这也是预处理器办不到的。



C++语言的函数内联机制既具备宏代码的效率，又增加了安全性，而且可以自由操作类的数据成员。所以在 C++程序中应该尽量用内联函数来取代宏代码，断言 assert 恐怕是唯一的例外。assert 是仅在 Debug 版本中起作用的宏，它用于检查“不应该”发生的情况。为了不在程序的 Debug 版本和 Release 版本之间引起差别，assert 不应该产生任何副作用。如果 assert 是函数，由于函数调用会引起内存、代码的变动，那么将导致 Debug 版本与 Release 版本存在某些差异，这并非我们所期望的。所以 assert 不是函数，而是宏



内联函数的另一个优点就是：函数被内联后，编译器就可以通过上下文相关的优化技术对结果代码执行更深入的优化，而这种优化在普通函数体内是无法单独进行的，因为一旦进入函数体内它也就脱离了调用环境的上下文。

3. 将内联函数放入头文件

关键字 inline 必须与函数定义体放在一起才能使函数成为内联，仅将 inline 放在函数声明前面不起任何作用。下面的foo函数不能成为内联函数：

```
1 inline void foo(int x, int y);//该语句在头文件中
2 void foo(int x, int y)//实现在.cpp文件中
3 {
4     //...
5 }
```

而正确的姿势如下：

```
inline void bar(int x, in y)//该语句在头文件中
{
    //...
}
```

所以说，C++ inline函数是一种“用于实现的关键字”，而不是一种“用于声明的关键字”。一般地，用户可以阅读函数的声明，但是看不到函数的定义。尽管在大多数教科书中内联函数的声明、定义体前面都加了 inline 关键字，但我认为 inline 不应该出现在函数的声明中。这个细节虽然不会影响函数的功能，但是体现了高质量C++/C 程序设计风格的一个基本原则：声明与定义不可混为一谈，用户没有必要、也不应该知道函数是否需要内联。

内联函数应该在头文件中定义，这一点不同于其他函数。编译器在调用点内联展开函数的代码时，必须能够找到 inline函数的定义才能将调用函数替换为函数代码，而对于在头文件中仅有函数声明是不够的。

当然内联函数定义也可以放在源文件中，但此时只有定义的那个源文件可以用它，而且必须为每个源文件拷贝一份定义(即每个源文件里的定义必须是完全相同的)，当然即使是放在头文件中，也是对每个定义做一份拷贝，只不过是编译器替你完成这种拷贝罢了。但相比于放在源文件中，放在头文件中既能够确保调用函数是定义是相同的，又能够保证在调用点能够找到函数定义从而完成内联(替换)。

4. 谨慎地使用内联

内联能提高函数的执行效率，为什么不把所有的函数都定义成内联函数?

以下摘自《高质量程序设计指南----C/C++语言》：

内联能提高函数的执行效率，为什么不把所有的函数都定义成内联函数呢？
如果所有的函数都是内联函数，还用得着“内联”这个关键字吗？

内联不是万灵丹，它以代码膨胀（拷贝）为代价，仅仅省去了函数调用的开销，从而提高程序的执行效率。注意：这里所说的“函数调用开销”并不包括执行函数体所需要的开销，而是仅指参数压栈、跳转、退栈和返回等操作。如果执行函数体内代码的时间比函数调用的开销大得多，那么 inline 的效率收益会很小。另一方面，每一处内联函数的调用都要拷贝代码，将使程序的总代码量增大，消耗更多的内存空间。

以下情况不宜使用内联：

- ✧ 如果函数体内的代码比较长，使用内联将导致可执行代码膨胀过大。
- ✧ 如果函数体内出现循环或者其他复杂的控制结构，那么执行函数体内代码的时间将比函数调用的开销大得多，因此内联的意义并不大。

不少人误以为让类的构造函数和析构函数成为内联函数更有效。殊不知，构造函数和析构函数会隐藏一些行为，如“偷偷地”调用基类或成员对象的构造函数和析构函数。所以不要轻易让构造函数和析构函数成为内联函数。

实际上，inline 在实现的时候就是对编译器的一种请求，因此编译器完全有权利取消一个函数的内联请求。一个好的编译器能够根据函数的定义体，自动取消不值得的内联（这进一步说明了 inline 不应该出现在函数的声明中），或者自动地内联一些没有 inline 请求的函数。因此，编译器往往选择那些短小而简单的函数来内联（内联候选函数）。

十、异常安全

异常是程序在执行期间产生的问题。C++ 异常是指在程序运行时发生的特殊情况，比如尝试除以零的操作。异常提供了一种转移程序控制权的方式。C++ 异常处理涉及到三个关键字：try、catch、throw。

- throw: 当问题出现时，程序会抛出一个异常。这是通过使用 throw 关键字来完成的。
- try: try 块中的代码标识将被激活的特定异常，它后面通常跟着一个或多个 catch 块。
- catch: 在您想要处理问题的地方，通过异常处理程序捕获异常。catch 关键字用于捕获异常。

throw表达式

抛出异常即检测是否产生异常，在C++中，其采用throw语句来实现，如果检测到产生异常，则抛出异常。该语句的格式为：

```
throw 表达式;
```

异常是一个表达式，其值的类型可以是基本类型，也可以是类

举个例子:

```
double division(double x, double y)
{
    if(y == 0)
        throw "Division by zero condition!";
    return x / y;
}
```

try-catch语句块

try-catch语句块的语法如下：

```
try
{
    //语句块
}
catch(异常类型)
{
    //具体的异常处理...
}
...
catch(异常类型)
{
    //具体的异常处理...
}
```

try...catch语句块的catch可以有多个，但至少要有一个。

try...catch语句的执行过程是：

- 执行 try块中的语句，如果执行的过程中没有异常抛出，那么执行完后就执行最后一个 catch块后面的语句，所有 catch块中的语句都不会被执行；
- 如果 try块执行的过程中抛出了异常，那么抛出异常后立即跳转到第一个“异常类型”和抛出的异常类型匹配的 catch块中执行（称作异常被该 catch块“捕获”），执行完后再跳转到最后一个 catch块后面继续执行。

举个例子:

```
void test()
```

```
{
    double x, y;
    cin >> x >> y;
    try
    {
        if(0 == y)
        {
            throw y;
        }
        else
        {
            cout << (x / y) << endl;
        }
    }
    catch(double d)
    {
        cout << "catch(double)" << endl;
    }
    catch(int e)
    {
        cout << "catch(int)" << endl;
    }
}
```