

0.本节知识点安排目的

C++是在C的基础之上，容纳进去了面向对象编程思想，并增加了许多有用的库，以及编程范式等。熟悉C语言之后，对C++学习有一定的帮助，本章节主要目标：

- 1. 补充C语言语法的不足，以及C++是如何对C语言设计不合理的地方进行优化的，比如：作用域方面、IO方面、函数方面、指针方面、宏方面等。
- 2. 为后续类和对象学习打基础。

1.C++关键字（C++98）

C++总计63个关键字，C语言32个关键字

ps：下面我们只是看一下C++有多少关键字，不对关键字进行具体的说明，后期在进行说明

asm	do	if	return	try	continue
auto	double	inline	short	typedef	for
bool	dynamic_cast	int	signed	typeid	public
break	else	long	sizeof	typename	throw
case	enum	mutable	static	union	wchar_t
catch	explicit	namespace	static_cast	unsigned	default
char	export	new	struct	using	friend
class	extern	operator	switch	virtual	register
const	false	private	template	void	true
const_cast	float	protected	this	volatile	while
delete	goto	reinterpret_cast			

2.命名空间

在C/C++中，变量、函数和后面要学到的类都是大量存在的，这些变量、函数和类的名称将都存在于全局作用域中，可能会导致很多冲突。使用命名空间的目的是对标识符的名称进行本地化，以避免命名冲突或名字污染，namespace关键字的出现就是针对这种问题的。

例如下面的例子：

```
#include <stdio.h>
#include <stdlib.h>
```

```
int rand = 10;
```



Redefinition of 'rand' as different kind of symbol

//C语言没办法解决类似这样的命名冲突问题，所以C++提出了namespace来解决

2.1 命名空间定义

定义命名空间，需要使用到namespace关键字，后面跟命名空间的名字，然后接一对{}即可，{}中即为命名空间的成员。

//1. 正常的命名空间定义

```
namespace bit{//bit是命名空间的名字，一般开发中使用项目名字做命名空间名。
    //命名空间中可以定义变量/函数/类型
    int rand = 10;

    int Add(int left,int right){
        return left + right;
    }

    struct Node{
        struct Node* next;
        int val;
    };
}
```

//2. 命名空间可以嵌套

```
namespace N1{
    int a,b;
    int Add(int left,int right){
        return left + right;
    }

    namespace N2{
        int c,d;
        int Sub(int left,int right){
            return left - right;
        }
    }
}
```

//3. 同一个工程中允许存在多个相同名称的命名空间，编译器最后会合并成同一个命名空间中

```
namespace N1{
int MUL(int left,int right){
    return left* right;
}
}
```

注意：一个命名空间就定义了一个新的作用域，命名空间中的所有内容都局限于该命名空间中

2.2 命名空间的使用

命名空间的使用有三种方式

1 加命名空间名称及作用域限定符

```
int main(int argc, const char * argv[])
{
    printf("%d\n", bit::rand);
    return 0;
}
```

2 使用using将命名空间中某个成员引入

```
int main(int argc, const char * argv[])
{
    using bit::rand;
    printf("%d\n", rand);
    return 0;
}
```

3 使用using namespace命名空间名称 引入

```
int main(int argc, const char * argv[])
{
    using namespace N1;
    printf("%d\n", Add(2, 5));
    return 0;
}
```

2.3 展开命名空间查找顺序

编译默认查找

- a、当前局部域
- b、全局域找
- c、到展开的命名空间中查找
- d、不同域可以定义同名的变量/函数/类型

3.C++输入&输出

```
#include <iostream>
```

//std是C++标准库的命名空间名，C++将标准库的定义实现都放到这个命名空间中

```
using namespace std;
```

```
int main(int argc, const char * argv[]){
```

```
    cout<<"hello world!"<<endl;
```

```
    return 0;
```

```
}
```

说明：

1. 使用cout标准输出对象(控制台)和cin标准输入对象(键盘)时，必须包含头文件以及按命名空间使用方法使用std。
2. cout和cin是全局的流对象，endl是特殊的C++符号，表示换行输出，他们都包含在包含头文件中。
3. <<是流插入运算符，>>是流提取运算符。

4. 使用C++输入输出更方便，不需要像printf/scanf输入输出时那样，需要手动控制格式。C++的输入输出可以自动识别变量类型。
5. 实际上cout和cin分别是ostream和istream类型的对象，>>和<<也涉及运算符重载等知识，这些知识我们后续才会学习，所以我们这里只是简单学习他们的使用。后面我们还有一个章节更深入的学习IO流用法及原理。

注意：早期标准库将所有功能在全局域中实现，声明在.h后缀的头文件中，使用时只需包含对应头文件即可，后来将其实现在std命名空间下，为了和C头文件区分，也为了正确使用命名空间，规定C++头文件不带.h，旧编译器(VC 6.0)中还支持<iostream.h>格式，后续编译器已不支持，因此推荐使用+std的方式。

```
#include <iostream>
//std是C++标准库的命名空间名，C++将标准库的定义实现都放到这个命名空间中
using namespace std;
//int main(int argc, const char * argv[]){
//    cout<<"hello world!"<<endl;
//    return 0;
//}

int main(int argc, const char * argv[]){
    int a; double b; char c;

    //可以自动识别变量的类型
    cin>>a;
    cin>>b>>c;

    cout<<a<<endl;
    cout<<b<<" "<<c<<endl;
    return 0;
}
```

//ps：关于cout和cin还有很多更复杂的用法，比如控制浮点数输出精度，控制整形输出进制格式等等。因为C++兼容C语言的用法，这些又用得不是很多，我们这里就不展开学习了。后续如果有需要，我们再配合文档学习。

```
2  4 c
2
4 c
```

std命名空间的使用惯例：

std是C++标准库的命名空间，如何展开std使用更合理呢？

1. 在日常练习中，建议直接using namespace std即可，这样就很方便。
2. using namespace std展开，标准库就全部暴露出来了，如果我们定义跟库重名的类型对象/函数，就存在冲突问题。该问题在日常练习中很少出现，但是项目开发中代码较多、规模大，就容易出现。所以建议在项目开发中使用，像std::cout这样使用时指定命名空间+using std::cout展开常用的库对象/类型等方式。

4.缺省参数

4.1 缺省参数概念

缺省参数是声明或定义函数时为函数的参数指定一个缺省值。在调用该函数时，如果没有指定实参，则采用该形参的缺省值，否则使用指定的实参。

```
void Func(int a = 0){
    cout<<a<<endl;
}
int main(int argc, const char * argv[]){
    Func();           //没有传参时，使用参数的默认值
    Func(10);         //传参时，使用指定的实参

    return 0;
}
```

4.2 缺省参数分类

· 全缺省参数

```
void Func(int a = 10, int b = 20, int c = 30){
    cout<<"a = "<<a<<endl;
    cout<<"b = "<<b<<endl;
    cout<<"c = "<<c<<endl;
}
```

· 半缺省参数

```
void Func(int a, int b = 10, int c = 20)
{
    cout<<"a = "<<a<<endl;
    cout<<"b = "<<b<<endl;
    cout<<"c = "<<c<<endl;
}
```

注意：

1. 半缺省参数必须从右往左依次来给出，不能间隔着给
2. 缺省参数不能在函数声明和定义中同时出现

```
//a.h
void Func(int a = 10);

// a.cpp
void Func(int a = 20)
{}
```

// 注意：如果生命与定义位置同时出现，恰巧两个位置提供的值不同，那编译器就无法确定到底该用那个缺省值。

3.缺省值必须是常量或者全局变量

4.C语言不支持（编译器不支持）

5.函数重载

自然语言中，一个词可以有多重含义，人们可以通过上下文来判断该词真实的含义，即该词被重载了。

比如：以前有一个笑话，国有两个体育项目大家根本不用看，也不用担心。一个是乒乓球，一个是男足。前者是“谁也赢不了！”，后者是“谁也赢不了！”

5.1 函数重载概念

函数重载：是函数的一种特殊情况，C++允许在同一作用域中声明几个功能类似的同名函数，这些同名函数的形参列表（参数个数 或 类型 或类型顺序）不同，常用来处理实现功能类似数据类型不同的问题

```
//1.参数类型不同
int Add(int left, int right){
    cout<<"int Add(int left, int right)"<<endl;
    return left + right;
}
double Add(double left, double right){
    cout<<"double Add(double left, double right)"<<endl;
    return left + right;
}
//2.参数个数不同
void f(){
    cout<<"f()"<<endl;
}
void f(int a){
    cout<<"f(int a)"<<endl;
}
//3.参数类型顺序不同
void f(int a, char b){
    cout<<"f(int a, char b)"<<endl;
}
void f(char a, int b){
    cout<<"f(char a, int b)"<<endl;
}
int main(int argc, const char * argv[]){
    Add(20, 30);
}
```

```

Add(1.0, 2.3);
f();
f(5);
f(10, 'a');
f('a', 10);
return 0;
}

```

Line 130 Col 1

```

int Add(int left, int right)
double Add(double left, double right)
f()
f(int a)
f(int a, char b)
f(char a, int b)
Program ended with exit code: 0

```

6.引用

6.1 引用概念

引用不是新定义一个变量，而是给已存在变量取了一个别名，编译器不会为引用变量开辟内存空间，它和它引用的变量共用同一块内存空间。

语法格式：类型& 引用变量名（对象名）= 引用实体；

例如：

```

void TestRef(){
    int a = 10;
    int& ra = a; //<====定义引用类型
    cout<<"a = "<<a<<endl;
    cout<<"ra = "<<ra<<endl;
}
int main(int argc, const char * argv[]){
    TestRef();
    return 0;
}

```

```

a = 10
ra = 10
Program ended with exit code: 0

```

注意：引用类型必须和引用实体是同种类型的；

6.2 引用特性

- 1.引用在定义时必须初始化
- 2.一个变量可以有多个引用
- 3.引用一旦引用一个实体，再不能引用其它实体

```
void TestRef1(){
```

```
    int a = 10;
```

```
    int& ra; //未初始化
```

```
}
```



Declaration of reference variable 'ra' requires an initializer

```
void TestRef1(){
```

```
    int a = 10;
```

```
    //int& ra; //未初始化
```

```
    int& ra = a;
```

```
    int& rra = ra;
```

```
    cout<<&a<<endl<<&ra<<endl<<&rra<<endl;
```

```
}
```

0x16fdfec6c

0x16fdfec6c

0x16fdfec6c

Program ended with exit code: 0

6.3 常引用

```
void TestConstRef()
```

```
{
```

```
    const int a = 10;
```

```
    //int& ra = a;    // 该语句编译时会出错，a为常量
```

```
    const int& ra = a;
```

```
    // int& b = 10;    // 该语句编译时会出错，b为常量
```

```
    const int& b = 10;
```

```
    double d = 12.34;
```

```
    //int& rd = d;    // 该语句编译时会出错，类型不同
```

```
    const int& rd = d;
```

```
}
```

```
int main(int argc, const char * argv){
```

```
    int x = 10;
```

```
    int& y = x;
```

```

//权限的缩小, 可以
const int& z =x;
//z++; 不可以
y++;
// 权限的方法
// m只读
// n变成我的别名, n的权限是可读可写
// 权限的放大, 不可以
const int m = 0;
//int& n = m;不可以
const int &n = m;//可以
int p = m;//可以 m拷贝的给p, p的修改不影响m
//权限的放大不可以
const int* p1 = &m;
p1++;//可以 而*p1不可以 const修饰的是*p1
//int *p2 = p1; 不可以
//权限的缩小
int* p3 = &x;
const int* p4 = p3;
return 0;
}

```

6.3 常引用

```

void TestConstRef(){
    const int a =10;
    //int& ra = a; 该语句编译时会出错, a是常量
    const int& ra = a;//权限的平移
    //int& b = 10; 该语句编译时会出错, b是常量
    const int& b = 10;
    double d = 12.34;
    //int& rd = d; 该语句编译时会出错, 类型不同
    const int& rd = d;//权限的缩小
}

```

6.4 使用场景

1. 做参数

```

void Swap(int &x,int & y){
    int tmp = x;
    x = y;
    y = tmp;
}

int main(int argc, const char * argv[]){
    int a = 10,b = 5;
    Swap(a, b);
    cout<<"a= " <<a<<endl<<"b= " <<b<<endl;
}

```

```
a= 5
b= 10
Program ended with exit code: 0
```

1. 做返回值

```
int& Count(){
    static int n = 0;
    n++;
    //...
    return n;
}
```

//...待补充

6.5 传值、传引用效率比较

以值作为参数或者返回值类型，在传参和返回期间，函数不会直接传递实参或者将变量本身直接返回，而是传递实参或者返回变量的一份临时的拷贝（且临时对象具有常性），因此用值作为参数或者返回值类型，效率是非常低下的，尤其是当参数或者返回值类型非常大时，效率就更低。

```
struct A { int a[10000]; };
void TestFunc1(A a) {}
void TestFunc2(A& a) {}
void TestRefAndValue()
{
    A a;
    // 以值作为函数参数
    size_t begin1 = clock();
    for (size_t i = 0; i < 10000; ++i)
        TestFunc1(a);
    size_t end1 = clock();
    // 以引用作为函数参数
    size_t begin2 = clock();
    for (size_t i = 0; i < 10000; ++i)
        TestFunc2(a);
    size_t end2 = clock();
    // 分别计算两个函数运行结束后的时间
    cout << "TestFunc1(A)-time:" << end1 - begin1 << endl;
    cout << "TestFunc2(A&)-time:" << end2 - begin2 << endl;
}
int main(int argc, const char * argv[])
{
    TestRefAndValue();
    return 0;
}
```

```
}
```

```
TestFunc1(A)-time:6519
TestFunc2(A&)-time:22
Program ended with exit code: 0
```

6.5.2 值和引用作为返回值类型的性能比较

```
struct A {
    int a[10000];
};
A TestFunc1(A a) {
    return a;
}
A& TestFunc2(A& a) {
    return a;
}
void TestRefAndValue()
{
    A a;
    // 以值作为函数参数
    size_t begin1 = clock();
    for (size_t i = 0; i < 10000; ++i)
        TestFunc1(a);
    size_t end1 = clock();
    // 以引用作为函数参数
    size_t begin2 = clock();
    for (size_t i = 0; i < 10000; ++i)
        TestFunc2(a);
    size_t end2 = clock();
    // 分别计算两个函数运行结束后的时间
    cout << "TestFunc1-time:" << end1 - begin1 << endl;
    cout << "TestFunc2-time:" << end2 - begin2 << endl;
}
int main(int argc, const char * argv[])
{
    TestRefAndValue();
    return 0;
}
```

```
TestFunc1-time:11283
TestFunc2-time:21
Program ended with exit code: 0
```

通过上述代码比较，发现传值和指针在作为传参以及返回值类型上效率相差很大。

6.6 引用和指针的区别

在语法概念上，引用就是一个别名，没有独立空间，和其引用实体共用同一块空间
在底层实现上实际是有空间的，因为引用是按照指针方式来实现的；

```
int a = 0;
009D1D8F  mov             dword ptr [a],0
int& b = a; // 语法上不开空间
009D1D96  lea             eax, [a]
009D1D99  mov             dword ptr [b],eax

int* p = &a; // 语法上要开空间
009D1D9C  lea             eax, [a]
009D1D9F  mov             dword ptr [p],eax
```

引用和指针的不同点：

1. 引用概念上定义一个变量的别名，指针存储一个变量地址。
2. 引用在定义时必须初始化，指针没有要求
3. 引用在初始化时引用一个实体后，就不能再引用其他实体，而指针可以在任何时候指向任何一个同类型实体；
4. 没有NULL引用。但有空指针
5. 在sizeof中含义不同：引用结果为引用类型的大小，但指针始终是地址空间所占字节个数（32位平台下占4个字节）
6. 引用自加即引用的实体加1，指针自加即指针向后偏移一个类型的大小
7. 有多级指针，但是没有多级引用
8. 访问实体方式不同，指针需要显示解引用，引用编译器自己处理
9. 引用比指针使用起来相对更安全

7.内联函数

7.1 概念

以inline修饰的函数叫做内联函数，编译时C++编译器会在调用内联函数的地方展开，没有函数调用建立栈帧的开销，内联函数提升程序运行的效率。

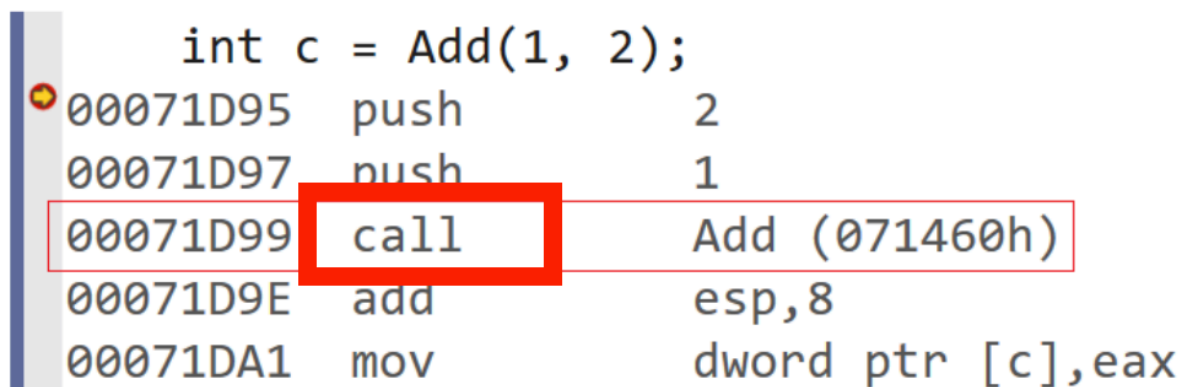
257

```
258 inline int& Add (int a, int b){
```

```
259     int c = a + b;
```

```
260     return c;  Reference to stack memory as
```

```
261 }
```

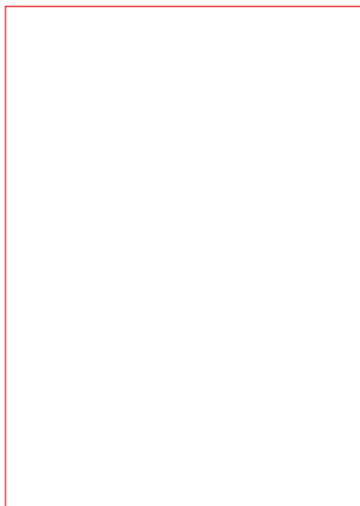


```
int c = Add(1, 2);  
00071D95 push 2  
00071D97 push 1  
00071D99 call Add (071460h)  
00071D9E add esp, 8  
00071DA1 mov dword ptr [c], eax
```

7.2 特性

1. inline是一种以空间换时间的做法，如果编译器将函数当成内联函数处理，在编译阶段，会用函数体替换函数调用，缺陷：可能会使目标文件变大，优势：少了调用开销，提高程序运行效率。
2. inline对于编译器而言只是一个建议，不同编译器关于inline实现机制可能不同，一般建议：将函数规模较小(即函数不是很长，具体没有准确的说法，取决于编译器内部实现、不是递归、且频繁调用的函数采用inline修饰，否则编译器会忽略inline特性。一般来说，内联机制用于优化规模较小、流程直接、频繁调用的函数。很多编译器都不支持内联递归函数，而且一个75行的函数也不大可能在调用点内联地展开
3. inline不建议声明和定义分离，分离会导致链接错误。因为inline被展开，就没有函数地址了，链接就会找不到。

100行 指令



10000个位置调用

展开: 10000*100

不展开: 10000+100



展开: 编译出来的可执行的程序变大

不展开10110次原因：10000次调用+100call

面试题-宏的优缺点

优点：1. 增加代码的复用性。2. 提高性能

缺点：

1. 不方便调试宏（因为预编译阶段进行了替换）
2. 导致代码可读性差，可维护性差，容易误用。
3. 没有类型安全的检查

C++有哪些技术替代宏？

1. 常量定义 换用const enum
2. 短小函数定义 换用内联函数

8.auto关键字（C++11）

8.1 类型别名思考

随着程序越来越复杂，程序中用到的类型也越来越复杂，经常体现在：

1. 类型难于拼写
2. 含义不明确导致容易出错

```
int main()
{
    int j = 0;

    // 右边初始化自动推导类型
    auto i = 0;
    std::map<std::string, std::string> dict;

    // 等价的，auto可以替代写起来比较长的类型的定义，简化代码
    //std::map<std::string, std::string>::iterator it = dict.begin();
    auto it = dict.begin();

    return 0;
}
```

当然，使用typedef给类型取别名可以简化代码，但是typedef会遇到新的难题

```
typedef char* pstring;

int main()
{
    const pstring p1;    // 编译成功还是失败?
    const pstring* p2;   // 编译成功还是失败?
    return 0;
}
```

在编程时，常常需要把表达式的值赋值给变量，这就要求在声明变量的时候清楚地知道表达式的类型。然而有时候要做到这点并非那么容易，因此C++11给auto赋予了新的含义。

8.2 auto简介

在早期C/C++中auto的含义是：使用auto修饰的变量，是具有自动存储器的局部变量，但遗憾的是一直没有人去使用它，大家可思考下为什么？（因为C语言不使用auto也可以达到自动销毁的状态）

C++11中，标准委员会赋予了auto全新的含义即：auto不再是一个存储类型指示符，而是作为一个新的类型指示符来指示编译器，auto声明的变量必须由编译器在编译时期推导而得。

```
int main(int argc, const char * argv[])
{
    int j = 0;
    // 右边初始化自动推导类型
    auto i = 0;
    std::map<std::string, std::string> dict;
    // 等价的，auto可以替代写起来比较长的类型的定义，简化代码
    std::map<std::string, std::string>::iterator it = dict.begin();
    auto it = dict.begin();
    return 0;
}
```

【注意】

使用auto定义变量时必须对其进行初始化，在编译阶段编译器需要根据初始化表达式来推导auto的实际类型。因此auto并非是一种"类型"的声明，而是一个类型声明时的"占位符"，编译器在编译期会将auto替换为变量实际的类型。

8.3 auto的使用细则

1. auto与指针和引用结合起来使用

用auto声明指针类型时，用auto和auto*没有任何区别，但用auto声明引用类型时必须加&


```
int main(int argc, const char * argv[])
{
    int x = 10;
    auto a = &x;
    auto* b = &x;
    auto& c = x;
    cout << typeid(a).name() << endl;
    cout << typeid(b).name() << endl;
    cout << typeid(c).name() << endl;
    return 0;
}
```

1. 在同一行定义多个变量

当在同一行声明多个变量时。这些变量必须是相同的类型，否则编译器将会报错，因为编译器实际只对第一个类型进行推导，然后推导出来的类型定义其他变量。

```
int main(int argc, const char * argv[])
{
    auto a = 1, b = 2;
    auto c = 3, d = 4.0; //该行代码会编译失败，因为c和d的初始化表达式类型不同
    return 0;
}
```

8.4 auto不能推导的场景

1.auto不能作为函数的参数

```
// 此处代码编译失败，auto不能作为形参类型，因为编译器无法对a的实际类型进行推导
void TestAuto(auto a)
{}
```

2.auto不能直接用来声明数组

3.为了避免与C++98中的auto发生混淆，C++11只保留了auto作为类型指示符的用法

4.auto在实际中最常见的优势用法就是跟以后会讲到的C++11提供的新式for循环，还有lambda表达式等进行配合使用。

9.基于范围的for循环（C++11）

9.1 范围for循环的语法

```
int main(int argc, const char * argv[])
{
    int array[] = { 1, 2, 3, 4, 5 };
    for (int i = 0; i < sizeof(array) / sizeof(array[0]); ++i)
    {
        array[i] *= 2;
    }
    for (int i = 0; i < sizeof(array) / sizeof(array[0]); ++i)
    {
        cout << array[i] << " ";
    }
    cout << endl;}
```

对于一个有范围的集合而言，由程序员来说明循环的范围是多余的，有时候还会容易犯错误。因此C++11中引入了基于范围的for循环。for循环后的括号由冒号“:”分为两部分：第一部分是范围内用于迭代的变量，第二部分则表示被迭代的范围。

```
// C++11 范围for
// 自动取数组array中，赋值给e
// 自动++，自动判断结束
int main(int argc, const char * argv[])
{
    int array[] = { 1, 2, 3, 4, 5 };
    for (auto& e : array)
    {
        e /= 2;
    }
    for (int x : array)
    {
        cout << x << " ";
    }
    cout << endl;
    return 0;
}
```

9.2 范围for循环使用条件

1. for循环迭代的范围必须是确定的 对于数组而言，就是数组中第一个元素和最后一个元素的范围；对于类而言，应该提供begin和end的方法，begin和end就是for循环迭代的范围。 注意：以下代码就有问题：因为for循环位置不确定

```
void TestFor(int array[])
{
    for(auto& e : array)
        cout<< e <<endl;
}
```

1. 迭代的对象要实现++和==的操作。(关于迭代器这个问题，以后会讲，现在提一下，没办法讲清楚，现在大家了解一下就可以了)

10. 指针空值nullptr(C++11)

10.1 C++98中的指针空值

在良好的C/C++编程习惯中，声明一个变量时最好给该变量一个合适的初始值，否则可能会出现不可预料的错误，比如未初始化的指针。如果一个指针没有合法的指向，我们基本都是按照如下方式对其进行初始化：

```
void TestPtr()
{
    int* p1 = NULL;
    int* p2 = 0;

    // .....
}
```

NULL实际是一个宏，在传统的C头文件(srddef.h)中，可以看到如下代码

```
#ifndef NULL
#ifdef __cplusplus
#define NULL    0
#else
#define NULL    ((void *)0)
#endif
#endif
```

可以看到，NULL可能被定义为字面常量0，或者被定义为无类型指针(void*)的常量。不论采取何种定义，在使用空值的指针时，都不可避免的会遇到一些麻烦，比如：

```
void f(int){
    cout<<"f(int)"<<endl;
}
void f(int*){
    cout<<"f(int*)"<<endl;
}
int main(int argc, const char * argv[]){
    f(0);
    f(NULL);
    f((int*)NULL);
    f(nullptr);
}
```

程序本意是想通过(NULL)调用指针版本的f(int)函数，但是由于NULL被定义成0，因此与程序的初衷相悖。

在C++98中，字面常量0既可以是一个整形数字，也可以是无类型的指针(void*)常量，但是编译器默认情况下将其看成是一个整形常量，如果要将其按照指针方式来使用，必须对其进行强转(void)。

注意：

1. 在使用nullptr表示指针空值时，不需要包含头文件，因为nullptr是C++11作为新关键字引入
2. 在C++11中，sizeof(nullptr) 与 sizeof((void*)0)所占的字节数相同。

3. 为了提高代码的健壮性，在后续表示指针空值时建议最好使用`nullptr`。