



























Lesson00---C++入门

【本节目标】

- 1. C++关键字
- 2. 命名空间
- 3. C++输入&输出
- 4. 缺省参数
- 5. 函数重载
- 6. 引用
- 7. 内联函数
- 8. auto关键字(C++11)
- 9. 基于范围的for循环(C++11)
- 10. 指针空值---nullptr(C++11)
- 11. 总结

1. C++关键字(C++98)

Aa  alligator	Bb  bear	Cc  cat	Dd  donkey	Ee  elephant	Ff  flamingo
Gg  giraffe	Hh  hippopotamus	Ii  iguana	Jj  jaguar	Kk  kangaroo	Ll  lion
Mm  macaw	Nn  newt	Oo  ostrich	Pp  pig	Qq  quail	Rr  rhinoceros
Ss  sheep	Tt  tiger	Uu  urial	Vv  vole	Ww  wolf	Xx  x-ray tetra
Yy  yak		Zz  zebra			

大学时学写代码：关键字----->语句----->实现算法逻辑

ps：下面的关键字，我们这里了解一下就可以了。因为以后会依次学。

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。

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. 普通的命名空间
namespace N1 // N1为命名空间的名称
{
    // 命名空间中的内容，既可以定义变量，也可以定义函数
    int a;
    int Add(int left, int right)
    {
        return left + right;
    }
}

//2. 命名空间可以嵌套
namespace N2
{
    int a;
    int b;

    int Add(int left, int right)
```

```

{
    return left + right;
}

namespace N3
{
    int c;
    int 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 命名空间使用

命名空间中成员该如何使用呢？比如：

```

namespace N
{
    int a = 10;
    int b = 20;
    int Add(int left, int right)
    {
        return left + right;
    }

    int Sub(int left, int right)
    {
        return left - right;
    }
}

int main()
{
    printf("%d\n", a); // 该语句编译出错，无法识别a
    return 0;
}

```

命名空间的使用有三种方式：

- 加命名空间名称及作用域限定符

```
int main()
{
    printf("%d\n", N::a);
    return 0;
}
```

- 使用using将命名空间中成员引入

```
using N::b;
int main()
{
    printf("%d\n", N::a);
    printf("%d\n", b);
    return 0;
}
```

- 使用using namespace 命名空间名称引入

```
using namespace N;
int main()
{
    printf("%d\n", N::a);
    printf("%d\n", b);
    Add(10, 20);
    return 0;
}
```

3. C++输入&输出



那C++是否也应该向这个美好的世界来声问候呢？我们来看下C++是如何来实现问候的。

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

int main()
{
    cout<<"Hello world!!!"<<endl;
    return 0;
}
```

说明:

1. 使用**cout标准输出(控制台)**和**cin标准输入(键盘)**时, 必须**包含<iostream>头文件**以及std标准命名空间。

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**<iostream>+std**的方式。

2. 使用C++输入输出更方便, 不需增加数据格式控制, 比如: 整形--%d, 字符--%c

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

int main()
{
    int a;
    double b;
    char c;

    cin>>a;
    cin>>b>>c;

    cout<<a<<endl;
    cout<<b<<" "<<c<<endl;

    return 0;
}
```

4. 缺省参数

大家知道什么是备胎吗?



C++中函数的参数也可以配备胎。

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默认值。在调用该函数时，如果没有指定实参则采用该默认值，否则使用指定的实参。

```
void TestFunc(int a = 0)
{
    cout<<a<<endl;
}

int main()
{
    TestFunc();      // 没有传参时，使用参数的默认值
    TestFunc(10);   // 传参时，使用指定的实参
}
```

4.2 缺省参数分类

- 全缺省参数

```
void TestFunc(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 TestFunc(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 TestFunc(int a = 10);
```

```
// a.c
void TestFunc(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 顺序)必须不同**，常用来处理实现功能类似数据类型不同的问题

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

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

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

int main()
{
    Add(10, 20);
    Add(10.0, 20.0);
    Add(10L, 20L);
}
```

```
    return 0;
}
```

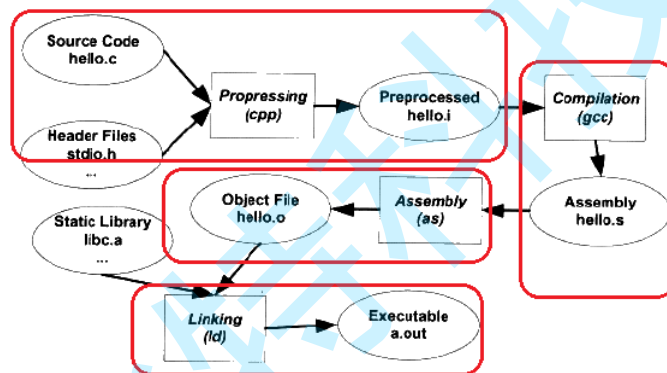
下面两个函数属于函数重载吗？

```
short Add(short left, short right)
{
    return left+right;
}

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

5.2 名字修饰(name Mangling)

在C/C++中，一个程序要运行起来，需要经历以下几个阶段：预处理、编译、汇编、链接。



Name Mangling是一种在编译过程中，将函数、变量的名称重新改编的机制，简单来说就是编译器为了区分各个函数，将函数通过某种算法，重新修饰为一个全局唯一的名称。

C语言的名字修饰规则非常简单，只是在函数名字前面添加了下划线。比如，对于以下代码，在最后链接时就会出错：

```
int Add(int left, int right);

int main()
{
    Add(1, 2);
    return 0;
}
```

编译器报错：error LNK2019: 无法解析的外部符号 Add，该符号在函数 _main 中被引用。

上述Add函数只给了声明没有给定义，因此在链接时就会报错，提示：在main函数中引用的Add函数找不到函数体。从报错结果中可以看到，C语言只是简单的在函数名前添加下划线。因此当工程中存在相同函数名的函数时，就会产生冲突。

由于C++要支持函数重载，命名空间等，使得其修饰规则比较复杂，不同编译器在底层的实现方式可能都有差异。

```
int Add(int left, int right);
double Add(double left, double right);

int main()
{
    Add(1, 2);
    Add(1.0, 2.0);
    return 0;
}
```

在vs下，对上述代码进行编译链接，最后编译器报错：

error LNK2019: 无法解析的外部符号 "double cdecl Add(double,double)" (?Add@@YANN@Z)

error LNK2019: 无法解析的外部符号 "int __cdecl Add(int,int)" (?Add@@YAH@Z)

通过上述错误可以看出，编译器实际在底层使用的不是Add名字，而是被重新修饰过的一个比较复杂的名字，被重新修饰后的名字中包含了：函数的名字以及参数类型。这就是为什么函数重载中几个同名函数要求其参数列表不同的原因。只要参数列表不同，编译器在编译时通过对函数名字进行重新修饰，将参数类型包含在最终的名字中，就可保证名字在底层的全局唯一性。

表 3-19

函数签名	修饰后名称
int func(int)	?func@@YAH@Z
float func(float)	?func@@YAM@Z
int C::func(int)	?func@C@@AAEHH@Z
int C::C2::func(int)	?func@C2@C@@AAEHH@Z
int N::func(int)	?func@N@@YAH@Z
int N::C::func(int)	?func@C@N@@AAEHH@Z

我们以 int N::C::func(int)这个函数签名来猜测 Visual C++的名称修饰规则（当然，你只须大概了解这个修饰规则就可以了）。修饰后名字由“?”开头，接着是函数名由“@”符号结尾的函数名；后面跟着由“@”结尾的类名“C”和名称空间“N”，再一个“@”表示函数的名称空间结束；第一个“A”表示函数调用类型为“__cdecl”（函数调用类型我们将在第4章详细介绍），接着是函数的参数类型及返回值，由“@”结束，最后由“Z”结尾。可以看到函数名、参数的类型和名称空间都被加入了修饰后名称，这样编译器和链接器就可以区别同名但不同参数类型或名字空间的函数，而不会导致link的时候函数多重定义。

请同学们自己验证下，Linux对于函数的名字是如何修饰的。

【扩展学习：C/C++函数调用约定和名字修饰规则】

[C++函数重载](#)

[C/C++的调用约定](#)

5.3 extern “C”

有时候在C++工程中可能需要将某些函数按照C的风格来编译，在函数前加extern "C"，意思是告诉编译器，将该函数按照C语言规则来编译。

```
extern "C" int Add(int left, int right);

int main()
{
    Add(1,2);
    return 0;
}
```

链接时报错：error LNK2019: 无法解析的外部符号_Add，该符号在函数 _main 中被引用

【面试题】

1. 下面两个函数能形成函数重载吗？有问题吗或者什么情况下会出问题？

```
void TestFunc(int a = 10)
{
    cout<<"void TestFunc(int)"<<endl;
}

void TestFunc(int a)
{
    cout<<"void TestFunc(int)"<<endl;
}
```

2. C语言中为什么不能支持函数重载？
3. C++中函数重载底层是怎么处理的？
4. C++中能否将一个函数按照C的风格来编译？

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。

比如：李逵，在家称为“铁牛”，江湖上人称“黑旋风”。



类型& 引用变量名(对象名) = 引用实体；

```

void TestRef()
{
    int a = 10;
    int& ra = a; //<====定义引用类型

    printf("%p\n", &a);
    printf("%p\n", &ra);
}

```

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，再不能引用其他实体

```

void TestRef()
{
    int a = 10;
    // int& ra;    // 该语句编译时会出错
    int& ra = a;
    int& rra = a;
    printf("%p %p %p\n", &a, &ra, &rra);
}

```

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;
}

```

6.4 使用场景

1. 做参数

```

void Swap(int& left, int& right)
{
    int temp = left;
    left = right;
    right = temp;
}

```

2. 做返回值

```
int& TestRefReturn(int& a)
{
    a += 10;
    return a;
}
```

下面代码输出什么结果？为什么？

```
int& Add(int a, int b)
{
    int c = a + b;
    return c;
}

int main()
{
    int& ret = Add(1, 2);
    Add(3, 4);
    cout << "Add(1, 2) is : "<< ret << endl;
    return 0;
}
```

注意：如果函数返回时，离开函数作用域后，其栈上空间已经还给系统，因此不能用栈上的空间作为引用类型返回。如果以引用类型返回，返回值的生命周期必须不受函数的限制(即比函数生命周期长)。

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，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。

```
#include <time.h>
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(int*)-time:" << end1 - begin1 << endl;
cout << "TestFunc2(int&)-time:" << end2 - begin2 << endl;
}

// 运行多次，检测值和引用在传参方面的效率区别
int main()
{
    for (int i = 0; i < 10; ++i)
    {
        TestRefAndValue();
    }

    return 0;
}

```

6.5.2 值和引用的作为返回值类型的性能比较

```

#include <time.h>
struct A
{
    int a[10000];
};

A a;

A TestFunc1()
{
    return a;
}

A& TestFunc2()
{
    return a;
}

void TestReturnByRefOrValue()
{
    // 以值作为函数的返回值类型

    size_t begin1 = clock();

```

```

    for (size_t i = 0; i < 100000; ++i)
        TestFunc1();
    size_t end1 = clock();

    // 以引用作为函数的返回值类型
    size_t begin2 = clock();
    for (size_t i = 0; i < 100000; ++i)
        TestFunc2();
    size_t end2 = clock();

    // 计算两个函数运算完成之后的时间
    cout << "TestFunc1 time:" << end1 - begin1 << endl;
    cout << "TestFunc2 time:" << end2 - begin2 << endl;
}

// 测试运行10次，值和引用作为返回值效率方面的区别
int main()
{
    for (int i = 0; i < 10; ++i)
        TestReturnByRefOrValue();

    return 0;
}

```

通过上述代码的比较，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。

```

int main()
{
    int a = 10;
    int& ra = a;

    cout<<"a = "<<a<<endl;
    cout<<"ra = "<<ra<<endl;

    return 0;
}

```

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 main()
{
    int a = 10;

    int& ra = a;
    ra = 20;

    int* pa = &a;
    *pa = 20;

    return 0;
}

```

我们来看下引用和指针的汇编代码对比：

int a = 10;	int a = 10;
mov dword ptr [a], 0Ah	mov dword ptr [a], 0Ah
int& ra = a;	int* pa = &a;
lea eax, [a]	lea eax, [a]
mov dword ptr [ra], eax	mov dword ptr [pa], eax
ra = 20;	*pa = 20;
mov eax, dword ptr [ra]	mov eax, dword ptr [pa]
mov dword ptr [eax], 14h	mov dword ptr [eax], 14h

引用和指针的不同点：

1. 引用在定义时**必须初始化**，指针没有要求
2. 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
3. **没有NULL引用**，但有NULL指针
4. 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)
5. 引用自加即引用的实体增加1，指针自加即指针向后偏移一个类型的大小
6. **有多级指针**，**但是没有多级引用**
7. 访问实体方式不同，**指针需要显式解引用**，引用编译器自己处理
8. **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**，没有函数压栈的开销，内联函数提升程序运行的效率。

<pre> int Add(int left, int right) { return left + right; } int main() { int ret = 0; ret = Add(1, 2); return 0; } </pre>	<pre> int ret = 0; mov dword ptr [ret], 0 ret = Add(1, 2); push 2 push 1 call Add @12C107Dh add esp, 8 mov dword ptr [ret], eax </pre>
--	---

如果在上述函数前增加inline关键字将其改成内联函数，在编译期间编译器会用函数体替换函数的调用。

查看方式：

1. 在release模式下，查看编译器生成的汇编代码中是否存在call Add
2. 在debug模式下，需要对编译器进行设置，否则不会展开(因为debug模式下，编译器默认不会对代码进行优化，以下给出vs2013的设置方式)



<pre> inline int Add(int left, int right) { return left + right; } int main() { int ret = 0; ret = Add(1, 2); return 0; } </pre>	<pre> int ret = 0; mov dword ptr [ret], 0 ret = Add(1, 2); mov eax, 1 add eax, 2 mov dword ptr [ret], eax </pre>
---	--

7.2 特性

1. inline是一种以空间换时间的做法，省去调用函数额开销。所以代码很长或者有循环/递归的函数不适宜使用作为内联函数。

2. **inline对于编译器而言只是一个建议**，编译器会自动优化，如果定义为inline的函数体内有循环/递归等等，编译器优化时会忽略掉内联。
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。

```
// F.h
#include <iostream>
using namespace std;

inline void f(int i);

// F.cpp
#include "F.h"
void f(int i)
{
    cout << i << endl;
}

// main.cpp
#include "F.h"
int main()
{
    f(10);
    return 0;
}

// 链接错误: main.obj : error LNK2019: 无法解析的外部符号 "void __cdecl f(int)" (?f@@YAXH@Z), 该符号在函数 _main 中被引用
```

【面试题】

宏的优缺点？

优点：

- 1.增强代码的复用性。
- 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
2. 函数定义 换用内联函数

8. auto关键字(C++11)

8.1 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？

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 TestAuto()
{
    return 10;
}

int main()
{
    int a = 10;
    auto b = a;
    auto c = 'a';
    auto d = TestAuto();

    cout << typeid(b).name() << endl;
    cout << typeid(c).name() << endl;
    cout << typeid(d).name() << endl;

    //auto e; 无法通过编译，使用auto定义变量时必须对其进行初始化
    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。

8.2 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声明引用类型时则必须加&

```
int main()
{
    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;
}
```

```
*a = 20;
*b = 30;
c = 40;

return 0;
}
```

2. 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，然后用推导出来的类型定义其他变量。

```
void TestAuto()
{
    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
}
```

8.3 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

```
void TestAuto()
{
    int a[] = {1, 2, 3};
    auto b[] = {4, 5, 6};
}
```

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。

5. auto不能定义类的非静态成员变量(暂不做讲解，后面讲)

6. 实例化模板时不能使用auto作为模板参数(暂不做讲解，后面讲)

9. 基于范围的for循环(C++11)

9.1 范围for的语法

在C++98中如果要遍历一个数组，可以按照以下方式进行：

```

void TestFor()
{
    int array[] = { 1, 2, 3, 4, 5 };
    for (int i = 0; i < sizeof(array) / sizeof(array[0]); ++i)
        array[i] *= 2;

    for (int* p = array; p < array + sizeof(array) / sizeof(array[0]); ++p)
        cout << *p << 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。

```

void TestFor()
{
    int array[] = { 1, 2, 3, 4, 5 };
    for(auto& e : array)
        e *= 2;

    for(auto e : array)
        cout << e << " ";

    return 0;
}

```

注意：与普通循环类似，可以用continue来结束本次循环，也可以用break来跳出整个循环。

9.2 范围for的使用条件

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的范围不确定

```

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

```

2. 迭代的对象要实现++和==的操作。

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(stddef.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()
{
    f(0);
    f(NULL);
    f((int*)NULL);
    return 0;
}
```

程序本意是想通过f(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 *)0。

10.2 nullptr 与 nullptr_t

为了考虑兼容性，C++11并没有消除常量0的二义性，C++11给出了全新的nullptr表示空值指针。C++11为什么不在NULL的基础上进行扩展，这是因为NULL以前就是一个宏，而且不同的编译器厂商对于NULL的实现可能不太相同，而且直接扩展NULL，可能会影响以前旧的程序。因此：**为了避免混淆，C++11提供了nullptr，即：nullptr代表一个指针空值常量。**nullptr是有类型的，其类型为nullptr_t，仅仅可以被隐式转化为指针类型，nullptr_t被定义在头文件中：

```
typedef decltype(nullptr) nullptr_t;
```

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作为新关键字引入的。
- 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。

11. 总结

11.1 本节知识点回顾

知识块	知识点	分类	掌握程度
C++关键字	C++98关键字	概念型	了解
命名空间	命名空间定义	概念型	掌握
	命名空间使用	应用型	掌握
C++输入输出	cout输出 cin输入	应用型	掌握
缺省参数	全缺省&半缺省	概念型	掌握
函数重载	概念和用法	概念型+考点型	熟悉
	name mangling	原理型	掌握
	extern "C"	考点型	掌握
引用	应用概念&特性	概念型	熟悉
	使用场景	应用型	熟悉
	效率比较	应用型	掌握
	引用底层原理	原理型	掌握
	引用和指针区别	考点型	熟悉
内联函数	概念和用法	概念型	掌握
	宏的优缺点？C++有哪些技术替代宏	考点型	熟悉
auto关键字	auto的简单使用	概念性	掌握
	auto的使用规则	应用型	掌握
	auto不能推导的场景	原理型	掌握
	auto的优势	概念性	掌握
范围for循环	范围for的语法	应用型	掌握
	范围for的条件	应用型	掌握
指针空值-null	0与NULL	概念性	掌握
	nullptr	应用型	掌握

11.2 本节作业

1. 将课堂代码敲一遍，做好笔记
2. 将本届知识点写一篇博客写一篇博客

比特科技