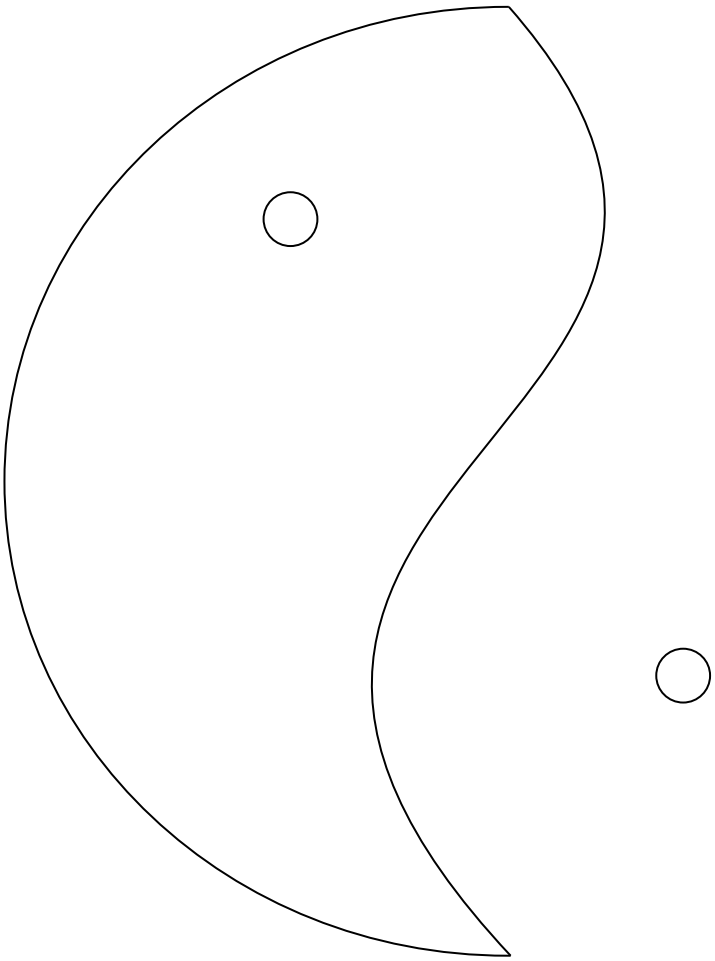
Obtuse

Wenqingqian



wenqq3@outlook.com

目录

第一章 C++

1.1 基础

1.1.1 <===>C 和 C++ 区别

1. C 语言是 C++ 的子集, C++ 可以很好兼容 C 语言. 但是 C++ 又有很多新特性, 如引用、智能指针、auto 变量等
2. C++ 是面对对象的编程语言; C 语言是面对过程的编程语言
3. C 语言有一些不安全的语言特性, 如指针使用的潜在危险、强制转换的不确定性、内存泄露等. 而 C++ 对此增加了不少新特性来改善安全性, 如 const 常量、引用、cast 转换、智能指针、try—catch 等等
4. C++ 可复用性高, C++ 引入了模板的概念, 后面在此基础上, 实现了方便开发的标准模板库 STL. C++ 的 STL 库相对于 C 语言的函数库更灵活、更通用

1.1.2 <===>C++ 从代码到可执行二进制文件的过程

1. 预编译: 这个过程主要的处理操作如下
   (a) 将所有的 #define 删除, 并且展开所有的宏定义
   (b) 处理所有的条件预编译指令, 如 #if、#ifdef
   (c) 处理 #include 预编译指令, 将被包含的文件插入到该预编译指令的位置
   (d) 过滤所有的注释
   (e) 添加行号和文件名标识
2. 编译: 这个过程主要的处理操作如下
   (a) 词法分析: 将源代码的字符序列分割成一系列的记号
   (b) 语法分析: 对记号进行语法分析, 产生语法树
   (c) 语义分析: 判断表达式是否有意义
   (d) 代码优化
   (e) 目标代码生成: 生成汇编代码
   (f) 目标代码优化

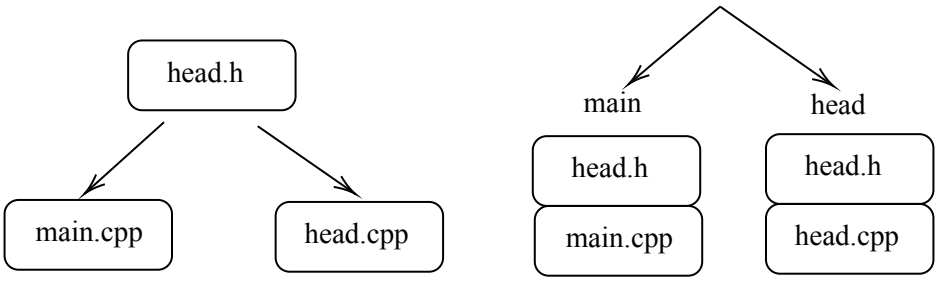
3. 汇编: 这个过程主要是将汇编代码转变成机器可以执行的指令
4. 将不同的源文件产生的目标文件进行链接, 从而形成一个可以执行的程序
   - 链接分为静态链接和动态链接
   - 静态链接, 是在链接的时候就已经把要调用的函数或者过程链接到了生成的可执行文件中, 就算你在去把静态库删除也不会影响可执行程序的执行; 生成的静态链接库, Windows 下以.lib 为后缀, Linux 下以.a 为后缀
   - 而动态链接, 是在链接的时候没有把调用的函数代码链接进去, 而是在执行的过程中, 再去找要链接的函数, 生成的可执行文件中没有函数代码, 只包含函数的重定位信息, 所以当你删除动态库时, 可执行程序就不能运行生成的动态链接库, Windows 下以.dll 为后缀, Linux 下以.so 为后缀

1.1.3 <===>static 关键字

1. 全局静态变量和局部静态变量: 初始化的静态变量会在数据段分配内存, 未初始化的静态变量会在 BSS 段分配内存. 直到程序结束, 静态变量始终会维持前值. 只不过全局静态变量和局部静态变量的作用域不一样
2. 静态函数: 静态函数只能在本源文件 (该翻译单元) 中使用
3. 类中的静态成员变量: 静态数据成员, 隐藏在类作用域中的全局变量 (可以通过类名 (Class::) 或类对象访问). 类中的 static 静态数据成员拥有一块单独的存储区, 而不管创建了多少个该类的对象. 所有这些对象的静态数据成员都共享这一块静态存储空间
4. 类中的静态成员函数: 静态成员函数也是类的一部分, 而不是对象的一部分
   只能访问静态数据成员: 当调用一个对象的非静态成员函数时, 系统会把该对象的起始地址赋给成员函数的 this 指针. 而静态成员函数不属于任何一个对象, 因此 C++ 规定静态成员函数没有 this 指针. 既然它没有指向某一对象, 也就无法对一个对象中的非静态成员进行访问

1.1.4 <===>extern 关键字与链接性

在 C++ 中, 翻译单元由实现文件及直接或间接包含的所有标头组成

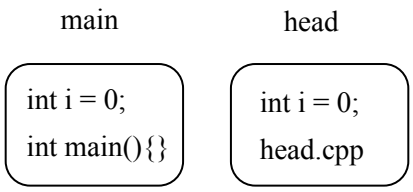


每个翻译单元由编译器单独编译, 最终将得到的.o 文件链接得到可执行文件. 如上图所示程序结构, 在链接时会出现如下问题:

假如在 head.h 文件中定义一个外部链接性变量

```
#ifndef HEAD_H
#define HEAD_H
int i = 0;
#endif
```

那么在链接时就会出错, 原因是如上两个翻译单元相当于



在两个文件中都定义了具有外部链接性的变量 i.

在一个作用域内, 变量能且只能被定义一次, 但是可以被多次声明 (one define rule)

定义

```
// 不能在局部作用域内使用
extern int v = 0;
int v;
```

声明

```
void func();
extern void func();
extern int v;
```

如果需要在多个文件, 指个多翻译单元中使用同一变量, 就必须把定义与声明分离. 变量的定义只能出现在一个文件中, 而在其它用到该变量的文件中对其声明.

所以往往头文件中只放一些声明而在另外的文件中进行定义.

常见的内部链接类型

1. const 全局变量
2. static 全局变量
3. static 函数
4. inline[1] 函数/变量[2]
5. 类内定义的成员函数
6. 类内定义的数据成员 (即非静态数据成员)

其它可视为无链接

常见的外部链接类型

1. 全局变量与函数 (非 const、static、inline)
2. 类外定义的数据成员 (即静态数据成员在类外的定义)
3. 类外定义的成员函数
4. extern const T v = INIT;

所有文件中的外部链接性变量会传递给链接器得到一张导出符号表. 记录本编译单元定义, 并且可提供给其它单元使用的符号及在本单元对应的地址.

所有文件中的声明会传递给链接器得到一张未解决符号表, 记录本编译单元有声明但不在本单元定义的符号及其对应的地址, 显然在导出符号表中不能存在相同的符号.

[1] inline 现在表示在链接时遇到不同编译单元出现了相同签名的函数时只保留一份, 不在是内联的意思.

[2] inline variable is C++17 extension

C11 标准关于 static 和 extern 的内容:

1. 若在 extern 声明标识符之前的可见范围内存在对该标识符的声明, 则该标识符的链接与先前声明相同. 若无先前声明, 或声明无链接, 则该标识符具有外部链接
2. 如果在同一翻译单元内, 同一标识符同时出现内外部链接, 则行为未定义

上述原文:

For an identifier declared with the storage-class specifier extern in a scope in which a prior declaration of that identifier is visible, if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.

1.1.5 <===>指针与引用

1.1.5.1 指针与引用的区别

1. 引用就是常量指针
2. 引用必须初始化, 且不能改变指向
3. 无法对引用进行取地址, 或者说只能取到引用指向的对象的地址
4. 引用的解引用操作由编译器自动进行

1.1.5.2 void* 指针

指针一般有三个含义

1. 指明数据的位置, 体现在指针的值
2. 表示数据的大小, 例如 int 指针表示四个字节为一组的数据, 体现在指针的步长、自身的加减法计算
3. 表示数据如何被解释, 例如 float 和 int 都是四字节, 但解释结果完全不同, 体现在指针解引用的结果

void* 指针为第一种, 只指明了数据的位置. 以 void* 的视角来看内存空间也仅仅是内存空间, 没办法访问内存空间中所存的对象

对于 void* 类型指针与其它类型指针的转换, C++ 有如下两个规定

1. 不允许从 void* 类型到其它类型的隐式转换, 允许从其它类型到 void* 类型的隐式转换, 常见于函数传参
2. 字面量 0 可以隐式转换成任意类型的空指针常量

1.1.5.3 nullptr <11>

在 C 中, 常用 NULL 来表示空指针

```
#define NULL ((void*)0)
```

然而在 C++ 中不允许从 void* 隐式转换成其它类型的指针, 所以定义了一种新的指针类型, 指针空值类型 std::nullptr_t , nullptr 是它的一个实例, 当对指针进行初始化时:

```
int *p = nullptr;
int *p = 0;
```

两者之间没有区别

但在重载情况下:

```
void func(int n);
void func(int *p);
```

nullptr 会匹配第二个, 而 0 实际上两个都能匹配, 在本实验环境中, 会匹配第一项且没有二义性问题

若再定义一个

```
void func(char *p);
```

则 nullptr 会出现二义性错误

至于空指针具体指向哪个地址由编译器管理, 可能是内存中的 0 号地址, 即指针数据全是 0, 也有可能不是

1.1.5.4 野指针

wild pointer 也叫悬挂指针, dangling pointer

有三种情况会造成野指针:

1. 指针未初始化
2. 指针指向的对象生命周期结束, 如将一个函数体内的局部变量地址传递给外部指针
3. 指针释放

当指针被 delete 之后, 指针值变为无效, 继续对该指针进行操作会出现 heap use after free 错误.

事实上, 当一个指针被 delete 之后, 指针指向的堆地址空间被释放, 随时可以被分配给其它对象, 但在不同的机器上会有不同的结果.

几种特殊情况, 指 delete 之后继续使用指针不报错的情况:

1. 指针的数据, 即它指向的地址发生改变, 对它进行解引用会得到 0
2. 地址不改变, 解引用会得到 0

OBTUSE

3. 地址不改变，该内存地址的值也未被擦除，解引用得到跟 delete 之前一样的结果

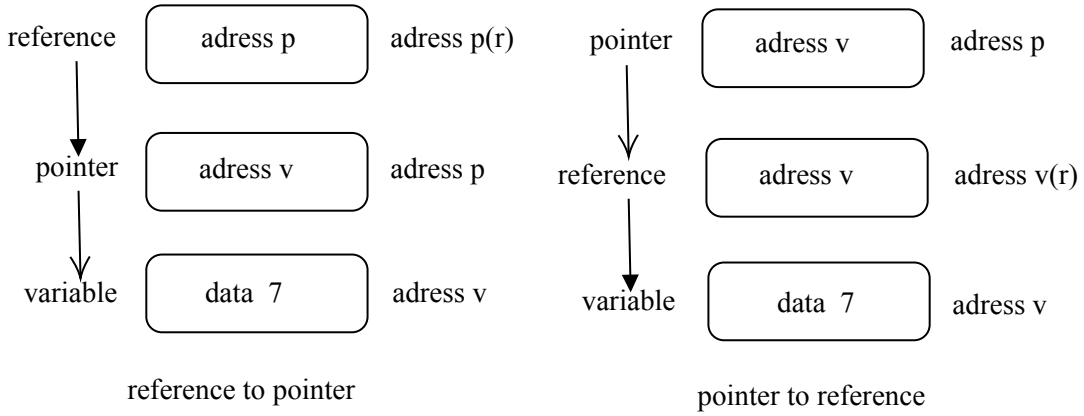
准确来讲，delete 的实际意义是将指针指向的内存空间释放，以便能分配给其它对象，但指针本身是否依旧指向那块内存，那块内存的数据是否被擦除未定义

给野指针赋 nullptr 初值避免麻烦

1.1.5.5 指向引用的指针，指向指针的引用

对引用地址的获取行为会得到引用指向的对象的地址

测试程序见 pointer_reference



1.1.6 const 关键字

1.1.7 malloc 和 new 区别

1.2 面向对象

1.2.1 多态的实现原理和应用场景

1.3 STL

1.3.1 线程安全的实现及标准容器库的线程安全性

1.3.2 sort 算法是怎么实现的

1.4 C++11

1.4.1 <===>decltype 和 auto

decltype

```
int *r=0;
decltype (*r+0) a; // 表达式结果为int, 因此a为int
```

auto 一般会忽略顶层 const，保留底层 const，引用本身就是一个顶层 const 指针，因此 auto 类型推导不会直接得到引用类型

```
decltype (r) a; //a是指针
decltype (*r) a = ini; //a是引用
decltype ((r)) a = ini; //a是指向指针的引用
decltype ((*r)) a = ini; //a是引用
```

decltype((variable)) 的结果永远是引用.

```
const int i = 1; const int&r = i;
auto a = i; //a是int
auto a = &i; //a是const int*(底层const)
auto a = r; //a是int
auto &a = i; //a是左引用
auto &&a;(万能引用)
```

1.4.2 <===>左右值

1.4.2.1 左值与右值

原文

左值是一个对象, 可以用 & 取地址, 而右值更贴近"值"本身. 一般来说, 表达式结束后, 值是否有显式的存储位置是左右值的区分 (右值可以隐式存在内存中)

左值:

1. 字符串字面量
2. 内置的前 ++ 与前--
3. 右值引用类型变量
4. 转型为左值引用的表达式
5. * 解引用的表达式

右值:

1. 非字符串的字面量以及枚举项
2. 置的后 ++ 与后- -
3. 内置的算术, 逻辑, 比较表达式
4. 内置取地址表达式, this 指针
5. 未命名的 lambda 表达式
6. 转型为非引用的表达式
7. 转型为右值引用的表达式

1.4.2.2 左值引用与右值引用

左右值并不是 C++11 才有的新概念, 而右值引用是 C++11 为了处理无需深拷贝的资源转移型需求而引入

左右引用只能处理对应的左右值. 一个例外是 const T&, 既可以引用左值, 也可以引用右值

即可以写出这样的代码

```
const int& clr = 1; const int& clr2 = clr;
```

对于右值来说, 不能真正取地址, 而引用实质就是操作地址指针, 因此理论上无法进行左引用. 对于左值来说, 可以显式转换成右值引用

如右值例 7, 转换为右值引用的表达式本身是一个右值, 但是通过右值引用绑

```
// 可以将左值转为右值, 再进行右引用
Test t(1);
// 使用std::move转为右值引用
Test&& t1 = std::move(t);
// 使用 static_cast 转为右值引用
Test&& t2 = static_cast <Test&&>(t);
// 使用C风格强转为右值引用
```

定到的对象自身是一个左值, 即左值例 3

```
Test&& t3 = (Test&&)t;
// 使用std :: forwad<T&&>为右值引用
Test&& t4 = std :: forward<T&&>(t);
```

1.4.3 <===>移动语义与完美转发

1.4.3.1 浅拷贝与深拷贝

当通过一个对象初始化另一个对象时,

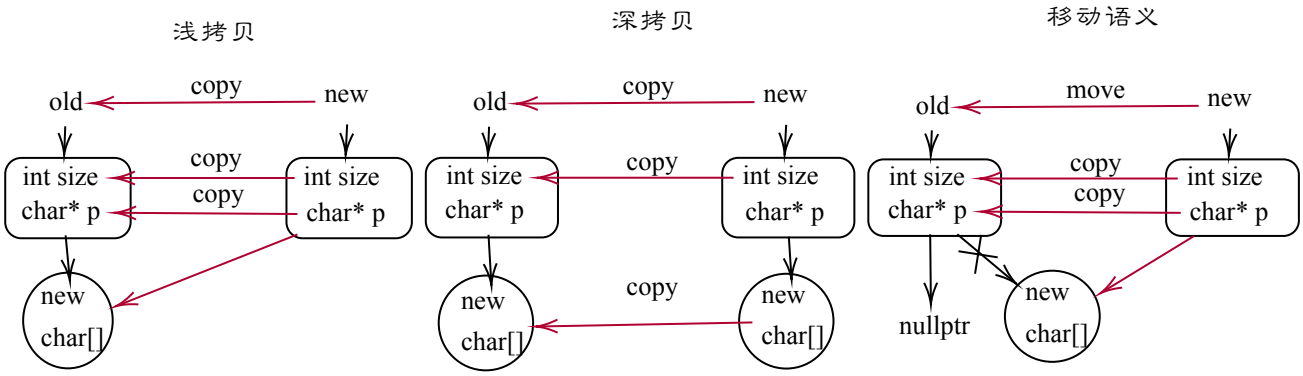
如 `Test a(10);Test b(a);` 时:

假设 Test 中有资源拖管

只记录资源 Data 的指针, 而非记录资源全部信息到类对象里面, 构造时从堆上用 new 申请内存供资源后面使用, 析构时从堆上释放内存, 回收掉资源, 中间过程资源读写操作都通过指针 p 来完成. C++ 对象构造与析构函数的成对调用, 也保证了安全性与不发生泄露 (RAII)

以右侧结构为例:

```
class Test1 public:
    Test1(int s):size(s)
        p = new char[size];
    ~Test1()
        if(p)  delete []p;
               p = nullptr;
    int size;
    char* p;
```



1. 浅拷贝: 没有真正 new 内存, 两者共用一块内存地址, 当最后都析构时, 这块内存会被析构两次, 发生 crash
2. 深拷贝: 将指针指向的资源深入复制了一份
3. 移动语义: 当原对象不再使用, 使用移动语义, 转移资源, 减少拷贝

参考代码见 copy_move_constructor

1.4.3.2 移动语义

移动语义: 一个对象的资源在销毁前, 将其转移给其它对象再用起来, 这样能减少资源带来的构造开销, 程序获得更高的效能

移动语义的使用场景就是需要深拷贝原对象的资源, 而原对象不再使用, 就可以使用移动语义, 本

质就是在浅拷贝的基础上将原对象具有资源的指针置空.

在 C++11 之前, 移动语义使用的问题:

1. 使用移动语义需要在构造函数中加额外的参数来选择是否使用移动语义
2. 因为需要修改类内数据, 无法使用拷贝构造函数 (const T&)
3. 非 const 引用又无法引用右值, 并且也需要 const 引用来处理 const 对象

C++11 直接统一, 增加 && 右值引用, 事情变简单, 想要深拷贝的就 T(const T&) 实现; 想要移动拷贝的就实现 T(T&&). 两者都实现时, 一旦传入是个右值, 根据重载机制会优先触发移动拷贝构造调用, 使用移动语意, 转移资源, 减少拷贝

-> 底层 const 可用于重载而顶层不能, 对于非 const 对象会优先匹配非 const 引用

1.4.3.3 万能引用与引用折叠

对于一个模板

```
template<typename T> T func(T para){}
```

因为引用与引用的对象是没有区别的, 当传入一个引用时, 无法像指针那样分辨出来.

```
int i = 1; func(i); -- int func<int >(int para){}
int&r = i; func(r); -- int func<int >(int para){}
int*p =&i; func(p); -- int*func<int*>(int*para){}
```

所以当需要引用一个新的对象, 或是引用一个引用需要标注参数是一个引用.

```
template<typename T> T& func(T& para){}
int i = 1; func(i); -- int& func<int>(int& para){}
int&r = i; func(r); -- int& func<int>(int& para){}
        //T的值仍是int
```

在引入右值引用 && 后情况出现了区别, 此时对于一个新的模板

```
template<typename T> T&& func(T&& para){}
```

它表明参数是一个引用, 但未兴是右引用, 因为 C++ 的引用折叠机制

- T& & 折叠为 tt T&
- T&& & 折叠为 T&
- T& && 折叠为 T&
- T&& && 折叠为 T&&

因此对于传入值为左值或右值会进行如下推导

```
int i = 1; func(i); -- int& func<int&>(int& para){}
           func(1); -- int&&func<int >(int&&para){}
```

当传入为左值 T=int&, 根据引用折叠机制 int& &&=int&; 当传入为右值 T=int .(万能引用处理

右值时, T 为 int 而不是 T&&)

万能引用可以接受左值和右值, 但具有如下规则:

1. 万能引用的 T 不能被再修饰, 不能被 cv 修饰限定, 否则转为普通右值引用
2. 在模板类中的模板函数使用万能引用, 只能接受模板函数的模板参数不接受模板类的模板参数, 否则转为普通右值引用

1.4.3.4 move 与 forward

引用移除:

```
template <class _Tp> struct remove_reference        {typedef _Tp type;};
template <class _Tp> struct remove_reference<_Tp& > {typedef _Tp type;};
template <class _Tp> struct remove_reference<_Tp&&> {typedef _Tp type;};
```

将类型转化为 remove_reference::type 时, 根据原类型自动选择合适的版本并得到去掉引用后的类型. 一般是在模板中传递模板参数时可以将模板参数转化成无引用类型

由于移动语义只接受右值, 对于一般对象需要将其转化为右值.

move

```
//原型
template <class _Tp>
typename remove_reference<_Tp>::type&&
move(_Tp&& __t) noexcept{
    return static_cast<typename remove_reference<_Tp>::type&&>(__t);
}
//传入为左值:
int i = 1; int&& r = move(i);
//函数原型
typename remove_reference<int&>::type&&
move(int& t) noexcept{
    return static_cast<typename remove_reference<int&>::type&&>(t);
}

//相当于
int&& move(int& t) noexcept{return static_cast<int&&>(t);}
```

1.万能引用模板推导T=int&, 原&&被引用折叠
2.引用移除接收到模板参数T=int&, 并返回去除引用后的结果即int
--步骤2的必要性在于将原模板参数引用去除后在添加&&才能得到右值引用类型, 原类型带有左引用时会导致添加的右引用被折叠
3.返回值是右值的原因是将其强转为右值引用的这个表达式本身是一种值的表

```
    达，一种临时变量，而不是右值引用类型是右值，如常见左值例3.
---
//传入为右值:
int&& r = move(1);
//函数原型
typename remove_reference<int >::type&&
move(int&& t) noexcept{
    return static_cast<typename remove_reference<int >::type&&>(t);
}
//相当于
int&& move(int&&t) noexcept{return static_cast<int&&>(t);}
```

如上述所言，当一个右值被右值引用后，引用本身就成了左值，这意味着无法将其按照右值进行继续传播.

forward：函数传进来是左引用，函数体内也保持左引用，传进来是右引用，函数体内也保持右引用，称为完美转发

forward

```
//处理左值作为左引用或者右引用
template <class _Tp>
_Tp&&
forward(typename remove_reference<_Tp>::type& __t) noexcept{
    return static_cast<_Tp&&>(__t);
}
//处理右值作为右引用
template <class _Tp>
_Tp&&
forward(typename remove_reference<_Tp>::type&& __t) noexcept{
    static_assert(!is_lvalue_reference<_Tp>::value,
                  "can not forward an rvalue as an lvalue");
    return static_cast<_Tp&&>(__t);
}
//底层都是static_cast<T&&>(t)
//需要显示传递模板参数
//由该函数易知，传入的模板参数类型决定了传出的类型

//函数参数使用remove的目的是
//不管传入的模板参数是什么，保证左右值会进入对应的版本
//以检测不经意间指定了错误的模板参数，如下面将要介绍的传递了左
    引用模板参数而函数参数是一个右值
```

具体场景

```
auto&& n = func(1);
```

```
1.                  | 2.                            | 3.
template<class T>   | template<class T>             | template<class T>
T&& func(T&& t){    | T&& func(T&& t){              | T&& func(T&& t){
    return t;       |     return forward<T>(t);     |     return
}//error           | }                             |     static_cast<T&&>(t);}
```

1. T=int, 返回类型为int&&, 而t是左值.
2.3. T=int, return forward<int>(t) == return static_cast<int&&>(t)
-. 原类型是左引用返回左引用, 原类型右引用返回右引用, 完美转发了对象保持原来的左右引用关系并向下层次转化, 不再因右引用表达式是左值从而变掉
当传入左值时, T=int&, 2.3. ==
int& func(int& t){ return forward/static_cast<int&>(t); }

大部分情况下,static_cast<T&&>(t) 等同 forward<T>(t)

特殊情况

```
template<class A, class B>
A&& StaticCastFun(A&& a, B&& b) {
    return static_cast<A&&>(b);
}//warning
template<class A, class B>
A&& ForwardCastFun(A&& a, B&& b) {
    return forward<A>(b);
}//error

struct TypeB{};
struct TypeA{
    TypeA() {}
    TypeA(const TypeA&) {}
    ~TypeA() {}
    TypeA(const TypeB&) {}
};
---
int main(){
    TypeA ta1;
    TypeB tb1;
    [&](const TypeA a, const TypeB b){
        StaticCastFun(a, b);
```

```
            ForwardCastFun(a, b);
        }(ta1,tb1);
        return 0;
    }
```

当传入参数与模板参数不是同一类别时，且存在TypeB到TypeA的转换则会在forward<A>(b)时将TypeB b通过TypeA提供的构造函数生成一个临时的TypeA变量并传入forward中

这个临时变量为右值，因此会触发forward右值版本，而A为const TypeA&

会返回对右值的左引用，因此forward分开两个版本并在右值处理中判断T是否为左引用. 除此之外与static_cast<T&&>无差别

测试程序见: lrvalue

1.4.4 <===>using

1.4.4.1 作用域导入

把目标的访问权限复制到 using 所在的当前作用域, 使得目标视为等同于在当前作用域定义

使用 using 导入命名空间, 即使一个命名空间中的所有名字都在该作用域中可见

```
using namespace std; // 导入整个命名空间到当前作用域
using std::cout;     // 只导入某个变量到当前作用域
```

在派生类中引用基类成员

```
class A{
public:     void func();
protected:  int val;
private:    int pval;
};

class B:private A{    //尽管派生类B对基类A是私有继承，但通过using声明，
public:               //派生类的对象就可以访问基类的
                      //proteced成员变量和public成员函数
    using A::func;    //using的声明在public中，视为在B的public中声明基类成员
    using A::val ;    //简单来说，就是将原本继承后的private权限改为public
//////////////////////
//public or private: //error
//  using A::pval;    //无法对基类private成员进行using
};
```

1.4.4.2 别名声明

通过 using 指定别名, 作用等同 typedef

1.4.4.3 别名模板

using 能够使用模板, 而 typedef 需要通过结构体包装

```
template<typename T> using  vec = vector<T>;
template<typename T> struct vec {
    typedef vector<T> type;
    or
    using vec = vector<T>;
};
vec<int>       v_using
vec<int>::type v_typedef
```

1.4.4.4 typename

使用 typedef 加结构体包装的形式实现别名模板存在一个问题, 在使用此别名模板时若没有指定模板参数而是传入另一个模板参数, 编译器会无法确认结构体中的这个 type 是不是类型.

template 内出现的名称如果相依于某个 template 参数, 称之为从属名称. 如果从属名称在 class 内成嵌套状 (即通过从属名称声明的成员 (::type) 在其作用域内与该从属名称也存在依赖关系), 称为嵌套从属名称, 即无法确认为类型还是变量的名称

C++ 规定在嵌套从属名称前加 typename 关键字以确定该名称是一个类型.

```
template<typename T, class U>
void func (T i){
            _typedef<int>::type a0 = i;  //非从属名称
            _using  <T  >       a1 = i;  //从属名称
            _typedef<T  >       a2    ;  //从属名称
    typename _typedef<T  >::type a3 = i;  //嵌套从属名称
    typename U            ::type a4 = i;  //嵌套从属名称
}
//using    可以接受模板   模板绑定在别名上
//typedef  不可以接受模板 模板绑定在结构体上
//        在结构体内再声明与模板参数相同的类型成员, 形成了嵌套
//        结构体_typedef可能在某个特化的情况下存在成员type不为类
    型的情况
```

特例: 请使用关键字 typename 标识嵌套从属名称, 但不能在基类列 (base class lists) 和成员初值列 (member initialization list) 中用 typename 修饰基类 (base class), 这是确定的一个

类型.

```
template<typename T>
class C{
public:
    C(){}
    C(T x):val(x){}
    T val;
};
template<typename T>
struct Base{
    typedef C<T> type;
};

template<typename T>
class D:public Base<T>::type{
public:
    D(int x):Base<T>::type(x){
        typename Base<T>::type tmpC;
    }
};
```

测试程序见: using_typedef

1.4.5 <===>范围循环

在 C++11 中, 凡是支持 begin, end 方法的对象都可以使用范围循环, 本质上是等于迭代器遍历, 只不过细节被隐藏使得看上去更简洁.

```
vector<int>v;       | = for(auto it = begin(v); it != end(v); ++it)
for(auto n:v) n++; |        auto n = *it, n++;

vector<int>v;       | = for(auto it = begin(v); it != end(v); ++it)
for(auto&n:v) n++; |        ++*it;//(*it)++;
```

1.4.5.1 begin 和 end

begin 和 end 方法对内置数组类型进行了重载, 对其它对象则由对象自身实现相关的迭代器方案, 常见的标准库容器 string, vector 都支持.begin().end() 方法.

部分 impl

```
template <class Tp, size_t Np>      template <class Tp, size_t Np>
Tp* begin(Tp (&array)[Np]){         Tp* end(Tp (&array)[Np]){
```

```
    return array;
}
```

```
    return array + Np;
}
```

```
template <class Cp>
auto begin(Cp& c)
-> decltype(c.begin()){
    return c.begin();
}
```

```
template <class Cp>
auto end(Cp& c)
-> decltype(c.end()){
    return c.end();
}
```

1.4.5.2 数组指针

数组类型与指针很相似, 在很多情况下会退化成一个指针, 失去一部分信息, 无法再使用 begin 和 end 获得头尾指针

```
int a[]={1,2,3};
int* p = a;   //指针, 指向a[0].
int(*p)[3]=&a;//数组指针, 相当于一个二级指针, p指向a.
int(&r)[3]=a; //数组引用, 跟数组没有区别, 仍保留数组大小信息.

void f(int a[]);   //数组仍会退化成一个指针 = void f(int* a);
void f(int(&a)[N]);//不会退化
void f2(auto& a);  //相当于 int(&a)[N]
void f2(auto a);   //相当于 int* a

f(a); //会出现二义性错误, 数组a能同时匹配数组类型和指针类型
f(p); //只会匹配第一个
```

对于一个二维数组, 若想使用范围循环, 在外层循环需要使用引用才能使原本该得到的一维数组不退化成指针, 否则在内层循环中无法对指针进行遍历.

```
int a[3][3]={1,2,3,4,5,6,7,8,9};
//for(int(&c)[3]:a)
//外层循环得到三个一维数组的引用
for(auto&c:a){
    for(int i:c){
        cout<<i;
    }
}
//for(int*c :a)
//外层循环得到三个一维数组退化后的指针
//即指向三个一维数组首元素地址的指针
for(auto c:a){
```

```
        for(int i=0;i<3;++i){
            cout<<c[i];
        }
    }
    auto ftest3_p = [](int p[][3]){
                //= [](int(*p)[3]) 两种写法相同
        for(int i=0;i<3;++i){
            for(int j=0;j<3;++j){
                //(*(p+i))解引用得到一级指针,指向含有三个元素的数
                    组头
                cout<<(*(p+i))[j];
            }
        }
    };
```

参考代码见: range_for

1.4.6 强制类型转换

C++ 引入新的强制类型转换机制, 主要是为了克服 C 语言强制类型转换没有从形式上体现转换功能和风险不同的问题.

C 语言的类型转换, 编译器会在下面这些 C++ 的转换方法去逐一评估, 直到挑选到合适的.

如果想清晰确认每一步转换是怎么进行的, 使用 C++ 式转换, 对于简单的, 嫌麻烦可由编译器代劳使用 C 式转换.

通过对不同形式的转换进行分类命名可以更清晰分辨程序在转换过程中的具体形式, 使得出现问题时更容易找到可能与之相关的那一类转换方法.

1.4.6.1 static_cast

低风险转换方式, 不能处理具有底层 const 属性的类型.

常用于基本内置类型的转换, void* 指针与其它类型指针的转换, 基类与派生类指针的转换.

```
void fstatic(){
    double a = 2.2;
    int    b = static_cast<int>(a);

    void* p  = &b;
    int*  ip = static_cast<int*>(p);
}
```

1.4.6.2 const_cast

只能用于删除运算对象的底层 const, 也只有这个函数可以修改对象的底层 const

```
void fconst(){
```

属性.

即将一个指向常量的指针转换成普通指针, 获得修改指向对象的权限, 此方法只适用于指向对象是非 const 对象, 若指向对象是 const 则行为未定义.

```
        int  n  = 1;
    const int* cp = &n;
    //(*cp)++; error
        int* p  = const_cast
                    <int*>(cp);
    (*p)++;

    const int  cn  = 1;
    const int* cp2 = &cn;
        int* p2  = const_cast
                     <int*>(cp2);
    (*p2)++;
    //未定义行为, 本环境未改变cn
}
```

1.4.6.3 reinterpret_cast

为运算对象的位模式提供底层重新解释, 用于进行各种不同类型的指针之间、不同类型的引用之间的转换, 执行的过程是逐个比特复制的操作.

自由程度高, 可以将 int* 指针转成 string*, 至于之后引发的错误...

对于基类指针与派生类指针的转换不会进行地址的偏移.

```
void freinterpret(){
    int     a  = 1;
    int*    p  = &a;
    float*  dp = reinterpret_cast
                   <float*>(p);
    //(*dp)=1.4013e-45
    //float和int都是32位但是编码格式
       不同
    //比特层面上相同, 不同的解释方式
    //得到不同的结果
}
```

1.4.6.4 dynamic_cast

TODO:dc

测试程序见: cast

1.4.7 <===>智能指针

1.4.7.1 RAII

resource acquisition is initialization

资源获取即变量初始化, 与之对应的便是变量析构即资源释放. 即将需要手动释放的堆资源转换成

自动析构的栈对象.

RAII 要求, 资源的有效期与持有资源的对象的生命期严格绑定, 即由对象的构造函数完成资源的分配, 同时由对象的析构函数, 完成资源的释放. 在这种要求下, 只要对象能正确地析构, 就不会出现资源泄露问题.

- 当函数有多个返回点，或者因异常等原因意外退出函数时，栈上对象会正常的被析构，同时将资源安全释放掉。如果不用 RAII，不仅可能忘记在某个返回点写销毁代码，而且在程序异常时无法保证资源自动释放.

对于异常情况见 noexcept

1.4.7.2 unique_ptr

智能指针就是使用 RAII 方法封装的裸指针, 智能指针不能和裸指针混用

1. 一个 unique_ptr 独享它指向的对象, unique_ptr 无法被拷贝, 可以通过转移构造函数将一个 unique_ptr 所指向的资源转移给另一个 unique_ptr
2. 相比于使用引用计数且实现了线程安全的 shared_ptr, unique_ptr 的效率更高, 在非必要情况下优先使用 unique_ptr
3. 通过将拷贝构造函数和拷贝赋值运算符设为 delete 实现禁止拷贝
4. 构造方式
   (a) 直接传入一个返回指针的 new 表达式
   (b) move 另一个 unique_ptr
   (c) make_unique 返回一个 unique_ptr 型临时变量, 相当于第二种构造方式
5. 主要函数
   (a) reset 将指针指向新的资源, 释放原有资源
   (b) release 放弃资源的控制权, 将资源指针返回

test 中实现了一个简易 unique_ptr 可以清晰看出 unique_ptr 的结构, 参考代码见: unique_ptr

1.4.7.3 shared_ptr and weak_ptr

当一份资源需要被多个对象共享时可能就需要使用 shared_ptr, 相比于 unique_ptr, shared_ptr 的实现更为复杂

标准库的 shared_ptr 实现为用原子操作实现的线程安全, test 中实现了一个简易的非线程安全的 shared_ptr 用以观察 shared_ptr 的大致结构

参考代码见: shared_ptr

大致结构以 test 中 shared_ptr 为例

1. 前提

OBTUSE

(a) shared_weak_ptr 内部存储了两个指针, 本身是以栈对象的形式存在而不是指针, 通过运算符重载 (->) 使其 在且仅在使用资源指针的时候, 智能指针本身能够像一个指针那样 (包括 unique_ptr). 这体现在智能指针的使用方式上, 对于智能指针本身的数据成员和成员函数需要使用 (.) 调用, 对于资源指针指向的对象直接使用 (->) 访问

(b) 明确一点, 只有使用一个 new 表达式来构造一个新的堆对象或者使用 make_shared 创建才会产生一个全新的指针对

(1. 资源指针, 指向一个堆资源 |2. 计数器指针, 指向一个用以记录指向该资源的 shared_ptr, weak_ptr 分别为多少的计数器)

除此之外无论是使用 shared_ptr 或者 weak_ptr 来初始化 shared_ptr 或者初始化 weak_ptr 实质上都没有产生新的资源, 只是不断的增加计数器中对应 weak 或者 shared 的数量

(c) weak_ptr 与 shared_ptr 的区别在于前者只是后者的辅助, 这体现在:

i. weak_ptr 不能使用 new 表达式来构造一个新的堆对象, 只能依附在 shared_ptr 上

ii. weak_ptr 与 shared_ptr 分开计数, 一个堆对象是否释放取决于指向它的 shared_ptr 的数量而与 weak_ptr 无关

iii. 不同的是, 计数器释放需要两个指针数量都为 0

iv. 基于上两条, 想使用一个 weak_ptr 访问对象需要先检测计数器中 shared 的数量是否为 0, 才能确定资源是否被释放, 接下来使用 weak_ptr 访问资源的通用做法是将其擢升为一个 shared_ptr(即使用 shared 中对应 weak 的构造函数, 这会增加 shared 的计数)

v. 上一条末先将 weak_ptr 转为 shared_ptr 再使用的原因在于为了明确两者间的关系 (前者只是后者的辅助), 不对 weak_ptr 进行运算符重载 (事实上资源指针作为 private 成员, 在重载运算符 (->) 中返回其本身), 也意味着 weak_ptr 无法访问资源, 兴须转为 shared_ptr

```
bool expired() const noexcept { return cnt == nullptr || cnt->
    use_count() == 0; }
//cnt为计数器指针, use_count()返回shared数量
shared_ptr<Ep> lock() const noexcept{
    return expired()?shared_ptr<Ep>() : shared_ptr<Ep>(*this);
}
```

2. Counter

(a) 保存指向同一资源的 shared 和 weak 数量

3. shared_ptr

(a) shared_ptr 使用引用计数来维护在多个 shared_ptr 指向同一资源时依旧能正确释放资源

(b) 引用计数使用指针 (计数器类) 的方式来保证各 shared_weak_ptr 之间正确访问

(c) 构造方式

i. 直接传入一个返回指针的 new 表达式

- 创建一个计数器, shared_count 初始化为 1, weak_count 初始化为 0

ii. 拷贝另一个 shared_ptr 或拷贝一个 weak_ptr
- 增加 shared 计数

iii. move 另一个 shared_ptr
- 将原 shared_ptr 的资源指针和计数指针置空

iv. make_shared 返回一个 shared_ptr 型临时变量

4. weak_ptr

(a) 构造方式

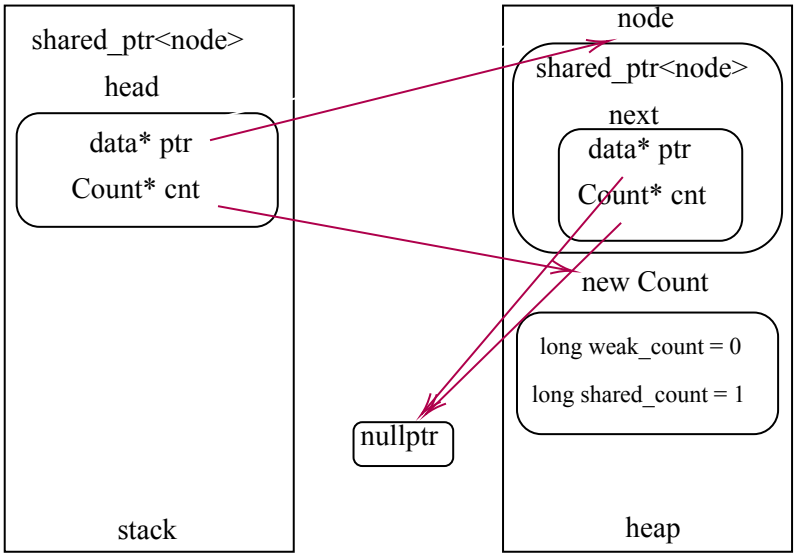
i. 拷贝另一个 weak_ptr 或拷贝一个 shared_ptr
- 增加 weak 计数

ii. move 另一个 shared_ptr
- 将原 weak_ptr 的资源指针和计数指针置空

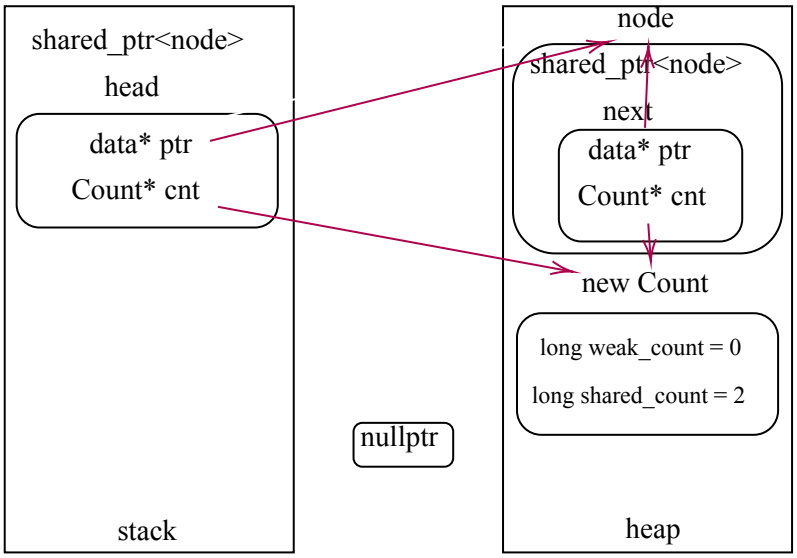
shared_ptr 存在的问题:

1. 环形引用
- 观察这样一个链表结构
- 测试程序见 smart_ptr 文件夹

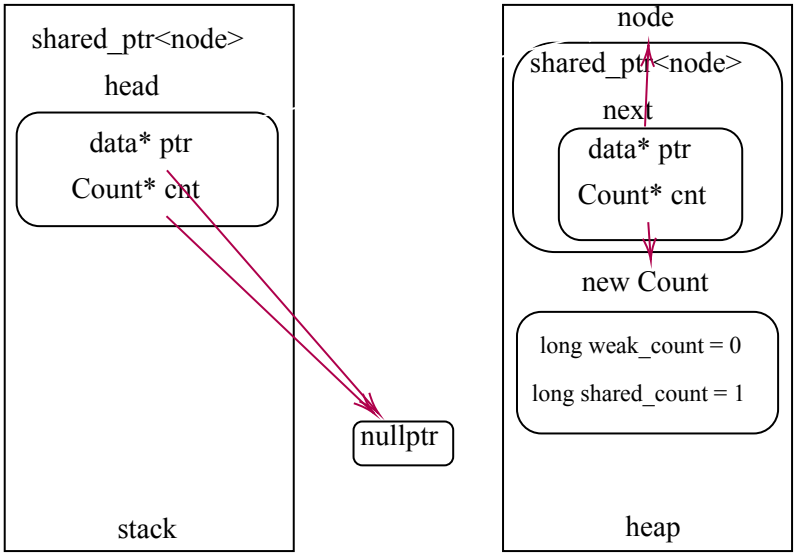
```
struct node{
    shared_ptr<node> next;
};
shared_ptr<node> head; (1)
head->next = head;     (2)
head.reset();          (3)   见test
```



(1)

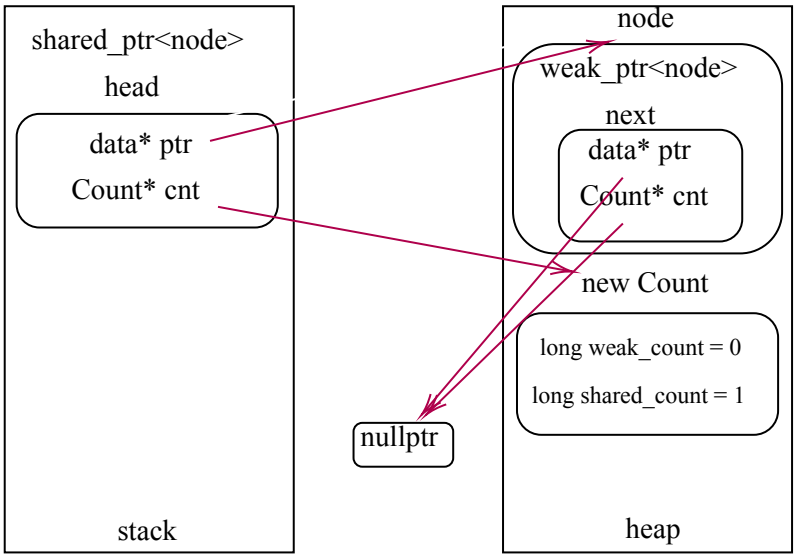


(2)

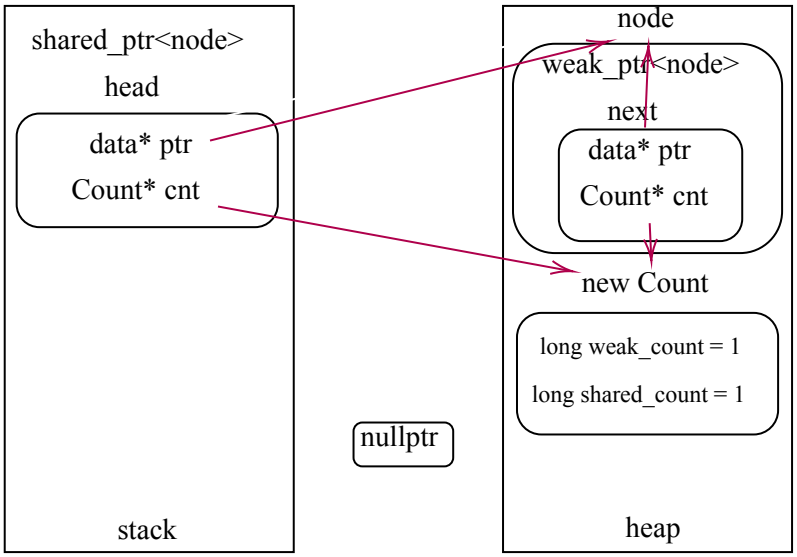


(3)

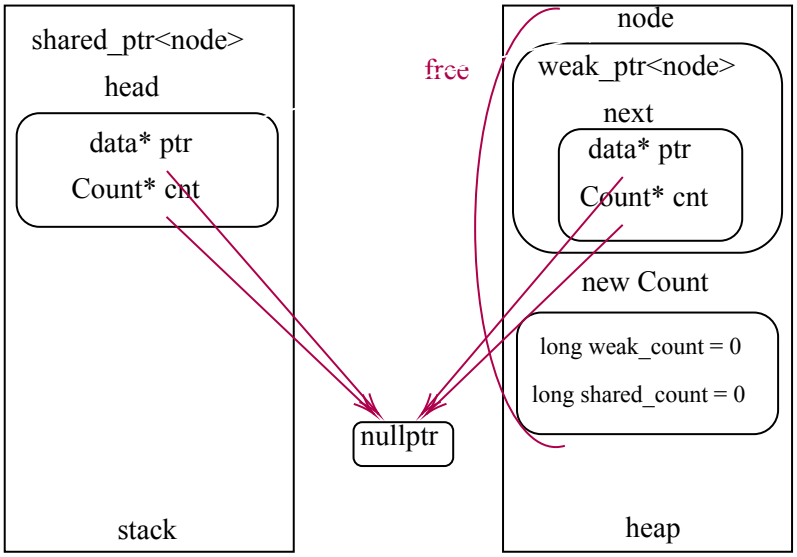
- 可以看出引用计数一直在正常工作, 问题在于当栈上 head 主动调用 reset 进行析构, 或者被动析构时, 此时引用计数为 2, 因此不会释放 node, 也就不会释放 next, 而位于堆上的 next 不会自动析构, 就造成了内存泄露
- 当一个含有 shared_ptr 的对象被构造在堆上, 那么 shared_ptr 本身也就在堆上, 若它指向另一个堆对象, 则该堆对象析构同步于 shared_ptr 本身析构, 而 shared_ptr 本身析构同步于自身所处的对象析构, 则若这两个对象是同一对象, 或者更广泛的连成一个环, 它们的析构是同步的, 那么就只能主动调用其中一个 shared_ptr 进行析构才能释放, 此时这个 shared_ptr 与普通指针并无区别, 一样需要主动 delete, 失去了它作为智能指针的意义
- 此时我们引入 weak_ptr, weak_ptr 只能指向 shared_ptr 指向的对象, 而它自身不实际拥有任何对象


shared_ptr<node>
head
data* ptr
Count* cnt
stack
node
weak_ptr<node>
next
data* ptr
Count* cnt
new Count
long weak_count = 0
long shared_count = 1
heap
nullptr

(1)


shared_ptr<node>
head
data* ptr
Count* cnt
stack
node
weak_ptr<node>
next
data* ptr
Count* cnt
new Count
long weak_count = 1
long shared_count = 1
heap
nullptr

(2)


shared_ptr<node>
head
data* ptr
Count* cnt
stack
free
node
weak_ptr<node>
next
data* ptr
Count* cnt
new Count
long weak_count = 0
long shared_count = 0
heap
nullptr

(3)

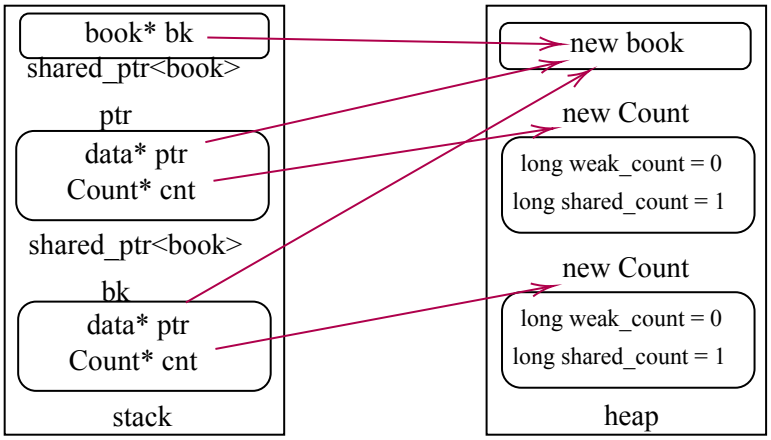
2. 返回对象本身的 shared_ptr

- 测试程序见 part4: GetAsMyObj

```
struct book{
    void getshared(student* stu){
        stu->getnewbook(shared_ptr<book>(*this));
    }
};
struct student{
    void getnewbook(shared_ptr<book> bp){
        ...
    }
};
main:
    student* stu = new student;
    shared_ptr<book> ptr(new book);
    ptr->getshared(stu);
```

- 如果试图在类内创建一个该对象智能指针并传递给另一个对象, 理论上讲此时这个 book 应该引用计数为 2 , 但观察两次 book 创建都是使用指针, 而指针初始化一个 shared_ptr 是会出现一个全新的指针对, 这个全新体现在计数器全新, 而资源指针实际上指向同一块资源. 上述代码等同于:

```
book* bk = new book;
shared_ptr<book> ptr(bk);
student::shared_ptr<book> bp(bk);
```



- 这会释放两次资源, 因此事实上需要的返回操作是在对象内部返回定义在外部的智能指针, 一种实现方法是引入 enable_shared_from_this

1.4.7.4 enable_shared_from_this

TODO: 侵入式

一种容易想到的方法是在每一个需要返回对象智能指针的类中定义一个智能指针, 在构造一个 shared_ptr 的时候就需要初始化类中的这个智能指针, 基于前文, 这个指针应该为 weak_ptr

更一般的, 让需要返回对象智能指针的类 public 继承一个 enable_shared_from_this, 在这个类中有一个 weak_ptr 用于进行上述相同的操作

```
templte<class Tp> class enable_shared_from_this
private:
    mutable weak_ptr<Tp> weak_this;
public:
    template <class Up> friend class shared_ptr;

    shared_ptr<Tp> shared_from_this()
    { return shared_ptr<Tp>(weak_this); }

struct book : enable_shared_from_this{
    void getshared(student* stu){
        stu->getnewbook(shared_from_this());
    }
};
```

而为了使用这个 weak_ptr, 需要在构造这个对象的同时, 对其中的 weak_ptr 进行构造, 可能需要一个类似这样的构造版本

```
explicit shared_ptr(Tp* p , bool need_shared = false)
:ptr(p),cnt(new Counter)
{ if(need_shared) p->weak_this=(*this); }//shared_ptr是
   enable_shared_from_this的友元
```

更一般的, 可以进行自动判断一个类是否继承了 enable_shared_from_this, 若继承, 则自然有 weak_this

上述代码就可更改为

```
explicit shared_ptr(Ep* e) :ptr(e),cnt(new Counter) { enable_weak_this(e,
   e); }

template<class Up, class Ptr>
enable_if_t<is_convertible_v<
    Ptr*, const enable_shared_from_this<Up>*
    >, /*SFINAE判断两种类型能否转换*/
void> enable_weak_this(const enable_shared_from_this<Up>* ep, Ptr* p)
```

OBTUSE

```
    noexcept{
     using rawUp = remove_cv_t<Up>;
     if(ep && ep->weak_this.expired()){
         ep->weak_this = shared_ptr<rawUp>(
             *this, const_cast<rawUp*>(p)
         );
     }
}//转换不成功则匹配下面这个版本，接受任意参数，不做任何行为
void enable_weak_this(...) noexcept {}
```

1.4.8 noexcept

1.4.9 SFINAE