

C++ Advanced 学习笔记

郑华

2019 年 5 月 6 日

目录

目录	3
第一章 内存管理	9
1.1 内存管理详解	9
1.1.1 内存分配方式	9
1.1.2 控制内存分配	11
1.1.3 指针参数是如何传递内存的	11
1.1.4 杜绝野指针	12
1.1.5 内存耗尽怎么处理	13
1.2 健壮指针和资源管理	13
1.2.1 RAII	13
1.2.2 智能指针	13
1.3 内存分配	14
1.3.1 内存空间分配示例	14
1.3.2 内存管理基础要素	14
1.3.3 malloc 申请空间布局结构	19
1.3.4 embedded pointers	20
1.4 STL 内存分配	21
1.5 STL 源码结构	23
1.6 内存泄漏	30
1.6.1 如何对付内存泄漏?	30
1.6.2 内存泄漏的发生方式	30
1.6.3 C/C++ 内存泄漏及其检测工具	32
1.6.4 检测内存泄漏	32
1.7 内存回收	32
1.7.1 内存对象大会战	32
1.7.2 垃圾回收方法	32
1.8 内存屏障	33
第二章 装载链接原理	35
2.1 CPU 体系	35

2.2	跨平台原理	35
2.3	C 例子-> 编译器与链接器	36
2.3.1	C 源文件	36
2.3.2	目标文件	37
2.3.3	目标文件的链接	38
2.4	执行期间-> 装载器程序	39
2.4.1	动态链接库	39
2.4.2	Main 函数之前	41
第三章	正则表达式	43
3.1	基础知识	43
3.1.1	整个字符串是否匹配	43
3.1.2	只返回一个匹配结果	43
3.1.3	返回多个匹配结果	43
3.2	子表达式匹配	44
第四章	异常处理	47
4.1	简介	47
4.2	异常处理机制	48
4.2.1	异常再引发	48
4.2.2	栈展开	48
4.2.3	未处理异常	49
4.2.4	描述函数可否引发异常	50
4.2.5	显示异常名字	50
4.2.6	异常标准库类结构	51
4.3	logic_error	52
4.3.1	invalid_argument	52
4.3.2	domain_error	53
4.3.3	length_error	53
4.3.4	out_of_range	53
4.3.5	future_error	53
4.3.6	bad_optional_access	53
4.4	runtime_error	54
4.4.1	range_error	54
4.4.2	overflow_error	54
4.4.3	underflow_error	54
4.4.4	regex_error	54
4.4.5	system_error	55
4.4.6	tx_exception	56
4.5	bad errors	56

4.5.1	bad_typeid	56
4.5.2	bad_cast	56
4.5.3	bad_weak_ptr	57
4.5.4	bad_function_call	57
4.5.5	bad_alloc	57
4.5.6	bad_exception	58
4.5.7	bad_variant_access	59
第五章	多线程	61
5.1	参考	61
5.2	须知	61
5.2.1	线程安全	61
5.2.2	对象的创建	62
5.3	概念	62
5.4	POSIX 线程	63
5.4.1	线程创建	63
5.4.2	线程 ID	64
5.4.3	线程属性	64
5.4.4	线程撤销	64
5.4.5	线程局部存储	64
5.4.6	线程清除	64
5.5	C++11 线程	64
5.5.1	头文件	64
5.5.2	线程类	64
5.5.3	线程间数据交互和数据争用 (Data Racing)	69
5.5.4	互斥锁	72
5.5.5	条件变量	76
5.5.6	期许与承诺	77
5.5.7	线程池	85
5.6	References	89
5.6.1	thread	89
5.6.2	atomic	95
5.6.3	mutex	95
5.6.4	condition_variable	101
5.6.5	Futures	105
第六章	多进程	113
第七章	泛型编程	115
7.1	decltype	115

7.2	完美转发	115
7.3	前言须知	116
7.4	函数模版	117
7.4.1	定义及使用	117
7.4.2	声明模版函数	117
7.4.3	模版也可以重载	117
7.4.4	实参的演绎-模版类型推导确定	117
7.4.5	定制非模版函数(重载函数模版)	118
7.5	函子	119
7.5.1	函数指针实现	119
7.5.2	函子	119
7.6	类模板	120
7.6.1	定义类模板	120
7.6.2	使用类模板	120
7.6.3	类模板别名	121
7.6.4	类模板显示特化	121
7.7	模版参数	121
7.7.1	非类型模版参数	121
7.7.2	默认模版参数	121
7.7.3	模版类型的模版参数	122
7.7.4	typename	122
7.7.5	模板类中再有模版成员	122
7.8	模版特化	123
7.8.1	全特化 Full Specialization	123
7.8.2	偏特化 Partial Specialization	123
7.9	模版友元	123
7.10	元编程	124
7.11	shared_ptr 实现原理	124
7.11.1	源码	124
7.11.2	分析	126
7.12	weak_ptr 实现原理	129
7.12.1	使用示例	129
7.12.2	实现原理	130
7.13	enable_shared_from_this	136
7.13.1	使用场合	136
7.13.2	使用场景分析	137
7.13.3	实现原理分析	138
7.14	sharedCount、weakCount	140
7.14.1	count base	140

7.14.2	shared count	144
7.14.3	weak Count	147
7.15	SFINAE and enable_if	149
7.16	参考	150
第八章	Effective	151
8.1	Effective C++	151
8.1.1	C++ 基本相关性能提升	151
8.1.2	C++ 构造/析构/赋值性能提升	152
8.1.3	资源管理	154
8.1.4	设计与声明	155
8.1.5	实现	155
8.1.6	继承与面向对象设计	156
8.2	More Effective C++	156
8.2.1	基础议题 (Basics)	156
8.2.2	操作符 (Operators)	157
8.2.3	异常 (Exceptions)	159
8.2.4	效率 (Efficiency)	160
8.2.5	技术 (Techniques,Idioms,Patterns)	161
8.2.6	杂项讨论 (Miscellany)	161
8.3	Effective Modern C++	162
8.3.1	Deducing Types	162
8.3.2	auto	162
8.3.3	Moving to Modern C++	162
8.3.4	Smart Pointers	162
8.3.5	Rvalue References, Move Semantics, and Perfect Forwarding	162
8.3.6	Lambda Expressions	162
8.3.7	The Concurrency API	162
8.3.8	Tweaks	162
8.4	Effective STL	162
8.4.1	容器	162
8.4.2	vector And string	163
8.4.3	关联容器	163
8.4.4	迭代器	164
8.4.5	算法	164
8.4.6	函数子、函数子类、函数及其他	165
8.4.7	在程序中使用 STL	166
8.5	提升 C++ 编程性能的技术	167
8.5.1	跟踪范例	167

8.5.2	虚函数	168
8.5.3	临时对象	168
8.5.4	内存池	168
8.5.5	引用计数	170
8.5.6	循环引用	172
8.5.7	代码优化	175
8.5.8	设计优化	175
8.5.9	可伸缩性	175
8.5.10	系统体系结构相关性	175
8.6	深入探索 C++ 对象模型	175
8.7	STL 源码剖析	175
8.8	参考	175
第九章	OpenMP 并行技术	177
第十章	GPU 并行技术	179
10.1	OpenCL	179
10.2	CUDA	179

第一章 内存管理

参考文献: <https://blog.csdn.net/zhanghefu/article/details/5003407>

1.1 内存管理详解

1.1.1 内存分配方式

虚拟内存布局

- **PCB** 保存进程 id、文件描述符等。
- **栈**, 在执行函数时, 函数内局部变量的存储单元都可以在栈上创建, 函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中, 效率很高, 但是分配的内存容量有限。
- **内存映射区**父子进程共享。
- **堆**, 就是那些由new分配的内存块, 他们的释放编译器不去管, 由我们的应用程序去控制, 一般一个new就要对应一个delete。如果程序员没有释放掉, 那么在程序结束后, 操作系统会自动回收。
- **自由存储区**, 就是那些由malloc等分配的内存块, 他和堆是十分相似的, 不过它是用free来结束自己的生命的。
- **全局/静态存储区**, 全局变量和静态变量被分配到同一块内存中, 在以前的C语言中, 全局变量又分为初始化的和未初始化的, 在C++里面没有这个区分了, 他们共同占用同一块内存区。
- **常量存储区**, 这是一块比较特殊的存储区, 他们里面存放的是常量, 不允许修改。

堆栈区分问题 void f() { int* p=new int[5]; }

这条短短的一句话就包含了堆与栈, 看到new, 我们首先就应该想到, 我们分配了一块堆内存, 那么指针 **p** 呢? 他分配的是一块栈内存, 所以这句话的意思就是: 在栈内存中存放了一个指向一块堆内存的指针 **p**。

在程序会先确定在堆中分配内存的大小, 然后调用 *operator new* 分配内存, 然后返回这块内存的首地址, 放入栈中, 他在VC6下的汇编代码如下:

```
00401028 push 14h  
0040102A call operator new (00401060)  
0040102F add esp,4
```

```
00401032 mov dword ptr [ebp-8],eax  
00401035 mov eax,dword ptr [ebp-8]  
00401038 mov dword ptr [ebp-4],eax
```

这里，我们为了简单并没有释放内存，那么该怎么去释放呢？是 `delete p` 吗？澳，错了，应该是 `delete []p`，这是为了告诉编译器：我删除的是一个数组，VC6 就会根据相应的 Cookie 信息去进行释放内存的工作。

堆栈区别 堆栈的主要区别有以下 6 点：

- **管理方式不同：**对于栈来讲，是由编译器自动管理，无需我们手工控制；对于堆来说，释放工作由程序员控制，容易产生 memory leak。
- **空间大小不同：**一般来讲在 32 位系统下，堆内存可以达到 4G 的空间，从这个角度来看堆内存几乎是没有什么限制的。但是对于栈来讲，一般都是有一定的空间大小的，例如，在 VC6 下面，默认的栈空间大小是 1M（好像是，记不清楚了）。当然，我们可以修改：打开工程，依次操作菜单如下：*Project->Setting->Link*，在 *Category* 中选中 *Output*，然后在 *Reserve* 中设定堆栈的最大值和 *commit*。
- **能否产生碎片不同：**对于堆来讲，频繁的 `new/delete` 势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题，因为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出，在他弹出之前，在他上面的后进的栈内容已经被弹出
- **生长方向不同：**对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长。
- **分配方式不同：**堆都是动态分配的，没有静态分配的堆。栈有 2 种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由 `alloca` 函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需我们手工实现。
- **分配效率不同：**栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是 C/C++ 函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。

从这里我们可以看到，堆和栈相比，由于大量 `new/delete` 的使用，容易造成大量的内存碎片；由于没有专门的系统支持，效率很低；由于可能引发用户态和核心态的切换，内存的申请，代价变得更加昂贵。所以栈在程序中是应用最广泛的，就算是函数的调用也利用栈去完成，函数调用过程中的参数，返回地址，*EBP* 和局部变量都采用栈的方式存放。所以，我们推荐大家尽量用栈，而不是用堆。

虽然栈有如此众多的好处，但是由于和堆相比不是那么灵活，有时候分配大量的内存空间，还是用堆好一些。

无论是堆还是栈，都要防止越界现象的发生（除非你是故意使其越界），因为越界的结果要么是程序崩溃，要么是摧毁程序的堆、栈结构，产生以想不到的结果，就算是在你的程序运行过程中，没有发生上面的问题，你还是要小心，说不定什么时候就崩掉，那时候 debug 可是相当困难的：）

1.1.2 控制内存分配

由于内存的限制，频繁的动态分配不定大小的内存会引起很大的问题以及堆破碎的风险。一个防止堆破碎的通用方法是从不同固定大小的内存池中分配不同类型的对象。对每个类重载 new 和 delete 就提供了这样的控制。

实现：C++_Advanced 内存池实现，针对不同的对象创建不同的内存管理池

1.1.3 指针参数是如何传递内存的

如果函数的参数是一个指针，不要指望用该指针去申请动态内存。如下示例中，Test 函数的语句 GetMemory(str, 200) 并没有使 str 获得期望的内存，str 依旧是 NULL，为什么？

```
void GetMemory( char *p, int num)
{
    p = (char *) malloc( sizeof( char ) * num );
}

void Test( void )
{
    char *str = NULL;
    GetMemory( str, 100 ); // str 仍然为 NULL
    strcpy( str, "hello" ); // 运行错误
}
```

毛病出在函数 GetMemory 中。编译器总是要为函数的每个参数制作临时副本，指针参数 p 的副本是 _p，编译器使 _p = p。如果函数体内的程序修改了 _p 的内容，就导致参数 p 的内容作相应的修改。这就是指针可以用作输出参数的原因。

在本例中，_p 申请了新的内存，只是把 _p 所指的内存地址改变了，但是 p 丝毫未变。所以函数 GetMemory 并不能输出任何东西。事实上，每执行一次 GetMemory 就会泄露一块内存，因为没有用 free 释放内存。

如果非得要用指针参数去申请内存，那么应该改用“指向指针的指针”，见示例：

```
void GetMemory2( char **p, int num )
{
    *p = (char *) malloc( sizeof( char ) * num );
}
```

由于“指向指针的指针”这个概念不容易理解，我们可以用函数返回值来传递动态内存。这种方法更加简单，见示例：

```
char *GetMemory3( int num )
{
    char *p = (char *) malloc( sizeof(char) * num );
    return p;
}
```

用函数返回值来传递动态内存这种方法虽然好用，但是常常有人把 `return` 语句用错了。这里强调不要用 `return` 语句返回指向“栈内存”的指针，因为该内存函数结束时自动消亡，见示例：

```
char *GetString( void )
{
    char p[] = "hello world";
    return p; // 编译器将提出警告
}
```

用调试器逐步跟踪 Test4，发现执行 `str = GetString` 语句后 `str` 不再是 `NULL` 指针，但是 `str` 的内容不是“hello world”而是垃圾。

如果把上述示例改写成如下示例，会怎么样？

```
char *GetString2( void )
{
    char *p = "hello world";
    return p;
}
```

函数 Test5 运行虽然不会出错，但是函数 `GetString2` 的设计概念却是错误的。因为 `GetString2` 内的“hello world”是常量字符串，位于静态存储区，它在程序生命期内恒定不变。无论什么时候调用 `GetString2`，它返回的始终是同一个“只读”的内存块。

1.1.4 杜绝野指针

“野指针”不是 `NULL` 指针，是指向“垃圾”内存的指针。人们一般不会错用 `NULL` 指针，因为用 `if` 语句很容易判断。但是“野指针”是很危险的，`if` 语句对它不起作用。“野指针”的成因主要有两种：

指针变量没有被初始化 任何指针变量刚被创建时不会自动成为 `NULL` 指针，它的缺省值是随机的，它会乱指一气。所以，指针变量在创建的同时应当被初始化，要么将指针设置为 `NULL`，要么让它指向合法的内存。例如

```
char *p = NULL;
char *str = (char *) malloc(100);
```

指针 `p` 被 `free` 或者 `delete` 之后 没有置为 `NULL`，让人误以为 `p` 是个合法的指针。

指针操作超越了变量的作用域范围 这种情况让人防不胜防，示例程序如下：

```
class A
{
public:
    void Func(void){ cout << "Func of class A" << endl; }

    void Test(void)
    {
        A *p;
        {
            A a;
            p = &a; // 注意 a 的生命周期
        }
        p->Func(); // p 是“野指针”
    }
}
```

函数 Test 在执行语句 p->Func() 时，对象 a 已经消失，而 p 是指向 a 的，所以 p 就成了“野指针”。但奇怪的是我运行这个程序时居然没有出错，这可能与编译器有关。

1.1.5 内存耗尽怎么处理

- 判断指针是否为NULL，如果是则马上用return语句终止本函数。
- 判断指针是否为NULL，如果是则马上用exit(1) 终止整个程序的运行。
- 为 new 和 malloc 设置异常处理函数。例如 Visual C++ 可以用_set_new_hander 函数为 new 设置用户自己定义的异常处理函数，也可以让 malloc 享用与 new 相同的异常处理函数。详细内容请参考 C++ 使用手册。

1.2 健壮指针和资源管理

1.2.1 RAII

在构造函数中分配资源，在析构函数中释放资源。

1.2.2 智能指针

见 C++_Basic_C11

循环引用问题 ->

引用计数shared_ptr

循环引用解决办法weak_ptr

1.3 内存分配

1.3.1 内存空间分配示例

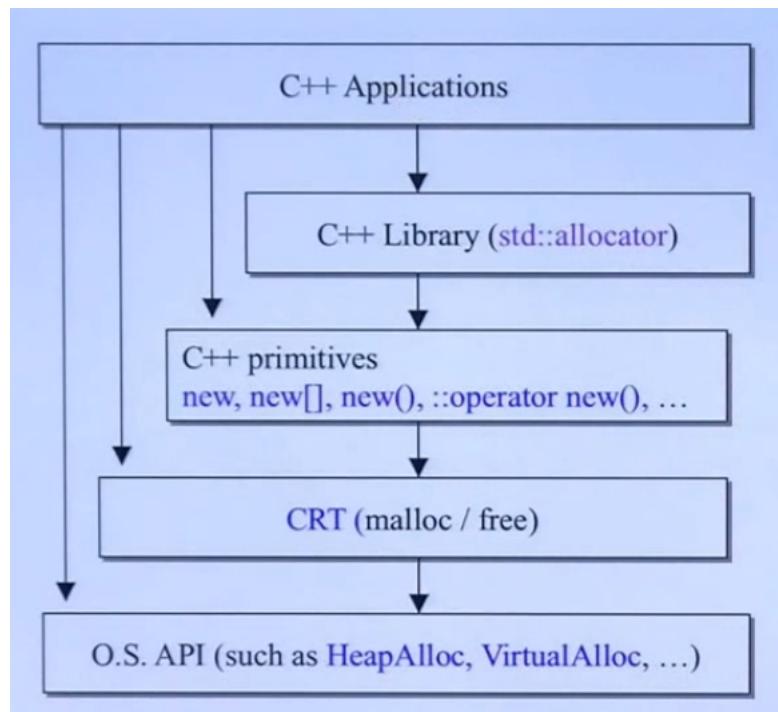


图 1.1: Mem Process

1.3.2 内存管理基础要素

表 1.1: C++ Memory Primitives

分配	释放	类型	可否重载
malloc()	free()	C 函数	不可
new	delete	C++ 表达式	不可
::operator new()	::operator delete()	C++ 函数	可
allocator<T>::allocate()	allocator<T>::deallocate()	C++ 标准库	可自由设计

new ->

```
Complex* pc = new Complex(1,2);
|
|
|
|
Complex* pc;
try{
    void* mem = operator new( sizeof(Complex) ); // 1 申请内存
    pc = static_cast<Complex*>(mem); // 2 转型
    pc->Complex::Complex(1,2); // 3 调用构造函数
}
catch( std::bad_alloc )
{
    ...
}
```

operator new() ->

```
void* operator new( size_t size, const std::nothrow_t&) _THROW0()
{
    void *p;
    while((p = malloc(size)) == 0)
    {
        _TRY_BEGIN
            if(_callnewh(size) == 0) break; // new handle: 当无内存使用用户指定处理函数进行处理
        _CATCH(std::bad_alloc) return (0);
        _CATCH_END
    }
    return (p);
}
```

new(已分配内存指针 p) expression placement new ->

给已分配内存调用合理的构造函数进行初始化。

```
new(p) Complex(1,2);
```

array new, array delete ->

```
Complex* pca = new Complex[3]; // 唤起3次 constructor, 无法从参数获取初值  
  
delete [] pca; //唤起3次deconstructor, 如果没有写中括号, 那么会以为为普通指  
针, 只调用一次析构。
```

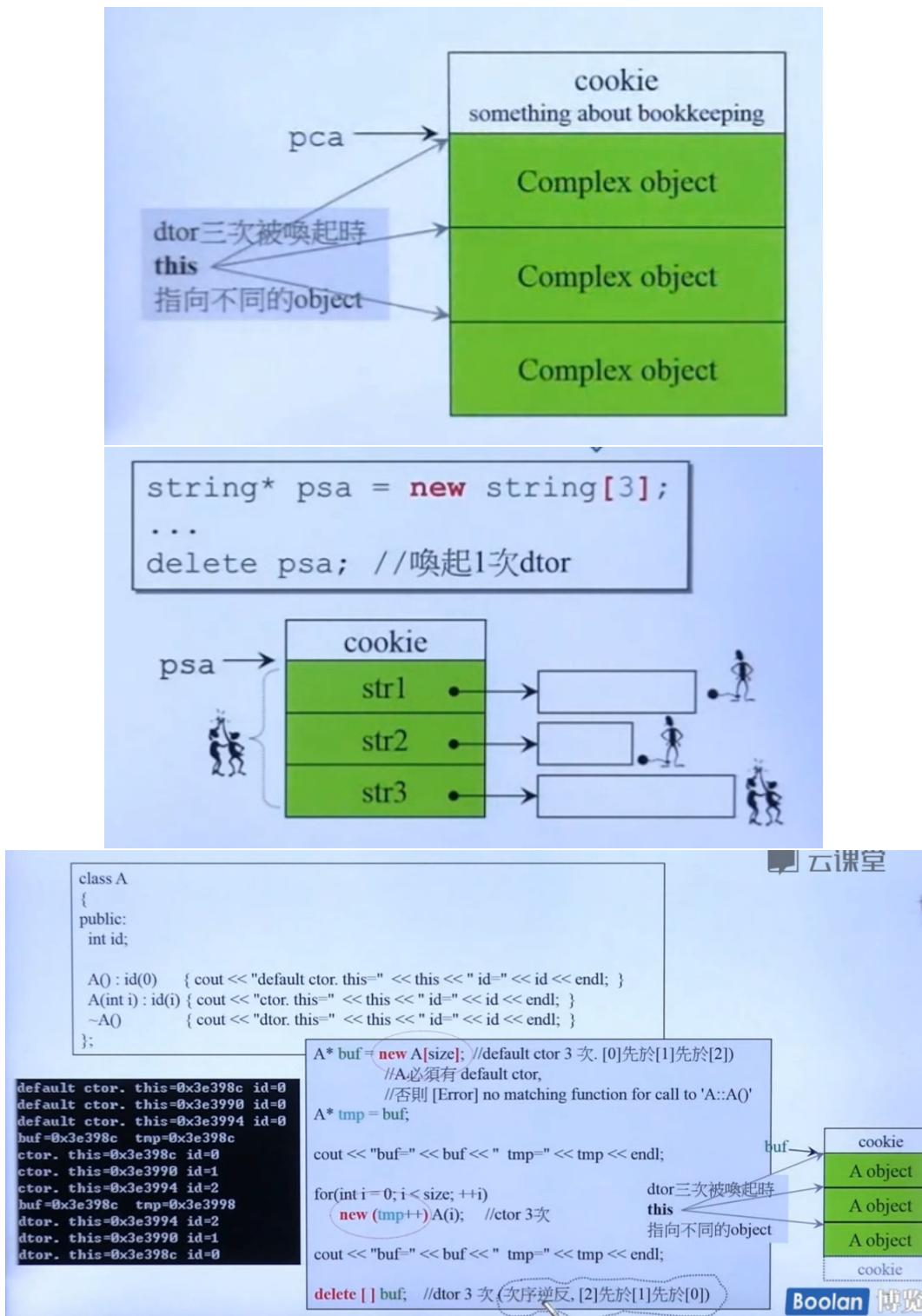


图 1.2: Array New Mem

cookie -> 负责记录申请数组的长度, 即调用delete 的次数

delete ->

```
Complex* pc = new Complex(1,2);  
...  
delete pc;  
|  
|_____  
|  
pc->~Complex(); // 先析构  
operator delete(pc); // 然后释放内存
```

operator delete(void *) ->

```
void __cdecl operator delete(void*p) __THROW0()  
{  
    free(p);  
}
```

总结-流程 示例->

```
Foo* p = new Foo(x);  
...  
delete p;
```

上述代码可具体表示为如下几个相关部分

1. **operator new** : Foo* p = (Foo*)operator new(sizeof(Foo));

notice: operator new -> Foo::operator new(size_t size) 即这里调用的是该类重载后的operator new，如果没有发生重载则调用原operator new,operator delete 相同。

2. **placement new** : new(p) Foo(x);

3. **deConstruct** : p->~Foo();

4. **operator delete** : operator delete(p);

allocator 将内存管理的细节隔离实现，简化不同类的编写效率与安全性。

```
class allocator
{
private:
    struct obj{
        struct obj* next; //embedded pointer
    }

public:
    void *allocate(size_t);
    void deallocate(void*, size_t);

private:
    obj* freeStore = nullptr;
    const int CHUNK = 5;
};

void allocator::deallocate(void* p, size_t)
{
    //将*p 收回插回freeList 前端
    ((obj*)p) ->next = freeStore;
    freeStore =(obj*)p;
}

// 什么类型的类都是如下使用allocator 的，因此会产生宏简化过程。
class Foo{
public:
    long L;
    string str;
    static allocator myAlloc;

public:
    Foo(long l):L(l){}
    static void* operator new(size_t size)
    {
        return myAlloc.allocate(size);
    }

    static void operator delete(void* pdead, size_t size)
    {
        return myAlloc.deallocate(pdead, size);
    }

};
allocator Foo::myAlloc;
```

marco for static allocator 简化上述 allocator 注入的代码量与程序员的工作量。

```
// Declare pool alloc
#define DECLARE_POOL_ALLOC() \
public: \
    void* operator new(size_t size){return myAlloc.allocate(size);} \
    void operator delete(void* p){return myAlloc.deallocate(p);} \
protected: \
    static allocator myAlloc;

// Implement pool alloc
#define IMPLEMENT_POOL_ALLOC(class_name) \
allocator class_name::myAlloc;

// class Foo 实现
class Foo{ \
    DECLARE_POOL_ALLOC() \
public: \
    long L; \
    string str; \
public: \
    Foo(long l):L(l){}
};

IMPLEMENT_POOL_ALLOC(Foo)

class Goo{ \
DECLARE_POOL_ALLOC() \
public: \
    complex<double> c; \
    string str; \
public: \
    Foo(complex<double>& x):c(x){}
};

IMPLEMENT_POOL_ALLOC(Goo)
```

new Handler 当内存分配失败的情况下使用 newHandler 进行处理，以释放空间满足分配或者终止程序。

1.3.3 malloc 申请空间布局结构

在使用 malloc 分配的内存中存在着大量的空余空间，如 cookie 信息等，基本结构如图1.3所示。

VC6 下的 malloc() 内存块布局

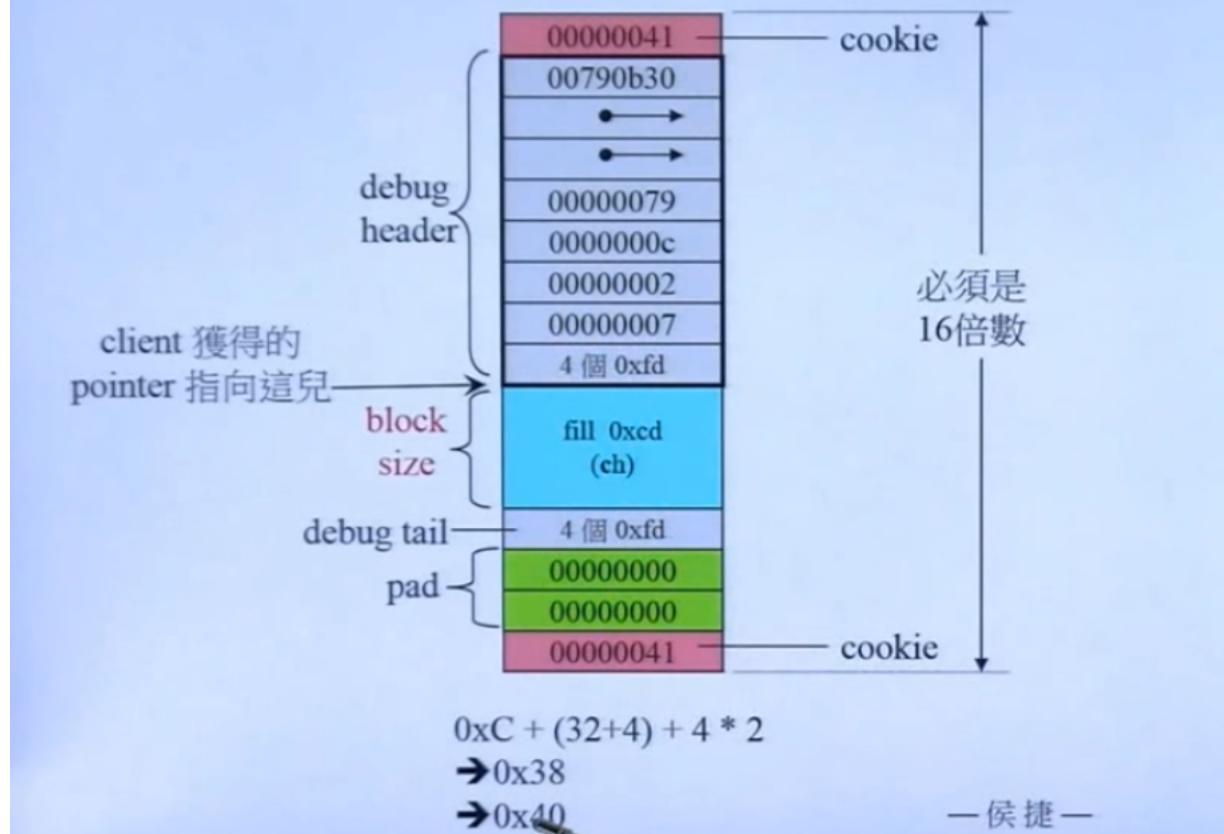


图 1.3: Malloc 结构图

1.3.4 embedded pointers

当客户端获得小区块，获得的即是 `char*`(指向某个 `union obj`). 此时虽然客户端没有如 `LString` 之类的信息可知区块大小，但由于这区块是给 `object` 所用，相等于 `object` 大小，`object Constructor` 自然不会过分。如图1.4所示。

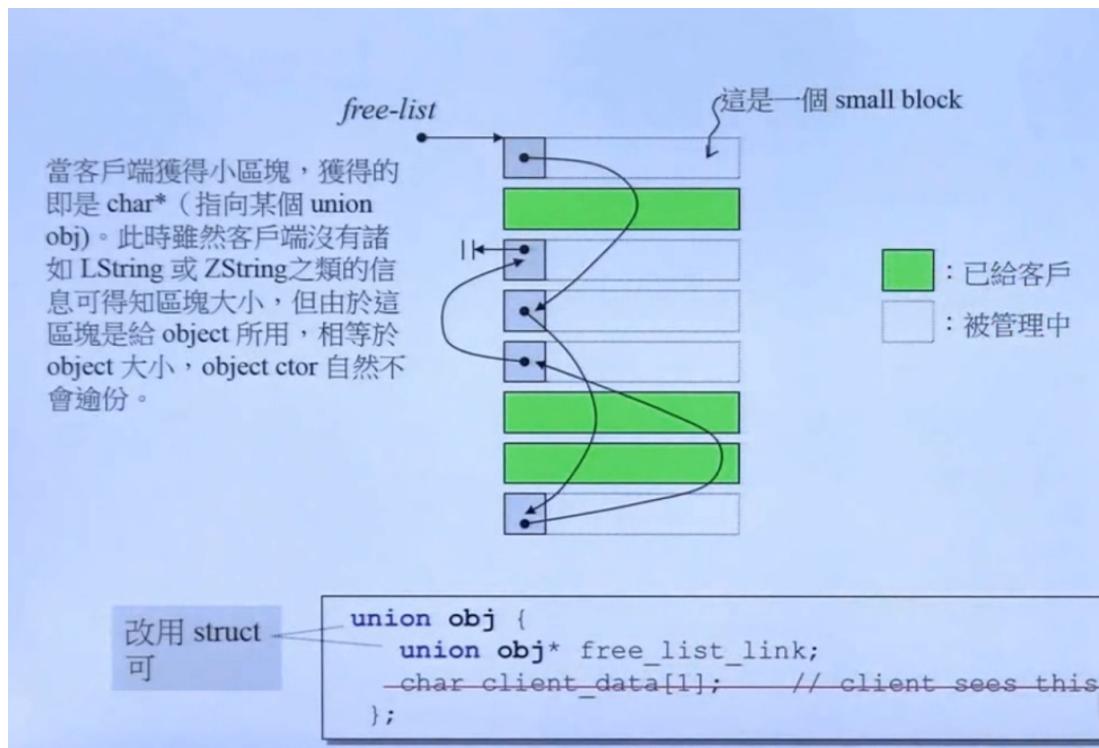


图 1.4: 嵌入指针

```
class Airplane{
private:
    // Data member 结构声明
    struct AirplaneRep{
        unsigned long miles;
        char type;
    };

private:
    union{
        AirplaneRep rep; // 此指针针对使用中的 objects
        Airplane* next; // 此指针针对内存池 freelist 上的 object
    }
};
```

1.4 STL 内存分配

STL freelist 何时释放 在看完源码后，知道了如何利用内存池进行管理内存，但是如果内存紧张，怎么知道 freelist 何时释放呢？析构？引用计数？

STL pool allocator 用法示例 STL(GNUC 4.9) 库中带有的内存池分配，但是标准的分配依旧是只调用 new expression。

```
#include <extend\pool_allocator.h>
```

```

template<typename Alloc>
void cookie_test(Alloc alloc, size_t n)
{
    typename Alloc::value_type *p1, *p2, *p3;
    p1 = alloc.allocate(n);
    p2 = alloc.allocate(n);
    p3 = alloc.allocate(n);

    cout<<"p1=" << p1 << '\t' <<"p2=" << p2 << '\t'<< endl;

    alloc.deallocate(p1, sizeof(typename Alloc::value_type));
    alloc.deallocate(p2, sizeof(typename Alloc::value_type));
    alloc.deallocate(p3, sizeof(typename Alloc::value_type));
}

// Test 主体
int main()
{
    // 使用标准内存池分配
    cout << sizeof(__gnu_cxx::__pool_alloc<int>) << endl;
    vector<int, __gnu_cxx::__pool_alloc<int>> vecPool;
    cookie_test(__gnu_cxx::__pool_alloc<double>(), 1);

    // 使用默认分配器分配
    cout << sizeof(std::allocator<int>) << endl;
    vector<int, std::allocator<int>> vecPool;
    cookie_test(std::allocator<double>(), 1);
}

```

原理概述 std::alloc->16 大小的链表负责分配 (0-8-16...-128bytes) 大小的内存管理池。超过 128 后，使用 malloc 进行内存管理。

战备

如果 x-size 的链表为空，且战备池已经占用，则申请 $20*(x\text{-size})*2$ 的内存池，否则使用 x-size 的对应指针指向该战备池，并在战备池中按照 x-size 进行切割分配管理，*2 的意图就是留相同的一份池用于战备，有则在内存池中分配。

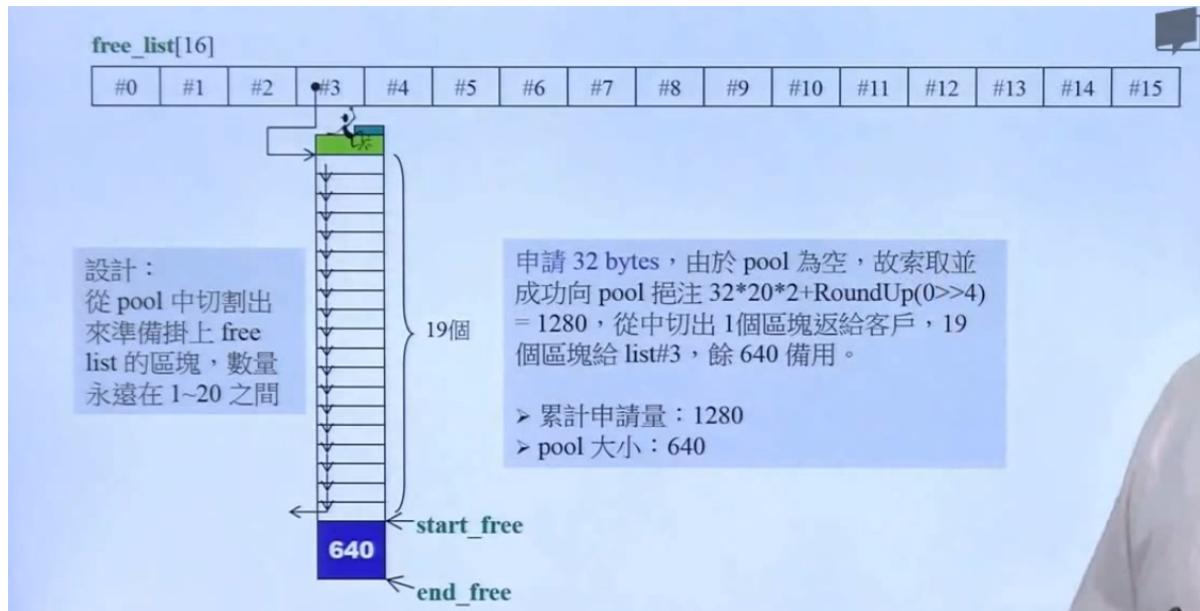


图 1.5: STD alloctor 架构图

在可用空间为 0 时，最多分配 20 个对象空间 *2 +RoundUp。
在剩余的战备空间不足一个对象时，就是堆碎片的处理。

堆碎片处理 剩余的碎片插入最接近的链表中。

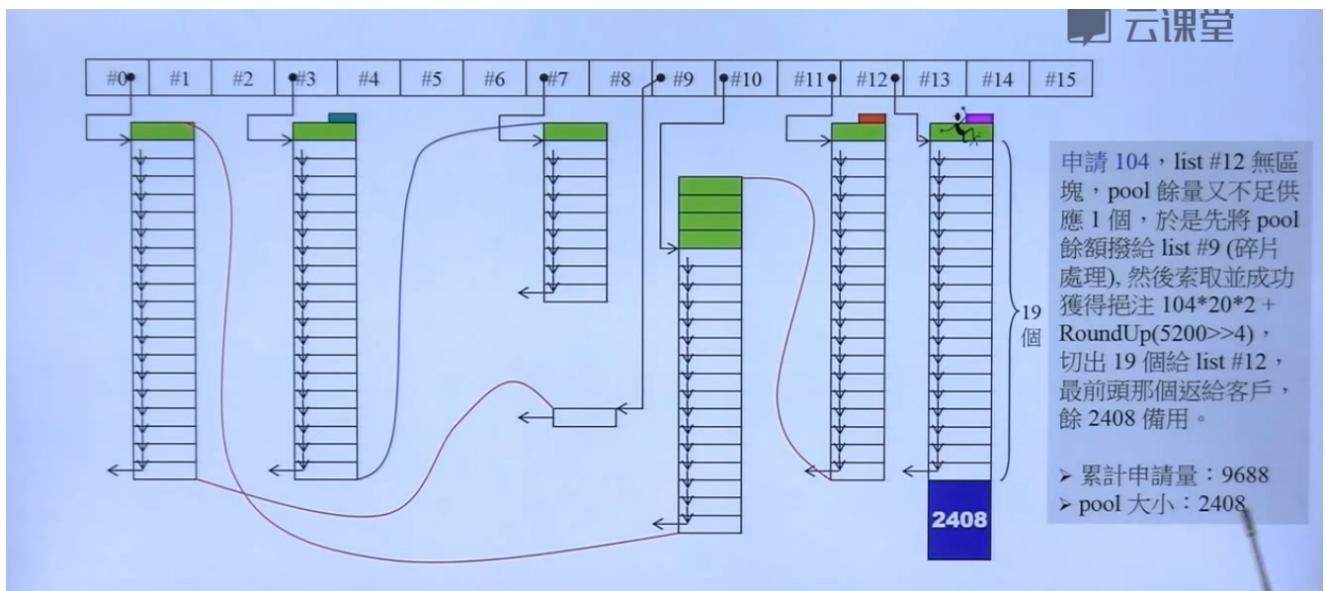


图 1.6: 堆碎片处理流程

内存不足解决 依次往右借战备池满足当下需求

1.5 STL 源码结构

SGI 的内存配置器分成两层：第一层直接调用 malloc，第二层是自己做了一个内存池来分配。下面说下第二层配置器的思路：

std::alloc 在设计时考虑到了以下几个方面：

- 向系统堆申请空间。
- 考虑多线程状态。
- 考虑内存不足时的措施。
- 考虑过多“小型区块”可能造成的内存碎片问题。

二级配置器内存申请过程：

- 首先判断是否小于 128bytes。
- 若小于则把申请长度填充至 8 的倍数。
- 利用 free_list+FREELIST_INDEX(n) 获取所需内存块在 free-lists 链表中的位置。
- 若申请失败则调用 refill 给 free-list 扩容。
- 若申请成功则将 free-list 的头节点保存之后指向下一个节点，返回保存的地址。

第二层配置器首先设置了一个 free_list[16] 的数组，数组中是指针，分别指向一块内存，指针类型定义如下：

```
union obj {
    union obj *free_list_link;
    char client_data[1];
};
```

总之，free_list 每个下标链接的内存大小为：8B、16B、24B、32B、40B、...、128B。共 16 个。

现在假如要分配一块 32B 的内存，但 free_list[3] 所指向的指针为 NULL，即空闲链表 free_list 中没有 32B 的内存，这时候就需要通过下面的 _S_chunk_alloc 来分配内存给 free_list 了。默认每次分配是分配 20 个大小为 32B 的内存。

即 _S_chunk_alloc 第二个参数传入值为 20，但系统不一定能分配 20 个，所以用的是引用。
_S_chunk_malloc 主要分三种情况来处理：

- 现有内存池容量满足你的需求：32B * 20 个，直接返回给你这么大一块内存；
- 现有内存池容量不能满足这么多个，即 20 个，但可以满足 1 个，那先给你 free_list 对应项分配一个 32B 再说；
- 现有内存池容量连一个都满足不了，那只能利用 malloc 从堆中分配内存。

从堆中分配内存时，首先把当前内存池中剩余的一些零碎内存赋给 free_list 中；

然后从堆中 malloc 内存，修改内存池的 _S_start_free、_S_end_free 指针。（这两个指针分别指向内存池的起始地址和结束地址）。

再然后递归调用 _S_chunk_malloc 函数。

```
class allocator : public __default_alloc_template
```

```
allocate() ~
```

```
static void* allocate(size_t __n)
{
```

```

void* __ret = 0;

if (__n > (size_t) _MAX_BYTES
    // 如果请求大于128字节，由一级内存配置器分配内存
    __ret = __mem_interface::allocate(__n);
else
    // 当内存请求小于等于128字节时
{
    _Obj* volatile* __my_free_list = _S_free_list
        + _S_freelist_index(__n);
    // 定位监管的 freelist

    _Obj* __restrict__ __result = *__my_free_list;
    if (__result == 0)
        // 如果 freelist 是空的，则通过调用 s_refill 分配 freelist
        __ret = _S_refill(_S_round_up(__n));
    else
        // 如果 freelist 有空闲内存块，则返回一个空闲块给客户
        // 同时监管 freelist 的头指针后移一个空闲块
    {
        *__my_free_list = __result -> _M_free_list_link;
        __ret = __result;
    }
}
return __ret;
};


```

refill() 函数原型: `void* __default_alloc_template::refill(size_t n)`

若 free-list 中没有可用区块时，则会调用 refill 为 free-list 填充新的空间。新的空间来自内存池，由 chunk_alloc() 申请。默认是取得 20 个新区块，但是如果内存池空间不足，则获得的区块数可能小于 20。如果 chunk_alloc() 返回一个区块，则直接返回，将该区块分配给调用者使用。free-list 不增加。若返回区块数大于 1，则从第二个区块开始扩展 free-list 链表，每个链表占 n 的长度。将第一个区块返回。

```

// Returns an object of size __n, and optionally adds to "size
// __n"'s free list. We assume that __n is properly aligned. We
// hold the allocation lock.

template<bool __threads, int __inst>
void*
__default_alloc_template<__threads, __inst>::__S_refill(size_t __n)
{
    int __nobjs = 20;
    // 默认20个，实际client需求的20倍
    char* __chunk = _S_chunk_alloc(__n, __nobjs);
    // 请求memory pool分配内存块，实际需求的20倍
}


```

```

    _Obj* volatile* __my_free_list;
    _Obj* __result;
    _Obj* __current_obj;
    _Obj* __next_obj;
    int __i;

    if (1 == __nobjs) return (__chunk);

    __my_free_list = _S_free_list + _S_freelist_index(__n);

    __result = (_Obj*)__chunk;
    //将返回给客户 20 块内存区块的第一块地址
    *__my_free_list = __next_obj = (_Obj*)(__chunk + __n);
    //修改监管 freelist 的首地址为第一个空闲内存块的地址

    for (__i = 1; ; __i++) {
        __current_obj = __next_obj;
        __next_obj = (_Obj*)((char *)__next_obj + __n);
        if (__nobjs - 1 == __i) {
            __current_obj -> _M_free_list_link = 0;
            break;
        } else {
            __current_obj -> _M_free_list_link = __next_obj;
        }
    }

    //上述 for 循环，是将 freelist 中的各结点串接起来。
}

return (__result);
}

```

chunk_alloc() 函数原型: `char* __default_alloc_template::chunk_alloc(size_t size,int& nobjs)`

其中 `nobjs` 是引用传参，是传入传出参数。表示的是 `chunk` 申请到的区块数，默认是 20。

函数先判断内存池中剩余的空间是否满足申请 20 个内存区块，若满足则直接返回。若不满足，则尽量申请多的区块，并利用 `nobjs` 的返回值标明申请的区块数量。若内存池剩余的空间不够申请 1 个内存区块，则先将这剩余的区块分配给适当的 free-list，然后再从系统堆中申请需求量的两倍加上附加量的内存空间，递归调用自己返回正确的 `nobjs` 值，由于 `nobjs` 最大为 20，剩余的未加入 free-list 的内存放入内存池中。若从堆中申请失败，则遍历整个 free-lists，看能否找到一个满足长度要求的内存区块。如果还是找不到，则调用一级配置器，也就是 `malloc()`，利用它的异常机制来处理，若未定义处理函数，则抛出 `bad_alloc` 异常。

```

template <bool __threads, int __inst>
char*
__default_alloc_template<__threads, __inst>::__S_chunk_alloc(size_t __size,
                                                               int& __nobjs)

```

```

{
    char* __result;
    size_t __total_bytes = __size * __nobjs;
    size_t __bytes_left = _S_end_free - _S_start_free; // 内存池剩余空间

    if (__bytes_left >= __total_bytes) { // 内存池剩余空间完全满足需求
        __result = _S_start_free;
        _S_start_free += __total_bytes;
        return(__result);
    } else if (__bytes_left >= __size) { // 内存池剩余空间不能完全满足需求，但足够供应一个(含)以上的区块
        __nobjs = (int)(__bytes_left/__size);
        __total_bytes = __size * __nobjs;
        __result = _S_start_free;
        _S_start_free += __total_bytes;
        return(__result);
    } else { // 内存池剩余空间连一个区块的大小都无法提供
        size_t __bytes_to_get =
2 * __total_bytes + _S_round_up(_S_heap_size >> 4);
        // Try to make use of the left-over piece.
        // 把内存池当前剩下的一些小残余零头利用一下。
        if (__bytes_left > 0) {
            _Obj* __STL_VOLATILE* __my_free_list =
                _S_free_list + _S_freelist_index(__bytes_left); //
                这里对吗？假如__bytes_left = 15，则
                _S_freelist_index(15) = 1,
                //
}

```

即
16B
的
位
置，
而
实
际
上
只
剩
下
了
15B
？

```
((_Obj*)_S_start_free) -> _M_free_list_link = *__my_free_list;
*__my_free_list = (_Obj*)_S_start_free;
```

```

    }

    _S_start_free = (char*)malloc(__bytes_to_get);
    if (0 == _S_start_free) {
        size_t __i;
        _Obj* __STL_VOLATILE* __my_free_list;
        _Obj* __p;
        // Try to make do with what we have. That can't
        // hurt. We do not try smaller requests, since that tends
        // to result in disaster on multi-process machines.
        for (__i = __size;
            __i <= (size_t) __MAX_BYTES;
            __i += (size_t) __ALIGN) {
            __my_free_list = _S_free_list + _S_freelist_index(__i);
            __p = *__my_free_list;
            if (0 != __p) {
                *__my_free_list = __p -> _M_free_list_link;
                _S_start_free = (char *)__p;
                _S_end_free = _S_start_free + __i; // 这里确定每一块内存大小都是__i吗？
                return (_S_chunk_alloc(__size, __nobjs));
                // Any leftover piece will eventually make it to the
                // right free list.
            }
        }
        _S_end_free = 0; // In case of exception.
        _S_start_free = (char*)malloc::allocate(__bytes_to_get);
        // This should either throw an
        // exception or remedy the situation. Thus we assume it
        // succeeded.
    }
    _S_heap_size += __bytes_to_get;
    _S_end_free = _S_start_free + __bytes_to_get;
    return (_S_chunk_alloc(__size, __nobjs));
}
}

```

可以看到chunk_alloc 函数的输入有两个：

1. int&__nobjs，待分配的元素个数；

2. size_t __size，每个元素的大小。

chunk_alloc 函数的程序流程图（为了方便观察，画的并不是标准的程序流程图）如下：

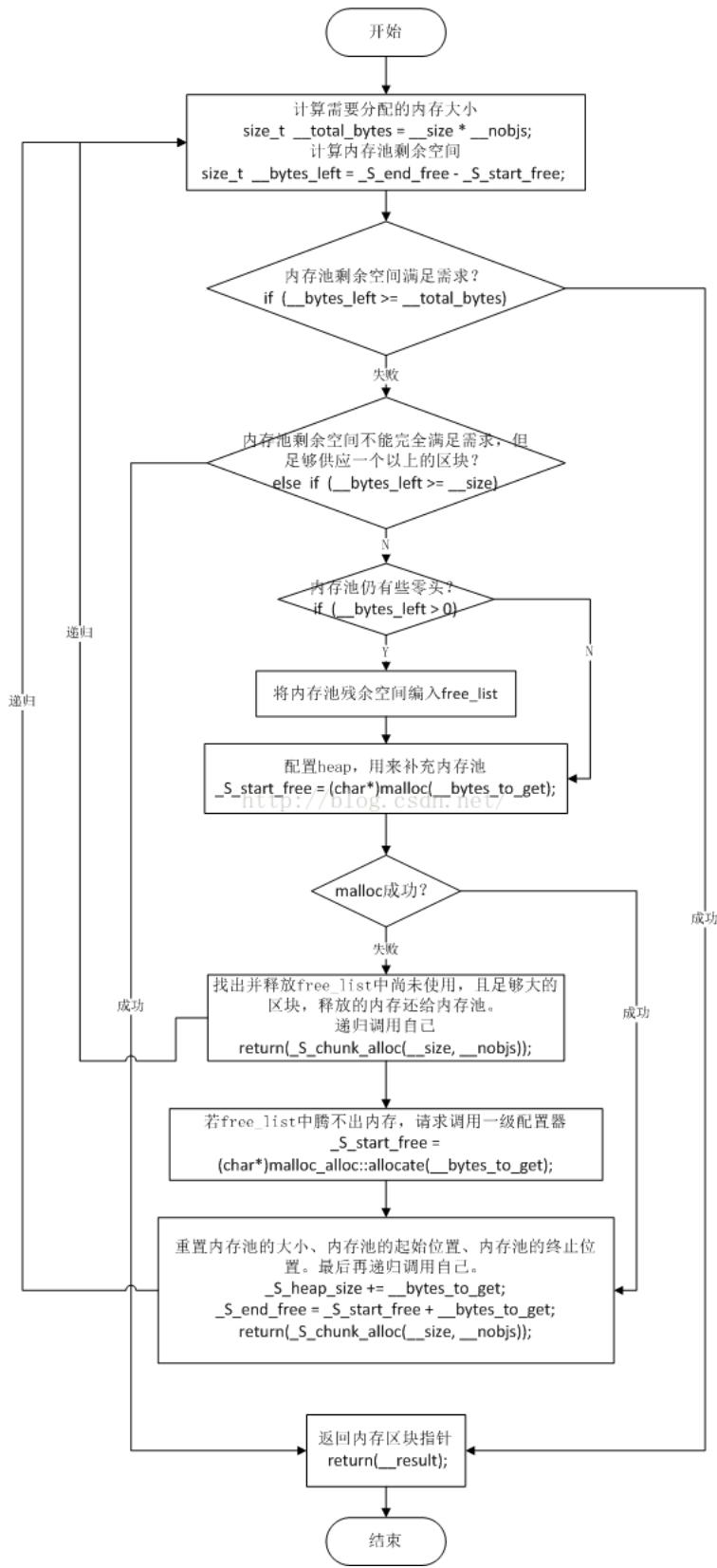


图 1.7: chunkAlloc 流程

deallocate() 若回收长度大于 128bytes 则直接调用一级配置器，也就是::operator delete。否则就在 free-lists 上寻找对应大小的区块，调整该 free-list，回收区块。

```

static void
deallocate( void* __p, size_t __n)

```

```
{  
    if (_n > (size_t) _MAX_BYTES)  
        __mem_interface::deallocate(__p, __n);  
    else  
    {  
        _Obj* volatile* __my_free_list  
            = _S_free_list + _S_freelist_index(__n);  
        _Obj* __q = (_Obj*)__p;  
  
        __q -> _M_free_list_link = *__my_free_list;  
        *__my_free_list = __q;  
    }  
}
```

1.6 内存泄漏

1.6.1 如何对付内存泄漏？

当你的代码中到处充满了 new 操作、delete 操作和指针运算的话，你将会在某个地方搞晕了头，导致内存泄漏，指针引用错误，以及诸如此类的问题。这和你如何小心地对待内存分配工作其实完全没有关系：代码的复杂性最终总是会超过你能够付出的时间和努力。于是随后产生了一些成功的技巧，它们依赖于将内存分配（allocations）与重新分配（deallocation）工作隐藏在易于管理的类型之后。

标准容器（standard containers）是一个优秀的例子。它们不是通过你而是自己为元素管理内存，从而避免了产生糟糕的结果。

模板和标准库实现了容器、资源句柄以及诸如此类的东西，更早的使用甚至在多年以前。异常的使用使之更加完善。

如果你实在不能将内存分配/重新分配的操作隐藏到你需要的对象中时，你可以使用资源句柄（resource handle），以将内存泄漏的可能性降至最低。这里有个例子：我需要通过一个函数，在空闲内存中建立一个对象并返回它。这时候可能忘记释放这个对象。毕竟，我们不能说，仅仅关注当这个指针要被释放的时候，谁将负责去做。使用资源句柄，这里用了标准库中的auto_ptr，使需要为之负责的地方变得明确了。

1.6.2 内存泄漏的发生方式

- 常发性内存泄漏。
- 偶发性内存泄漏。
- 一次性内存泄漏。
- 隐式内存泄漏。程序在运行过程中不停的分配内存，但是直到结束的时候才释放内存。严格说这里并没有发生内存泄漏，因为最终程序释放了所有申请的内存。但是对于一个服

务器程序，需要运行几天，几周甚至几个月，不及时释放内存也可能导致最终耗尽系统的所有内存。所以，我们称这类内存泄漏为隐式内存泄漏。举一个例子：

```
class Connection
{
public:
    Connection( SOCKET s );
    ~Connection();
    ...

private:
    SOCKET _socket;
    ...
};

class ConnectionManager
{
public:
    ConnectionManager() {}

    ~ConnectionManager() {
        list::iterator it;

        for( it = _connlist.begin(); it != _connlist.end(); ++it ) {
            delete (*it);
        }
        _connlist.clear();
    }

    void OnClientConnected( SOCKET s ) {
        Connection* p = new Connection(s);
        _connlist.push_back(p);
    }

    void OnClientDisconnected( Connection* pconn ) {
        _connlist.remove( pconn );
        delete pconn;
    }
private:
    list _connlist;
};
```

假设在 Client 从 Server 端断开后，Server 并没有呼叫 OnClientDisconnected() 函数，那么代表那次连接的 Connection 对象就不会被及时的删除（在 Server 程序退出的时候，所有 Connection 对象会在 ConnectionManager 的析构函数里被删除）。当不断的有连接建立、断开时隐式内存泄漏就发生了。

从用户使用程序的角度来看，内存泄漏本身不会产生什么危害，作为一般的用户，根本感觉不到内存泄漏的存在。真正有危害的是内存泄漏的堆积，这会最终消耗尽系统所有的内

存。从这个角度来说，一次性内存泄漏并没有什么危害，因为它不会堆积，而隐式内存泄漏危害性则非常大，因为较之于常发性和偶发性内存泄漏它更难被检测到。

1.6.3 C/C++ 内存泄漏及其检测工具

1.6.4 检测内存泄漏

1.7 内存回收

C++ 将内存划分为三个逻辑区域：堆、栈和静态存储区。既然如此，我称位于它们之中的对象分别为堆对象，栈对象以及静态对象。那么这些不同的内存对象有什么区别了？堆对象和栈对象各有什么优劣了？如何禁止创建堆对象或栈对象了？

1.7.1 内存对象大会战

几乎所有的临时对象都是栈对象。比如，下面的函数定义：

```
Type fun(Type object);
```

这个函数至少产生两个临时对象，首先，参数是按值传递的，所以会调用拷贝构造函数生成一个临时对象object_copy1，在函数内部使用的不是使用的不是 object，而是object_copy1，自然，object_copy1 是一个栈对象，它在函数返回时被释放；还有这个函数是值返回的，在函数返回时，如果我们不考虑返回值优化（NRV），那么也会产生一个临时对象object_copy2，这个临时对象会在函数返回后一段时间内被释放。

静态存储区。所有的静态对象、全局对象都于静态存储区分配。关于全局对象，是在 `main()` 函数执行前就分配好了的。其实，在 `main()` 函数中的显示代码执行之前，会调用一个由编译器生成的 `_main()` 函数，而 `_main()` 函数会进行所有全局对象的构造及初始化工作。而在 `main()` 函数结束之前，会调用由编译器生成的 `exit` 函数，来释放所有的全局对象。

那么，局部静态对象了？局部静态对象通常也是在函数中定义的，就像栈对象一样，只不过，其前面多了个 `static` 关键字。局部静态对象的生命期是从其所在函数第一次被调用，更确切地说，是当第一次执行到该静态对象的声明代码时，产生该静态局部对象，直到整个程序结束时，才销毁该对象。

`class` 的静态成员?`class` 的静态成员对象的生命期，`class` 的静态成员对象随着第一个 `class object` 的产生而产生，在整个程序结束时消亡。也就是有这样的情况存在，在程序中我们定义了一个 `class`，该类中有一个静态对象作为成员，但是在程序执行过程中，如果我们没有创建任何一个该 `class object`，那么也就不会产生该 `class` 所包含的那个静态对象。还有，如果创建了多个 `class object`，那么所有这些 `object` 都共享那个静态对象成员。

1.7.2 垃圾回收方法

许多 C 或者 C++ 程序员对垃圾回收嗤之以鼻，认为垃圾回收肯定比自己来管理动态内存要低效，而且在回收的时候一定会让程序停顿在那里，而如果自己控制内存管理的话，分配和释放

时间都是稳定的，不会导致程序停顿。最后，很多 C/C++ 程序员坚信在 C/C++ 中无法实现垃圾回收机制。这些错误的观点都是由于不了解垃圾回收的算法而臆想出来的。

其实垃圾回收机制并不慢，甚至比动态内存分配更高效。因为我们可以只分配不释放，那么分配内存的时候只需要从堆上一直获得新的内存，移动堆顶的指针就够了；而释放的过程被省略了，自然也加快了速度。现代的垃圾回收算法已经发展了很多，增量收集算法已经可以让垃圾回收过程分段进行，避免打断程序的运行了。而传统的动态内存管理的算法同样有在适当的时间收集内存碎片的工作要做，并不比垃圾回收更有优势。

而垃圾回收的算法的基础通常基于扫描并标记当前可能被使用的所有内存块，从已经被分配的所有内存中把未标记的内存回收来做的。C/C++ 中无法实现垃圾回收的观点通常基于无法正确扫描出所有可能还会被使用的内存块，但是，看似不可能的事情实际上实现起来却并不复杂。

首先，通过扫描内存的数据，指向堆上动态分配出来内存的指针是很容易被识别出来的，如果有识别错误，也只能是把一些不是指针的数据当成指针，而不会把指针当成非指针数据。这样，回收垃圾的过程只会漏回收掉而不会错误的把不应该回收的内存清理。

其次，如果回溯所有内存块被引用的根，只可能存在于全局变量和当前的栈内，而全局变量（包括函数内的静态变量）都是集中存在于 bss 段或 data 段中。

垃圾回收的时候，只需要扫描 bss 段、data 段以及当前被使用着的栈空间，找到可能是动态内存指针的量，把引用到的内存递归扫描就可以得到目前正在使用的所有动态内存了。

如果肯为你的工程实现一个不错的垃圾回收器，提高内存管理的速度，甚至减少总的内存消耗都是可能的。如果有兴趣的话，可以搜索一下网上已有的关于垃圾回收的论文和实现了的库，开拓视野对一个程序员尤为重要。

1.8 内存屏障

<https://blog.csdn.net/liumf2005/article/details/8489265>
<https://blog.csdn.net/u012233832/article/details/79619648>
https://blog.csdn.net/world_hello_100/article/details/50131497
<https://blog.csdn.net/u010559006/article/details/82595922>
<https://my.oschina.net/u/269082/blog/873612>

内存屏障其实就是因为编译器优化和 CPU 对寄存器和 cache 的使用，导致对内存的操作不能够及时的反映出来，比如 cpu 写入后，读出来的值可能是旧的内容。

举个例子，对一个变量赋值然后读出它的值这一看似“原子”的操作，因为内存是没有 ALU 计算单元的，所以内存没有计算的能力。而 CPU 一般情况下是不直接读写内存的（emmintrin.h 应用例外），所以这一个过程可以看作（编译器优化后）：

读取内存数据到 cache → CPU 读取 cache/寄存器 → CPU 的计算 → 将结果写入 cache/寄存器 → 写回数据到内存

有人可能会问，为什么要这么麻烦，因为是编译器优化和 CPU 优化的结果，内存的时延比 CPU 高的多，是 10ns 级别，所以会通过读写寄存器或拆 cache 优化。这有可能导致一个问题，cache 中的数据和内存实际的数据不一致，当多线程情况下，有可能会读“脏数据”，或者带来线程执行结果不一致。这就是所谓的“内存屏障”（但不仅限于此 case）。

大部分内存屏障导致的问题都出现在内核态，用户态需要注意的方面不多。而且用户也有相应的解决方案，最简单的就是锁机制，还有 **volatile** 关键字，可以把可能出现的 cache 读脏的数据 `volatile int tmp = 0;` 这样每次操作都会从内存获取。

其核心旨意其实就是防止编译器对其进行优化（可以预防上文中提到的编译器导致的内存屏障），也就是每次 **cpu** 要获取一个变量，都要去内存中重新读取，而不是从寄存器 **Cache** 中。

第二章 装载链接原理

参考文献: <https://tech.meituan.com/linker.html>

为了让读者可以对源代码如何编译到二进制可执行程序有一个整体的了解, 将会从以下几个方面介绍一下程序编译, 链接和装载的基本原理。

2.1 CPU 体系

我们现在大部分同学接触到的 PC 机或者服务器使用的 CPU 都是 X86_64 指令集体系结构, 这是一种基于 CISC (复杂指令集体系结构)。我们在计算机组成原理的课程里面都学到, 其实 CPU 指令集类型中除了 CISC, 还有另外一种 RISC 类型的 CPU 体系结构, 也就是简单指令集体系结构, 比如 SUN 的 SPARC 指令集, IBM 的 PowerPC 指令集都是基于 RISC 指令集的 CPU 体系结构。我们这里不去深究各种体系结构的细节, 我们关心的是在其中一种 **CPU 体系结构** 中编译的代码能够在另一种体系结构下面运行么?

答案是否定的, 因为所谓的二进制程序, 其实都是有一条一条的 CPU 指令组成, 二进制程序执行的过程中, 也是由 CPU 把这些指令 load 到指令流中一条一条执行。不同的 CPU 体系结构的指令集是不一样的, 指令的长度和组成都有区别。所以让 SPARC 的 CPU 去执行一个编译成 X86 的 CPU 指令集的二进制程序是不可行的。

2.2 跨平台原理

为什么经过 gcc/g++ 编译过的二进制程序不能跨平台执行呢?

java 程序能够跨平台执行是因为不同系统平台上面安装的 java 虚拟机能够识别同一种 java 字节码。那么我们是不是可以推断, 不同的操作系统二进制程序不能跨平台执行, 是因为不同操作系统下面二进制文件的格式不同呢?

事实确实是这样的。我们都知道的是一个程序编译成二进制之后, 运行的时候是从 main 函数开始执行的。但是这个程序是怎么样 load 到内存中的, 执行流又是如何准确的定位到 main 函数的地址的。其实这些工作都是操作系统替我们做的。

首先, 操作系统肯定要分配一块虚拟地址空间;

然后, 系统需要把二进制程序中的代码和数据 load 到这个地址空间中,

随后, 系统会根据某种特定的文件格式, 找到其中某一个特定的位置 (初始化段), 做一些程序运行前的初始化工作, 比如环境变量初始化和全局变量的处理, 然后开始执行我们的 main 函数。这里的“某种特定的文件格式”就是为什么二进制程序不能跨平台运行的原因。

这里我真正想说的是，每一种操作系统有自己的二进制文件格式，操作系统把二进制可执行程序 load 到内存中之后，会根据默认的这种格式寻找各种数据，比如代码段，数据段和初始化段。所以说 Windows 下面的 exe 可执行文件，lib 静态库，dll 动态库是不可以直接运行在 Linux 系统下面的；MacOS 下面的 Mach-O 可执行文件，静态链接库（a 库），动态链接库（so 库）也是不能够直接放在 Linux 系统下面运行的。反之亦然，Linux 下面的 ELF 可执行文件，静态链接库（a 库），动态链接库（so 库）同样不能够在 Window 系统下面运行。

2.3 C 例子-> 编译器与链接器

说完了 CPU 体系结构和操作系统对二进制文件格式的影响，下面我们从几个例子看一下从源代码文件如何经过处理最终变成一个可执行文件。不同的系统下有不同的编译器，比如 Windows 下有 vs 自带的 C++ 编译器，Linux 和 Unix 下面有 gcc/g++ 编译器。也有很多不同的编程语言，各自有自己的编译器把相应的源代码编译成二进制可执行程序。尽管有些实现细节不同，这些编译器的工作原理和过程是一致的。由于 Linux 和 c 语言使用相对广泛，同时笔者对 Linux 和 C/C++ 相对熟悉，本文剩下的部分都是基于在 Linux 平台下使用 gcc/g++ 编译器编译 c/c++ 源代码进行说明和解释。

本文的初衷是让工程师对程序源代码如何通过编译器，链接器和装载器最终成为一个进程运行在系统中的整个过程有一个基本的理解，所以并不会涉及到编译器如何通过进行词法分析，语法分析和语义分析最终得到目标二进制文件。因此本文剩下的部分主要集中在 gcc/g++ 如何形成一个 Linux 认识的 elf 可执行文件的。

2.3.1 C 源文件

```
int g_a = 1;           // 定义有初始值全局变量
int g_b;              // 定义无初始值全局变量
static int g_c;        // 定义全局 static 变量
extern int g_x;        // 声明全局变量
extern int sub();       // 函数声明

int sum(int m, int n) {           // 函数定义
    return m+n;
}
int main(int argc, char* argv[]) {
    static int s_a = 0;           // 局部 static 变量
    int l_a = 0;                 // 局部非 static 变量
    sum(g_a,g_b);
    return 0;
}
```

2.3.2 目标文件

在我们用 gcc 编译这个程序来查看编译器做了哪些事情之前，其实我们可以简单梳理一下，为了让这个程序能够运行起来，编译器至少都需要做哪些事情。

首先，我们都会默认 CPU 会根据程序的代码一条一条执行；其实，当遇到了条件判断或者函数调用，程序就会发生指令流的跳转；还有，程序代码执行的过程中需要操作各种变量指向的数据。从这三个“理所当然”的行为里面，我们可以推断出编译器至少需要做哪些事情。

第一，CPU 肯定不能理解这些高级语言代码，编译器需要把代码编译成二进制指令。

第二，指令流跳转的时候，CPU 怎么能找到要跳转的位置，编译器需要为每个定义的函数所在的位置定义一个标签，每个标签有一个地址，调用每个函数的时候就相当于跳转到那个标签指向的地址。

第三，CPU 如何能找到那些变量指向的数据，编译器需要为每一个变量定义一个标签，每个标签同样有一个地址，这个地址指向内存中的数据空间。

我们来实际看一下编译器的行为，我们先把这个这个编译成目标文件看一下：

```
gcc -c test.c -o test.o && nm test.o
```

```
0000000000000000 D g_a
0000000000000004 C g_b
0000000000000000 b g_c
          U g_x
0000000000000014 T main
0000000000000004 b s_a.1597
0000000000000000 T sum
```

首先我们用 gcc -c 命令把 test.c 源码文件编译成 test.o 目标文件，需要注意的是虽然目标文件也是二进制文件，但是和可执行文件是有区别的，目标文件仅仅把当前的源码文件编译成二进制文件，并没有经过链接过程，是不能够执行的。然后我们用 nm 命令可以查看一下目标文件的 **symbol** 信息。这里我们看到了 nm 命令的输出默认有三列，其中最左边的一列是变量的相对地址，中间的一列表示变量所在的段的类型，右面表示变量的名字。

- 首先最左边这一列是变量在所在段的相对地址，我们看到 g_a 和 g_c 的相对地址是相同的，这并不冲突，因为他们处于不同的段中（D 和 b 表示它们在目标文件中处于不同的段中）。
- 第二列表示变量所处的段的类型，比如我们这里看到了有 D,C,b,T 这些类型的段，实际上编译器支持的段类型比这个还多。我们同样不去深究各个段类型的意思，只要明白不同的段存放的是不同的数据即可，比如 **D 段就是数据段**，专门存放有初始值的全局变量，**T 段表示代码段**，所有的代码编译后的指令都放到这个段中。在这里我们可以注意到同一个段中的变量相对地址是不能重复的。
- 第三列表示变量的名字，这里我们看到局部的静态变量名字被编译器修改为 s_a.1597，我们应该能猜得到编译器这么做的原因。s_a 是一个局部静态变量，作用域限制在定义它的代码块中，所以我们在不同的作用域中声明相同名字的局部静态变量，比如我们可以在 sum 函数中声明另外一个 s_a。但是我们上面提过，局部静态变量属于全局变量的范畴，它是存在于程序运行的整个生命周期的，所以为了支持这个功能，编译器对这种局部的静

态变量名字加了一个后缀以便标识不同的局部静态变量。

为什么这里的变量声明 `g_x` 没有地址呢？我们在 C 源码文件部分曾经提到过，**变量和函数的声明本质上是给编译器一个承诺**，告诉编译器虽然在本文件中没有这个变量或者函数定义，但是在其他文件中一定有，所以当编译器发现程序需要读取这个变量对应的数据，但是在源文件中找不到的时候，就会把这个变量放在一个特殊的段（段类型为 U）里面，表示后续链接的时候需要在后面的目标文件或者链接库中找到这个变量，然后链接成为可执行二进制文件。

从上面这些信息的解释其实我们可以看出来，编译器没有那么复杂，它做的任何事情都是为了支持语言级别的功能。

2.3.3 目标文件的链接

上节我们留下了一个需要在其他目标文件中寻找的变量名。这一小节，我们讨论一下，在编译器把各个 C 源代码文件编译成目标文件之后，链接器需要对这些目标文件做什么样的处理。

首先我们尝试一下对上一小节得到的目标文件链接一下看看有什么结果：`gcc test.o -o test`

```
test.o: In function ‘main’:  
  undefined reference to ‘g_x’  
collect2: ld returned 1 exit status
```

当我们尝试把这个目标文件进行链接成为可执行文件时，链接器报错了。因为我们之前通过变量声明承诺过的变量并没有在其他的目标文件或者库文件中找到，所以链接器无法得到一个完整可执行程序。我们尝试用另外一个 C 程序修复这个问题：

```
int g_x = 100;  
int sub() {}
```

把这个文件编译成目标文件`gcc -c test2.c -o test2.o; nm test2.o`

```
0000000000000000 D g_x  
0000000000000000 T sub
```

现在我们尝试把这两个目标文件链接成为可执行文件：`gcc test.o test2.o -o test; nm test`，这时我们发现输出了比目标文件多很多的信息，其中定义了很多为了实现不同语言级别的功能而需要的段，在这里我们关心的是源文件中定义的那些变量对应的 symbol 及其地址，如下图所示：

```
00000000004005e8 T __fini  
0000000000400390 T __init  
00000000004003d0 T __start  
...  
0000000000601018 D g_a  
0000000000601038 B g_b  
0000000000601030 B g_c  
000000000060101c D g_x  
00000000004004c8 T main  
0000000000601034 B s_a.1597  
0000000000400504 T sub  
00000000004004b4 T sum
```

在最终的可以执行文件里面,我们可以看到,首先,之前在第一个源文件中声明的变量g_x 和声明的函数最终在第二个目标文件中找到了定义;其次,在不同目标文件中定义的变量,比如 g_a、g_x 都会放在了数据段中(段类型为D);还有,之前在目标文件中变量的相对地址全部变成了绝对地址。

所以我们再一次进行总结一下链接器需要对源代码进行的处理:

- 对各个目标文件中没有定义的变量,在其他目标文件中寻找到相关的定义。
- 把不同目标文件中生成的同类型的段进行合并。
- 对不同目标文件中的变量进行地址重定位。

这也是链接器所需要实现的最基本的功能。

2.4 执行期间-> 装载器程序

上面的几个小节中我们讨论了编译器把一个 C 源码文件编译成一个目标文件需要做的最基本的处理,也讨论了链接器把多个目标文件链接成可执行文件时需要具备的最基本的功能。在这一个小节我们来讨论一下可执行文件如何被系统装载运行的。

2.4.1 动态链接库

我们都知道,在我们写程序的过程中,不会自己实现所有的功能,一般情况下会调用我们需要的系统库和第三方库来实现我们的功能。在上面两个小节的示例代码中,为了说明问题的简单起见,我们仅仅声明,定义了几个变量和函数,并没有使用任何的库函数。那么现在假设我们需要调用一个库函数提供的功能,这个时候可执行文件又是什么样的呢,我们再看一个小例子:

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

int main(int argc, char* argv[]) {
    char buf[32];
    strncpy(buf, "Hello ,World\n", 32);
    printf("%s", buf);
}
```

我们把这个文件编译成可执行文件并且查看一下它的 symbols:

```
gcc test3.c -o test3; nm test3
```

```
00000000004005b4 T main
U printf@@GLIBC_2.2.5
U strcpy@@GLIBC_2.2.5
```

现在我们检查的是可执行文件,为什么可执行文件里面仍然有这种没有地址的 symbols 呢?这里可执行文件中的“未定义”的 symbols 其实是为了支持动态链接库的功能。

我们先来回顾一下动态链接库应该有一个什么样的功能。所谓动态链接库是指,程序在运行的时候才去定位这个库,并且把这个库链接到进程的虚拟地址空间。对于某一个动态链接库来

说，所有使用这个库的可执行文件都共享同一块物理地址空间，这个物理地址空间在当前动态链接库第一次被链接时 load 到内存中。

现在我们看一下二进制文件中对动态链接库中的函数怎么处理的，objdump -D test3 | less，搜索printf我们应该能看到以下内容：

```
0000000000400490 <strncpy@plt>:  
400490: ff 25 6a 0b 20 00 jmpq *0x200b6a(%rip) #  
601000 <_GLOBAL_OFFSET_TABLE_+0x18>  
400496: 68 00 00 00 00 pushq $0x0  
40049b: e9 e0 ff ff ff jmpq 400480 <_init+0x20>  
...  
00000000004004b0 <printf@plt>:  
4004b0: ff 25 5a 0b 20 00 jmpq *0x200b5a(%rip) #  
601010 <_GLOBAL_OFFSET_TABLE_+0x28>  
4004b6: 68 02 00 00 00 pushq $0x2  
4004bb: e9 c0 ff ff ff jmpq 400480 <_init+0x20>
```

我们看到可执行文件中为 strncpy 和 printf 分别生成了三个代理 symbol，然后代理 symbol 指向的第一条指令就是跳转到 _GLOBAL_OFFSET_TABLE_ 这个 symbol 对应的代码段中的一个偏移位置，而在 linux 中，这个 _GLOBAL_OFFSET_TABLE_ 对应的代码段是为了给“地址无关代码”做动态地址重定位用的。我们提过，动态链接库可以映射到不同进程的不同的虚拟地址空间，所以属于“地址无关代码”，链接器把对这个函数的调用代码跳转到程序运行时动态装载地址。

Linux 提供了一个很方便的命令查看一个可执行文件依赖的动态链接库，我们查看一下当前可执行文件的动态库依赖情况：ldd test3：

```
linux-vdso.so.1 => (0x00007fff413ff000)  
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe202ae7000)  
/lib64/ld-linux-x86-64.so.2 (0x00007fe202eb2000)
```

ldd 命令模拟加载可执行程序需要的动态链接库，但并不执行程序，后面的地址部分表示模拟装载过程中动态链接库的地址。如果尝试多次运行 ldd 命令，我们会发现每次动态链接库的地址都是不一样的，因为这个地址是动态定位的。

我们平常工作中，如果某一个二进制可执行文件报错找不到某个函数定义，可以用这个命令检查是否系统丢失或者没有安装某一个动态链接库。

我们在上面的小程序最后加一个 sleep(1000);，然后查看一下运行时的内存映射分配，cd /proc/21509 && cat maps，应该可以看到下面这一段：

```
7feeff61f000-7feeff7d4000 r-xp 00000000 fd:01 135891 /  
lib/x86_64-linux-gnu/libc-2.15.so  
7feeff7d4000-7feeff9d3000 ---p 001b5000 fd:01 135891 /  
lib/x86_64-linux-gnu/libc-2.15.so  
7feeff9d3000-7feeff9d7000 r--p 001b4000 fd:01 135891 /  
lib/x86_64-linux-gnu/libc-2.15.so  
7feeff9d7000-7feeff9d9000 rw-p 001b8000 fd:01 135891 /  
lib/x86_64-linux-gnu/libc-2.15.so
```

我们可以看到进程运行时，系统为 libc 库在进程地址空间中映射了四个段，因为每个段权限不同，所以不能合并为一个段。对这些动态链接库的调用最终会跳转到这里显示的地址中。

根据以上这些信息，我们在这里继续总结一下链接器需要对动态链接库需要做的最基本的事情：

- 链接库在将目标文件链接成可执行文件的时候如果发现某一个变量或者函数在目标文件中找不到，会按照 gcc 预定义的动态库寻找路径寻找动态库中定义的变量或者函数。
- 如果链接库在某一个动态链接库中找到了该变量或者函数定义，链接库首先会把这个动态链接库写到可执行文件的依赖库中，然后生成这个当前变量或者函数的代理 symbol。
- 在 `_GLOBAL_OFFSET_TABLE_` 代码中生成真正的动态跳转指令，并且在库函数（比如 `strncpy`, `printf`）代理 symbol 中跳转到 `_GLOBAL_OFFSET_TABLE_` 中相应的偏移位置。

前面我们一直在讨论动态链接库（so 库），其实在各个平台下面都有静态链接库，静态链接库的链接行为跟目标文件非常类似，但是由于静态库有一些问题，比如因为每个可执行文件都有静态库的一个版本，这导致库升级的时候很麻烦等问题，现在静态库用的非常少，所以这里我们不去深究。

2.4.2 Main 函数之前

我们总结一下当我们通过 bash 运行一个程序的时候，Linux 做了哪些事情：

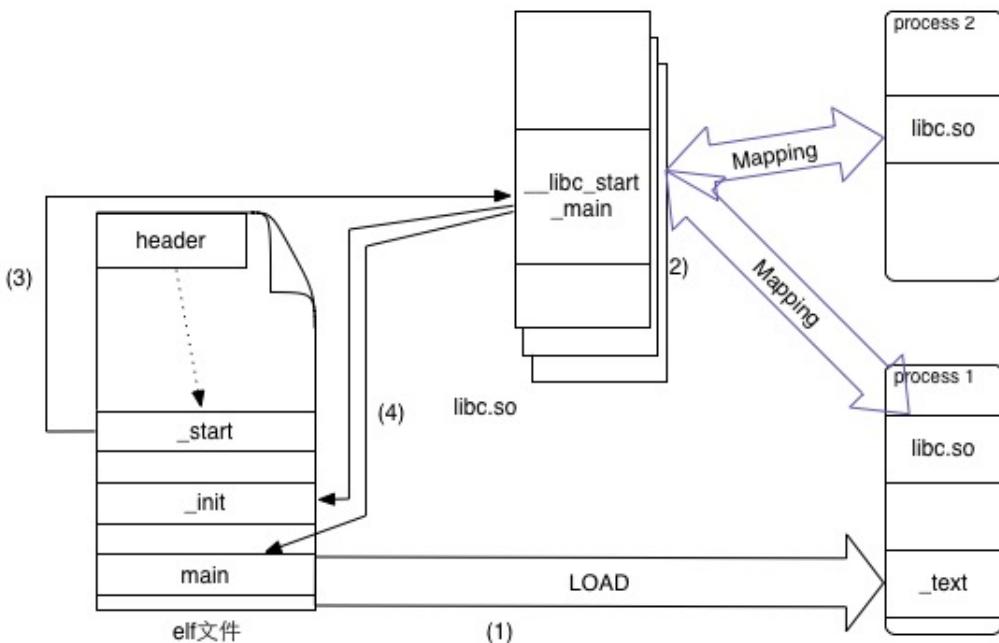


图 2.1: 进入 Main 之前操作系统做的事情

- 首先 bash 进行 fork 系统调用，生成一个子进程，接着在子进程中运行 execve 函数指定的 elf 二进制程序（Linux 中执行二进制程序最终都是通过 execve 这个库函数进行的），execve 会调用系统调用把 elf 文件 load 到内存中的代码段 (`_text`) 中。

- 如果有依赖的动态链接库，会调用动态链接器进行库文件的地址映射，动态链接库的内存空间是被多个进程共享的。
- 内核从 elf 文件头得到_start 的地址，调度执行流从_start 指向的地址开始执行，执行流在_start 执行的代码段中跳转到 libc 中的公共初始化代码段 __libc_start_main，进行程序运行前的初始化工作。
 - __libc_start_main 的执行过程中，会跳转到_init 中全局变量的初始化工作，随后 调用我们的main 函数，进入到主函数的指令流程。

第三章 正则表达式

3.1 基础知识

头文件 #include <regex>

3.1.1 整个字符串是否匹配

regex_match

```
regex reg1("\\w+day");
string s1 = "saturday";
string s2 = "saturday_and_sunday";
smatch r1;
smatch r2;
cout << boolalpha << regex_match(s1, r1, reg1) << endl;      //true
cout << boolalpha << regex_match(s2, r2, reg1) << endl;      //false
cout << "s1 匹配结果: " << r1.str() << endl;                  //saturday
cout << "s2 匹配结果: " << r2.str() << endl;                  //空
cout << endl;
```

3.1.2 只返回一个匹配结果

regex_match

```
smatch rr1;
smatch rr2;
cout << boolalpha << regex_search(s1, rr1, reg1) << endl;    //true
cout << "s1 匹配结果: " << rr1.str() << endl;                //
    saturday
cout << boolalpha << regex_search(s2, rr2, reg1) << endl;    //true
cout << "s1 匹配结果: " << rr2.str() << endl;                //
    saturday
cout << endl;
```

3.1.3 返回多个匹配结果

iterator

```

cout << "iterator 结果: " << endl;
sregex_iterator it(s2.begin(), s2.end(), reg1);
sregex_iterator end;
for(; it != end; ++it)
{
    cout << it->str() << endl;
    //cout << *it << endl; 错误
}

cout << "token_iterator 结果: " << endl;
sregex_token_iterator tit(s2.begin(), s2.end(), reg1);
sregex_token_iterator tend;
for(; tit != tend; ++tit)
{
    cout << tit->str() << endl;
    cout << *tit << endl;
}

```

3.2 子表达式匹配

```

regex reg2("(\\d{1,3})(\\d{1,3})(\\d{1,3})(\\d{1,3})");
string ip = "0:11:222:333";
smatch m;
regex_match(ip, m, reg2);
cout << "输出: str()" << endl;
cout << m.str() << endl;      //0:11:222:333
cout << m.str(1) << endl;      //0
cout << m.str(2) << endl;      //11
cout << m.str(3) << endl;      //222
cout << m.str(4) << endl;      //333

cout << "输出: [i]" << endl; //结果同上
cout << m[0] << endl;
cout << m[1] << endl;
cout << m[2] << endl;
cout << m[3] << endl;
cout << m[4] << endl;

string ip2 = "0:11:222:333\4:55:66:7";
sregex_iterator ip_it(ip2.begin(), ip2.end(), reg2);
sregex_iterator ip_end;
for(; ip_it != ip_end; ++ip_it)
{
    cout << ip_it->str() << endl;
    cout << ip_it->str(1) << endl;
}

```

```
    cout << ip_it->str(2) << endl;
    cout << ip_it->str(3) << endl;
    cout << ip_it->str(4) << endl;
}
```


第四章 异常处理

4.1 简介

用于处理软件程序运行时的错误，并用于处理软件系统中可预知或不可预知的问题。这样就可以保证软件系统运行的稳定性与健壮性

C++ 应用程序中在考虑异常设计时，并不是所有的程序模块处理都需要附加上异常情况的处理。相对来说，异常处理没有普通方法函数调用速度快。过度的错误处理会影响应用程序运行的效率。通常在 C++ 开发的软件系统中，应用程序都由对应的库、组件以及运行的具体不同模块组成。在设计时，异常的处理应充分考虑到独立程序库以及组件之间的情况。便于使用者在程序出现异常情况下，使用库或者组件的开发者能够快速定位出库、组件还是应用程序的错误。

编写流程

1. 使用 Try 块
2. 引发异常
3. 使用 catch 捕获特定类型

try 体中可以直接抛出异常，或者在 **try** 体中调用的函数体中间接的抛出

```
try
{
    ...
    // 可能出错产生异常的代码

    throwtypen(); //throw new Type(); 一个对象或值
} catch (type1)
{
    ...
    // 对应类型的异常处理代码
} catch (type2)
{
    ...
    // 对应类型的异常处理代码
} catch (...)
{
}
```

或者

```

try
{
    function(); // 调用可能会抛出异常的函数方法
} catch (type1)
{
    ...
    // 对应类型的异常处理代码
} catch (type2)
{
    ...
    // 对应类型的异常处理代码
}

```

4.2 异常处理机制

4.2.1 异常再引发

- 可以在基本任务完成后重新引发所处理的异常，最后再throw
- 主要用于在程序终止前写入日志和实施特殊的清除任务

```

try
{
    throw AnException();
}
catch (...)
{
    ...
    throw;
}

```

4.2.2 栈展开

- 异常引发代码和异常处理程序可能属于不同函数
- 当异常发生时，沿着异常处理块的嵌套顺序逆向回溯查找能够处理该异常的 catch 字句，即如果当前函数没有处理代码，则返回上一层调用函数查找 catch 处理代码，以此类推
- 如找到对应的 catch 字句，处理该异常
- 异常处理完后，程序保持 catch 字句所在的函数栈框架，不会返回引发异常的函数栈框架，即程序停留在 catch 处理程序处。
- 函数栈框架消失时，局部对象被自动析构，但是如果未执行 delete 操作，动态分配的目标对象就未被析构

流程如图4.1所示

```

#include <iostream>

using namespace std;

```

```

int main()
{
    try {
        cout << "Before_Throw" << endl;
        throw 1; // 抛出对象 new Obj
        cout << "After_Throw_In_Try" << endl;
    }
    catch (int& i) // 捕获对象 Obj&
    {
        cout << "After_Throw_In_Catch" << endl;
    }

    //如果没有匹配的catch块，就会调用terminate():如果没设置terminate就终止程序..就不会执行下面的语句了

    cout << "Main!" << endl;
    cin.get();
}

```

```

D:\Code_Work\C++\PracticingC++\Test_Excepti
Before Throw
After Throw In Catch
Main!

```

图 4.1: 流程演示

4.2.3 未处理异常

- 所有未处理异常由预定义函数std::terminate() 函数处理
- 可以使用std::set_terminate() 函数设置std::terminate() 函数处理例程。

```

void term_func() { exit( -1 ); }

int main()
{
    try
    {

```

```

        set_terminate( term_func );
        throw "Out of memory!";
    }
    catch( int ){ /* ... */ }
    return 0;
}

```

4.2.4 描述函数可否引发异常

- 声明时，函数后加 throw() 表示不能引发异常等价于 c11noexcept 或 noexcept(true)
- 声明时，函数后加 throw(T) 表示引发某类异常
- 声明时，函数后加 throw(...) 表示可以引发任意形式异常等价于 c11noexcept(false)
- 可能引发异常时，c11 使用 noexcept(noexcept(expr)),expr 为可转换为 true 或 false 常数表达式

c11 建议使用 noexcept 来表示该函数是否可以抛出异常

```

class JuStack
{
public:
    JuStack( int cap ) : _stk( new int [cap+1]), _cap( cap ), _cnt( 0 ),
                        _top( 0 ) { }
    virtual ~JuStack() { if( _stk ) delete _stk, _stk = NULL; }
public:
    int Pop() throw( EStackEmpty ); // int pop() noexcept( false );
    void Push( int value ) throw( EStackFull );
    bool IsFull() const { return _cap == _cnt; }
    bool IsEmpty() const { return _cnt == 0; }
    int GetCapacity() const { return _cap; }
    int GetCount() const { return _cnt; }
private:
    int * _stk;
    int _cap, _cnt, _top;
};

```

4.2.5 显示异常名字

C++ 程序在编译是会将函数名称进行变化，导致栈追踪时不好读，此时可以使用 demangle 方法对函数名进行转换，使其回到原先声明的名字，方便阅读差错。

```

// 显示异常的具体代码
#include <exception>
#include <iostream>
#include <cxxabi.h>

struct empty { };

```

```

template <typename T, int N>
struct bar { };

int main()
{
    int      status;
    char    *realname;

    // exception classes not in <stdexcept>, thrown by the implementation
    // instead of the user
    std::bad_exception e;
    realname = abi::__cxa_demangle(e.what(), 0, 0, &status);
    std::cout << e.what() << "\t=>" << realname << "\t:" << status << '\n';
    free(realname);

    // typeid
    bar<empty,17> u;
    const std::type_info &ti = typeid(u);

    realname = abi::__cxa_demangle(ti.name(), 0, 0, &status);
    std::cout << ti.name() << "\t=>" << realname << "\t:" << status << '\n';
    free(realname);

    return 0;
}

/*
This prints

St13bad_exception      => std::bad_exception : 0
3barI5emptyLi17EE      => bar<empty, 17> : 0
*/

```

4.2.6 异常标准库类结构

所有的异常类型都继承自 `std::exception`，在头文件 `<exception>` 中定义。具体细节见下图：

```

#include <variant>
#include <string>

int main()
{

```

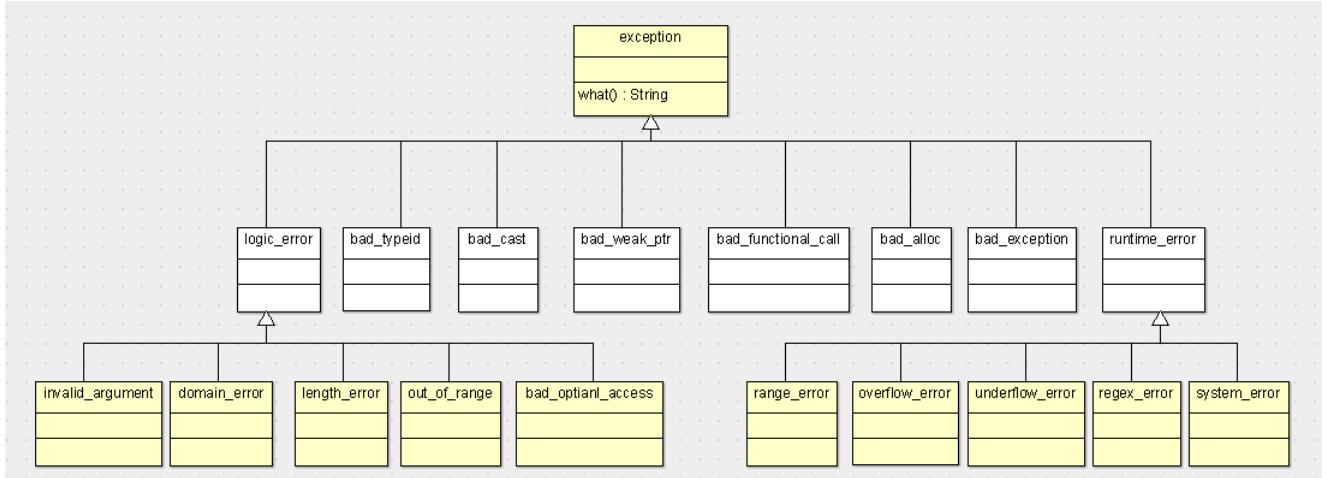


图 4.2: Exception 层次结构

```

std::variant<int, float> v, w;
v = 12; // v contains int
int i = std::get<int>(v);
w = std::get<int>(v);
w = std::get<0>(v); // same effect as the previous line
w = v; // same effect as the previous line

// std::get<double>(v); // error: no double in [int, float]
// std::get<3>(v); // error: valid index values are 0 and 1

try {
    std::get<float>(w); // w contains int, not float: will throw
}
catch (std::bad_variant_access&) {}

std::variant<std::string> v("abc"); // converting constructors work
when unambiguous
v = "def"; // converting assignment also works when unambiguous
}

```

4.3 logic_error

Defines a type of object to be thrown as exception. It reports errors that are a consequence of faulty logic within the program such as violating logical preconditions or class invariants and **may be preventable**.

4.3.1 invalid_argument

Defines a type of object to be thrown as exception. It reports errors that arise because an argument value has not been accepted.

This exception is thrown by std::bitset::bitset, and the std::stoi and std::stof families of functions.

4.3.2 domain_error

Defines a type of object to be thrown as exception. It may be used by the implementation to report domain errors, that is, situations where the inputs are outside of the domain on which an operation is defined.

The standard library components do not throw this exception (mathematical functions report domain errors as specified in math_errhandling). Third-party libraries, however, use this. For example, boost.math throws std::domain_error if boost::math::policies::throw_on_error is enabled (the default setting).

4.3.3 length_error

Defines a type of object to be thrown as exception. It reports errors that are consequence of attempt to exceed implementation defined length limits for some object.

This exception is thrown by member functions of std::basic_string and std::vector::reserve

4.3.4 out_of_range

Defines a type of object to be thrown as exception. It reports errors that are consequence of attempt to access elements out of defined range.

It may be thrown by the member functions of std::bitset and std::basic_string, by std::stoi and std::stod families of functions, and by the bounds-checked member access functions (e.g. std::vector::at and std::map::at)

4.3.5 future_error

The class std::future_error defines an exception object that is thrown on failure by the functions in the thread library that deal with asynchronous execution and shared states (std::future, std::promise, etc.). Similar to std::system_error, this exception carries an error code compatible with std::error_code.

4.3.6 bad_optional_access

Defines a type of object to be thrown by std::optional::value when accessing an optional object that does not contain a value.

4.4 runtime_error

4.4.1 range_error

Defines a type of object to be thrown as exception. It can be used to report range errors (that is, situations where a result of a computation cannot be represented by the destination type)

The only standard library components that throw this exception are std::wstring_convert::from_bytes and std::wstring_convert::to_bytes.

The mathematical functions in the standard library components do not throw this exception (mathematical functions report range errors as specified in math_errhandling).

4.4.2 overflow_error

Defines a type of object to be thrown as exception. It can be used to report arithmetic overflow errors (that is, situations where a result of a computation is too large for the destination type)

The only standard library components that throw this exception are std::bitset::to_ulong and std::bitset::to_ullong.

The mathematical functions of the standard library components do not throw this exception (mathematical functions report overflow errors as specified in math_errhandling). Third-party libraries, however, use this. For example, boost.math throws std::overflow_error if boost::math::policies::throw_on_error is enabled (the default setting).

4.4.3 underflow_error

Defines a type of object to be thrown as exception. It may be used to report arithmetic underflow errors (that is, situations where the result of a computation is a subnormal floating-point value)

The standard library components do not throw this exception (mathematical functions report underflow errors as specified in math_errhandling). Third-party libraries, however, use this. For example, boost.math throws std::underflow_error if boost::math::policies::throw_on_error is enabled (the default setting).

4.4.4 regex_error

Defines the type of exception object thrown to report errors in the regular expressions library.

```
#include <regex>
#include <iostream>

int main()
{
    try {
        std::regex re("[a-b][a]");
    }

    catch (const std::regex_error& e) {
```

```

        std::cout << "regex_error_caught:" << e.what() << '\n';
        if (e.code() == std::regex_constants::error_brack) {
            std::cout << "The code was error_brack\n";
        }
    }
}

```

4.4.5 system_error

`std::system_error` is the type of the exception thrown by various library functions (typically the functions that interface with the OS facilities, e.g. the constructor of `std::thread`) when the exception has an associated `std::error_code`, which may be reported.

ios_base::failure

The class `std::ios_base::failure` defines an exception object that is thrown on failure by the functions in the Input/Output library.

`std::ios_base::failure` may be defined either as a member class of `std::ios_base` or as a synonym (typedef) for another class with equivalent functionality.

```

#include <iostream>
#include <fstream>
int main()
{
    std::ifstream f("doesn't exist");
    try {
        f.exceptions(f.failbit);
    } catch (const std::ios_base::failure& e)
    {
        std::cout << "Caught an ios_base::failure.\n"
        << "Explanatory string: " << e.what() << '\n'
        << "Error code: " << e.code() << '\n';
    }
}

/*
Caught an ios_base::failure.
Explanatory string: ios_base::clear: unspecified iostream_category
error
Error code: iostream:1
*/

```

filesystem::filesystem_error

The class std::filesystem::filesystem_error defines an exception object that is thrown on failure by the throwing overloads of the functions in the filesystem library.

4.4.6 tx_exception

Defines an exception type that can be used to cancel and roll back an atomic transaction initiated by the keyword atomic_cancel

If T is not TriviallyCopyable, the program that specializes std::tx_exception<T> is ill-formed.

4.5 bad errors

4.5.1 bad_typeid

An exception of this type is thrown when a typeid operator is applied to a dereferenced null pointer value of a polymorphic type.

```
#include <iostream>
#include <typeinfo>

struct S { // The type has to be polymorphic
    virtual void f();
};

int main()
{
    S* p = nullptr;
    try {
        std :: cout << typeid(*p).name() << '\n';
    } catch(const std :: bad_typeid& e) {
        std :: cout << e.what() << '\n';
    }
}

//Attempted a typeid of NULL pointer!
```

4.5.2 bad_cast

bad_any_cast

Defines a type of object to be thrown by the value-returning forms of std::any_cast on failure.

4.5.3 bad_weak_ptr

std::bad_weak_ptr is the type of the object thrown as exceptions by the constructors of std::shared_ptr that take std::weak_ptr as the argument, when the std::weak_ptr refers to an already deleted object.

```
#include <memory>
#include <iostream>
int main()
{
    std::shared_ptr<int> p1(new int(42));
    std::weak_ptr<int> wp(p1);
    p1.reset();
    try {
        std::shared_ptr<int> p2(wp);
    } catch(const std::bad_weak_ptr& e) {
        std::cout << e.what() << '\n';
    }
}

//std::bad_weak_ptr
```

4.5.4 bad_function_call

std::bad_function_call is the type of the exception thrown by std::function::operator() if the function wrapper has no target.

```
#include <iostream>
#include <functional>

int main()
{
    std::function<int()> f = nullptr;
    try {
        f();
    } catch(const std::bad_function_call& e) {
        std::cout << e.what() << '\n';
    }
}

//bad function call
```

4.5.5 bad_alloc

std::bad_alloc is the type of the object thrown as exceptions by the allocation functions to report failure to allocate storage

```
#include <iostream>
```

```

#include <new>

int main()
{
    try {
        while (true) {
            new int[100000000ull];
        }
    } catch (const std::bad_alloc& e) {
        std::cout << "Allocation failed:" << e.what() << '\n';
    }
}

// Allocation failed: std::bad_alloc

```

4.5.6 bad_exception

`std::bad_exception` is the type of the exception thrown by the C++ runtime in the following situations:

1. If a dynamic exception specification is violated and `std::unexpected` throws or rethrows an exception that still violates the exception specification, but the exception specification allows `std::bad_exception`, `std::bad_exception` is thrown.
2. If `std::exception_ptr` stores a copy of the caught exception and if the copy constructor of the exception object caught by `current_exception` throws an exception, the captured exception is an instance of `std::bad_exception`.

```

#include <iostream>
#include <exception>
#include <stdexcept>

void my_unexp() { throw; }

void test() throw(std::bad_exception)
{
    throw std::runtime_error("test");
}

int main()
{
    std::set_unexpected(my_unexp);
    try {
        test();
    } catch(const std::bad_exception& e)
    {
        std::cerr << "Caught:" << e.what() << '\n';
    }
}

```

```
}
```

```
//Caught std::bad_variant_access
```

4.5.7 bad_variant_access

std::bad_variant_access is the type of the exception thrown in the following situations:

- std::get(std::variant) called with an index or type that does not match the currently active alternative
- std::visit called to visit a variant that is valueless_by_exception

```
#include <variant>
#include <iostream>

int main()
{
    std::variant<int, float> v;
    v = 12;
    try {
        std::get<float>(v);
    }
    catch(const std::bad_variant_access& e) {
        std::cout << e.what() << '\n';
    }
}

//bad_variant_access
```


第五章 多线程

5.1 参考

多线程使用: <http://www.tuicool.com/articles/VFjyYnq>

<http://www.2cto.com/kf/201507/419768.html>

<http://www.2cto.com/kf/201504/390787.html>

<http://blog.jobbole.com/81265/>

内部实现源码: <http://blog.jobbole.com/105900/>

5.2 预知

5.2.1 线程安全

线程安全就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。线程不安全就是不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据

比如一个 `ArrayList` 类，在添加一个元素的时候，它可能会有两步来完成：1. 在 `Items[Size]` 的位置存放此元素；2. 增大 `Size` 的值。

在单线程运行的情况下，如果 `Size = 0`，添加一个元素后，此元素在位置 0，而且 `Size=1`；

而如果是在多线程情况下，比如有两个线程，线程 A 先将元素 1 存放在位置 0。但是此时 CPU 调度线程 A 暂停，线程 B 得到运行的机会。线程 B 向此 `ArrayList` 添加元素 2，因为此时 `Size` 仍然等于 0（注意，我们假设的是添加一个元素是要两个步骤，而线程 A 仅仅完成了步骤 1），所以线程 B 也将元素存放在位置 0。然后线程 A 和线程 B 都继续运行，都增加 `Size` 的值，结果 `Size` 等于 2。那好，我们来看看 `ArrayList` 的情况，期望的元素应该有 2 个，而实际只有一个元素，造成丢失元素，而且 `Size` 等于 2。这就是“线程不安全”了

线程安全的条件

- 多个线程同时访问时，其表现出正确的行为。
- 无论操作系统怎么调度这些线程，无论这些线程的执行顺序如何交织
- 调用端代码 无须额外的同步或其他协调动作。

线程安全的类 可以通过同步原语(类自身拥有的 mutex(互斥器) 来保护内部状态。

对象的 race condition 这个不能由类自身拥有的 mutex(互斥器) 来保护，可以借助 boost 的 shared,weaked ptr 实现的 observer 模式解决。

5.2.2 对象的创建

对象构造时要做到线程安全，唯一要求是在构造期间不要泄露 this 指针，即：

- 不要在构造函数中注册任何回调
- 不要在构造函数中将 this 传给跨线程的对象
- 即便在最后一行也不行：派生会先执行父类的构造的，而你写的可能就是父类.. 这样对方就讲返回一个半成品..

5.3 概念

线程是比进程更小的程序执行单位

多个线程可共享全局数据，也可使用专有数据

内核线程

- 操作系统内核支持多线程调度与执行
- 内核线程使用资源较少，仅包括内核栈和上下文切换时需要的保存寄存器内容的空间

轻量级进程

- 由内核支持的独立调度单元，调度开销小于普通的进程
- 系统支持多个轻量级进程同时运行，每个都与特定的内核线程相关联

用户线程

- 建立在用户空间的多个用户级线程，映射到轻量级进程后调度执行
- 用户线程在用户空间创建、同步和销毁，开销较低
- 每个线程具有独特的 ID

进程与线程的比较

- 线程空间不独立，有问题的线程会影响其他线程；进程空间独立，有问题的进程一般不会影响其他进程
- 创建进程需要额外的性能开销
- 线程用于开发细颗粒度并行性，进程用于开发粗颗粒度并行性
- 线程容易共享数据，进程共享数据必须使用进程间通讯机制

5.4 POSIX 线程

5.4.1 线程创建

线程创建函数

- 头文件”pthread.h”
- 函数 int pthread_create(pthread_t* thread, const pthread_attr_t* attr, void* (*func)(void*), void* arg);

线程创建流程

- 定义指向pthread_t 对象的指针对象，pthread_t 对象用于存储新线程的ID
- 定义指向线程属性pthread_attr_t 对象的指针对象；线程属性对象控制线程与程序其他部分（可能是其他线程）的交互；如果传递NULL，则使用缺省属性构造新线程
- 定义指向线程函数的指针对象，使其指向固定格式的线程函数
- 实现线程函数；线程函数的参数和返回值均为哑型指针；需要传递多个参数时，打包成单个void* 型的指针对象
- 线程退出时使用返回值将数据传递给主调线程；多个结果同样可以打包传递

线程创建说明

- pthread_create() 函数在线程创建完毕后立即返回，它并不等待线程结束
- 原线程与新线程如何执行与调度有关，程序不得依赖线程先后执行的关系
- 可以使用同步机制确定线程的先后执行关系

线程退出方式

- 线程函数结束执行
- 调用pthread_exit() 函数显式结束
- 被其他线程撤销

示例代码 ->

```
#include <pthread.h>
#include <iostream>
void * PrintAs( void * unused )
{
    while( true )    std::cerr << 'a';
    return NULL;
}
void * PrintZs( void * unused )
{
    while( true )    std::cerr << 'z';
    return NULL;
}
```

```

    }
    int main()
{
    pthread_t thread_id;
    pthread_create( &thread_id, NULL, &PrintAs, NULL );
    PrintZs( NULL );
    return 0;
}

```

5.4.2 线程 ID

5.4.3 线程属性

5.4.4 线程撤销

5.4.5 线程局部存储

5.4.6 线程清除

5.5 C++11 线程

5.5.1 头文件

- <atomic>: 包含std::atomic 和std::atomic_flag 类, 以及一套 C 风格的原子类型和与 C 兼容的原子操作的函数。
- <thread>: 包含std::thread 类以及std::this_thread 命名空间。
- <mutex>: 包含与互斥量(mutex、lock_guard、unique_lock) 相关的类以及其他类型和函数。
- <condition_variable>: 包含与条件变量相关的类, 包括std::condition_variable 和 std::condition_variable_any。
- <future>: 包含两个Promise 类(std::promise 和std::package_task)和两个Future 类(std::future 和std::shared_future) 以及相关的类型和函数

5.5.2 线程类

首先其类模板模型如下图所示:

```

namespace std
{
    struct thread
    {
        // native_handle_type 是连接 thread 类和操作系统 SDK API 之间的桥梁。
        typedef implementation-dependent native_handle_type;
    };
}

```

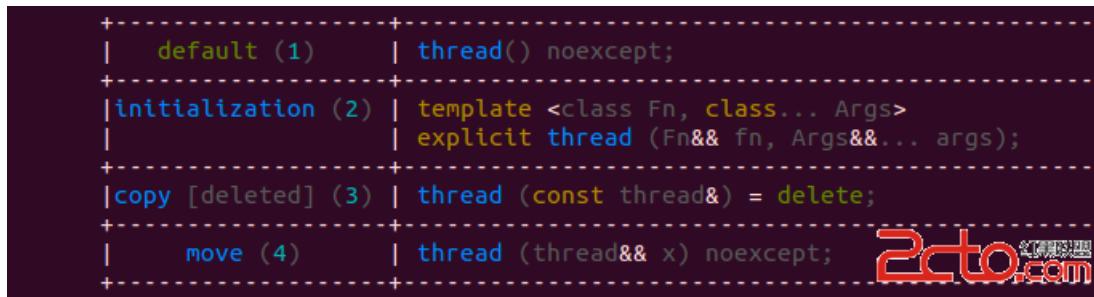


图 5.1: Thread Pattern

```

native_handle_type native_handle() ;
// 线程ID
struct id
{
    id() noexcept;
    // 可以由==, < 两个运算衍生出其它大小关系运算。
    bool operator==(thread::id x, thread::id y) noexcept;
    bool operator<(thread::id x, thread::id y) noexcept;
    template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>&out, thread::id id);
    // 哈希函数
    template <class T> struct hash;
    template <> struct hash<thread::id>;
};

id get_id() const noexcept;
// 构造与析构
thread() noexcept;
// 注意：前面为函数，后面为函数的参数表..的实参值。
template<class F, class... Args> explicit thread(F&f, Args&&... args)
);
~thread();
thread(const thread&) = delete;
thread(thread&&) noexcept;
thread& operator=( const thread&) = delete;
thread& operator=(thread&&) noexcept;
//
void swap(thread&) noexcept;
bool joinable() const noexcept; // 判断线程是否可联
void join(); // 等待线程结束
void detach(); // 分离线程
// 获取物理线程数目
static unsigned hardware_concurrency() noexcept;
}

namespace this_thread
{

```

```

        thread::id get_id();
        void yield();
        // 阻塞当前线程至指定时点
        template<class Clock, class Duration>
        void sleep_until(const chrono::time_point<Clock, Duration>&
                         abs_time);
        // 阻塞当前线程指定时长
        template<class Rep, class Period>
        void sleep_for(const chrono::duration<Rep, Period>& rel_time);
    }
}

```

初始化方式.. 创建如果指定函数，那么创建成功后就立刻执行，否则等待函数指定时执行。
join 函数用于等待结束。

线程局部存储使用thread_local 关键字

1. thread_run_func_var_args:

```

#include <iostream>
#include <thread>

using namespace std;

void myFirstThreadTask( int num)
{
    for ( int i = 0; i < 10; ++i)
    {
        cout << "myFirstThreadTask's num = " << num++ << endl;
        this_thread::sleep_for(chrono::milliseconds(10));
    }
}

void mySecondThreadTask( int &num)
{
    for ( int i = 0; i < 10; ++i)
    {
        cout << "mySecondThreadTask's num = " << num++ << endl;
        this_thread::sleep_for(chrono::milliseconds(10));
    }
}

int _tmain( int argc, _TCHAR* argv[])
{
    int n = 5;
    thread myThread0; //myThread1为一个空线程，不代表任何可执行对象
                      //myThread1与 myFirstThreadTask 函数绑定，n为其按值传递参数
}

```

```

    thread myThread1(myFirstThreadTask, n);

    //myThread2与mySecondThreadTask函数绑定, n为其引用传递参数
    thread myThread2(mySecondThreadTask, std::ref(n));
    myThread1.join();
    myThread2.join();

    n++;

    //现在myThread2不代表任何可执行对象, myThread3与
    //mySecondThreadTask函数绑定
    thread myThread3(std::move(myThread2));
    //myThread3.join();
    return 0;
}

```

2. thread_run_lambda:

```

void threadRunLambda(void)
{
    int a = 100,
        b = 200;
    thread* t = new thread(
        [] (int ia, int ib)
    {
        cout << (ia + ib) << endl;
    },
    a,
    b );
    t->join();
    delete t;
}

```

3. thread_run_member_func:

```

// Method1:
struct God
{
    void create(string& anything)
    {
        cout << "create" << anything << endl;
    }
};

void threadRunMemberFunction(void)
{
    God god;
    string s = "some";
}

```

```

        thread* t = new thread( &God::create , god , s );// 虽然函数定义为
        引用，但是还是值传递参数，如果需要引用，需要添加std::ref(s);
        // t = new thread(&God::create,god,std::ref(s)); 这样才会真正的引
        用传递参数
        t->join();
        delete t;
    }

// Method2:
class Factor{
public:
    void operator() (int a){
        cout <<"create"<< a << endl;
    }
};

void threadRunObject(void)
{
    Factor factor;
    int a = 1;

    thread t(factor , a);
    thread t2(Factor() ,a)
    t .join();
    t2.join();
}

```

4. bind()将函数转化为return Type(*pointer)()

```

#include <iostream>
#include <thread>
class Worker {
public:
    Worker( int a = 0, int b = 0 ) : _a(a) , _b(b) { }
    void ThreadFunc() { ..... }
private: int _a, _b;
};

int main()
{
    Worker worker( 10 , 20 );
    std::thread t( std::bind( &Worker::ThreadFunc , &worker ) );
    t.join();
    return 0;
}

```

5.5.3 线程间数据交互和数据争用 (Data Racing)

同一个进程内的多个线程之间多是免不了要有数据互相来往的，队列和共享数据是实现多个线程之间的数据交互的常用方式，封装好的队列使用起来相对来说不容易出错一些，而共享数据则是最基本的也是较容易出错的，因为它会产生数据争用的情况，即有超过一个线程试图同时抢占某个资源，比如对某块内存进行读写等，如下例所示：

```
static void inc(int *p)
{
    for(int i = 0; i < COUNT; ++i)
        (*p)++;
}

void ThreadDataRacing()
{
    int a = 0;
    thread th1(inc, &a);
    thread th2(inc, &a);

    th1.join();
    th2.join();

    cout<<"a = "<<a<<endl;
}
```

这是简化了的极端情况，我们可以一眼看出来这是两个线程在同时对 `&a` 这个内存地址进行写操作，但是在实际工作中，在代码的海洋中发现它并不一定容易。从表面看，两个线程执行完之后，最后的 `a` 值应该是 `COUNT * 2`，但是实际上并非如此，因为简单如 `(*p)++` 这样的操作并不是一个原子动作，要解决这个问题，对于简单的基本类型数据如字符、整型、指针等，C++ 提供了原子模版类 `atomic`，而对于复杂的对象，则提供了最常用的锁机制，比如互斥类 `mutex`，门锁 `lock_guard`，唯一锁 `unique_lock`，条件变量 `condition_variable` 等。

解决方案：

1. **atomic**: 针对简单的基本类型数据如字符、整型、指针等

```
static void inc	atomic<int> *p )
{
    for(int i = 0; i < COUNT; i++)
    {
        (*p)++;
    }
}

void threadDataRacing(void)
{
    atomic<int> a(0);

    thread ta(inc, &a);
```

```

    thread tb( inc , &a );

    ta . join ();
    tb . join ();

    cout << "a=" << a << endl;
}

```

2. **lock_guard**:`lock_guard`是一个范围锁，本质是 RAII(Resource Acquire Is Initialization)，在构建的时候自动加锁，在析构的时候自动解锁，这保证了每一次加锁都会得到解锁。即使是调用函数发生了异常，在清理栈帧的时候也会调用它的析构函数得到解锁，从而保证每次加锁都会解锁，但是我们不能手工调用加锁方法或者解锁方法来进行更加精细的资源占用管理

```

static mutex g_mutex;
static void inc(int *p )
{
    for (int i = 0; i < COUNT; i++)
    {
        lock_guard<mutex> _(g_mutex);
        (*p)++;
    }
}

void threadLockGuard( void )
{
    int a = 0;

    thread ta( inc , &a );
    thread tb( inc , &a );

    ta . join ();
    tb . join ();

    cout << "a=" << a << endl;
}

```

3. **mutex**:如果要支持手工加锁，可以考虑使用 `unique_lock` 或者直接使用 `mutex`。`unique_lock` 也支持 RAII，它也可以一次性将多个锁加锁；如果使用 `mutex` 则直接调用 `mutex` 类的 `lock`, `unlock`, `trylock` 等方法进行更加精细的锁管理

Mutual exclusion algorithms **prevent** multiple threads from simultaneously accessing shared resources. This prevents data races and provides support for synchronization between threads.

```

static mutex g_mutex;
static void inc(int *p )
{
    thread_local int i; // TLS 变量

```

```

    for ( ; i < COUNT; i++)
    {
        g_mutex.lock();
        (*p)++;
        g_mutex.unlock();
    }
}

void threadMutex( void )
{
    int a = 0;

    thread ta( inc, &a );
    thread tb( inc, &a );

    ta.join();
    tb.join();
    cout << "a=" << a << endl;
}

```

在上例中,我们还使用了线程本地存储(**TLS**)变量,我们只需要在变量前面声明它是thread_local即可。*TLS*变量在线程栈内分配,线程栈只有在线程创建之后才生效,在线程退出的时候销毁,需要注意不同系统的线程栈的大小是不同的,如果*TLS*变量占用空间比较大,需要注意这个问题。*TLS*变量一般不能跨线程,其初始化在调用线程第一次使用这个变量时进行,默认初始化为0。

4. **conditon_variable**: 对于线程间的事件通知,提供了条件变量类**condition_variable**,可视为**pthread_cond_t**的封装,使用条件变量可以让一个线程等待其它线程的通知(*wait*,*wait_for*,*wait_until*),也可以给其它线程发送通知(*notify_one*,*notify_all*),条件变量必须和锁配合使用,在等待时因为有解锁和重新加锁,所以,在等待时必须使用可以手工解锁和加锁的锁,比如**unique_lock**,而不能使用**lock_guard**

```

#include <condition_variable>
mutex m;
condition_variable cv;
void threadCondVar( void )
{
    # define THREAD_COUNT 10
    thread** t = new thread*[THREAD_COUNT];
    int i;
    for (i = 0; i < THREAD_COUNT; i++)
    {
        t[ i ] = new thread( [] ( int index ) {
            unique_lock<mutex> lck(m);
            cv.wait_for(lck, chrono::hours(1000));
            cout << index << endl;
        }, i );
    }
}

```

```

        this_thread::sleep_for( chrono::milliseconds(50));
    }
    for( i = 0; i < THREAD_COUNT; i++)
    {
        lock_guard<mutex> _(m);
        cv.notify_one();
    }
    for( i = 0; i < THREAD_COUNT; i++)
    {
        t[ i]->join();
        delete t[ i];
    }
    delete t;
}

```

5.5.4 互斥锁

多线程编程一般避免不了同步的问题，C++11 这里也提供了非常方便的方法来进行解决。标准中提供了一下四种互斥锁，分别是：

- **Mutex**: 基本的 Mutex 类，独占的互斥量，不能递归使用，提供了核心函数 lock() 和 unlock()
- **Recursive_mutex**: 递归 Mutex 类，允许在同一个线程中对一个互斥量的多次加锁与解锁操作。一般在项目模块分工中可以使用，这样即使其他人使用了同样的锁也不会导致死锁
- **Timed_mutex**: 定时递归 Mutex 类，带超时的独占的互斥量，不能递归使用，除了递归，还可以在某个时间段里或者某个时刻到达之间获取该互斥量。当一个线程在临界区操作的时间非常长，可以用定时锁指定时间
- **Recursive_timed_mutex**: 定时递归 Mutex 类，带超时的递归互斥量，综合 timed_mutex 和 recursive_mutex

下面是一个使用基本锁的小例子，该程序会按顺序输出 5 对 enter 和 leave，分别对应 5 个线程。如果注释掉函数 f 中 mt 的 lock 和 unlock 函数，则线程没有同步，此时先输出 5 句 enter，然后 5s 后再输出 5 句 leave.

Mutex

```

#include <iostream>
#include <thread>
#include <string>
#include <mutex>
#include <vector>
using namespace std;

mutex mt; // 注意，全局的哟..

```

```

void f()
{
    mt.lock();
    cout << "EnterCriticalSection" << endl;
    this_thread::sleep_for(chrono::seconds(5));
    cout << "LeaveCriticalSection" << endl;
    mt.unlock();
}

int main()
{
    vector<thread> v(5);
    for (auto &i : v) i = thread(f);
    for (auto &i : v) i.join();
}

```

lock_guard<T_mutex> m(T_mutex) : 从对象初始化时到对象销毁时，所有权不可转移，对象生存期内不允许手动加锁和解锁

unique_lock<T_mutex> m(T_mutex) : 提供比 lock_guard 更灵活的方式，可以指定什么时候上锁，什么时候解锁，但是可以保证任何时候都会在析构时将锁释放掉。

```

void func()
{
    // 当作用域为函数的全部作用域时，lock_guard 与 unique_lock 起到了同样的作用
    lock_guard<std::mutex> locker(mutex);
    unique_lock<std::mutex> locker2(mutex2);

    // 但是在申请锁的时候不想上锁，lock_guard 就不行了，这是unique_lock 可以这样做
    unique_lock<std::mutex> locker3(mutex3, std::defer_lock);
    ... stuff
    locker3.lock();
    ... stuff..
    // 在处理完这些事之后 就不需要锁了，unique 可以unlock()
    locker3.unlock();
    ... stuff
    // 又需要了
    locker3.lock();

    // 而且unique_lock 可以被移动，而lock_guard 不可以，但是两者都不可以复制
    unique_lock<std::mutex> locker4 = std::move(locker3);
}

```

Recursice_mutex 为什么需要使用 Recursive_mutex, 看例子

```
struct Complex {
    std::mutex mutex;
    int i;

    Complex() : i(0) {}

    void mul(int x){
        std::lock_guard<std::mutex> lock(mutex);
        i *= x;
    }

    void div(int x){
        std::lock_guard<std::mutex> lock(mutex);
        i /= x;
    }

    void both(int x, int y){
        std::lock_guard<std::mutex> lock(mutex);
        mul(x);
        div(y);
    }
};

int main(){
    Complex complex;
    complex.both(32, 23);

    return 0;
}

// If you launch this application, you'll see that the program will never
terminates. The problem is very simple. In the both() function, the thread
acquires the lock and then calls the mul() function. In this function, the
threads tries to acquire the lock again, but the lock is already locked.
This is a case of deadlock. By default, a thread cannot acquire the same
mutex twice.
```

There is a simple solution to this problem: std::recursive_mutex. This mutex can be acquired several times by the same thread.

```
struct Complex {
    std::recursive_mutex mutex;
    int i;

    Complex() : i(0) {}

    void mul(int x){
```

```

        std :: lock_guard<std :: recursive_mutex> lock (mutex) ;
        i *= x;
    }

void div(int x){
    std :: lock_guard<std :: recursive_mutex> lock (mutex) ;
    i /= x;
}

void both(int x, int y){
    std :: lock_guard<std :: recursive_mutex> lock (mutex) ;
    mul(x);
    div(y);
}
};

//This time, the application works correctly.

```

Call once

Sometimes **you want a function to be called only once no matter the number of threads that are used**. Imagine a function that has two parts. The first part has to be called only once and the second has to be executed every time the function gets called. We can use the std::call_once function to fix this problem very easily.

```

std :: once_flag flag;

void do_something(){
    std :: call_once(flag, [](){ std :: cout << "Called once" << std :: endl; });

    std :: cout << "Called each time" << std :: endl;
}

int main(){
    std :: thread t1(do_something);
    std :: thread t2(do_something);
    std :: thread t3(do_something);
    std :: thread t4(do_something);

    t1.join();
    t2.join();
    t3.join();
    t4.join();

    return 0;
}

```

注意：

1. 多次获取互斥量可能会发生死锁，所以我们调用 std::recursive_mutex 递归锁，允许同一线程多次获得该锁，**一般不要使用递归锁**，原因：
 - 用到递归锁会使得程序的逻辑变复杂，使用到递归锁的程序一般可以简化
 - 递归锁比非递归锁效率低
 - 递归锁的可重入次数是有限的，超过也会报错
2. 可以使用带超时时间的互斥锁，避免阻塞在等待互斥锁上
3. unique_lock：是一个通用的互斥量封装类。与 lock_guard 不同，它还支持延迟加锁、时间锁、递归锁、锁所有权的转移并且还支持使用条件变量。这也是一个不可复制的类，但它是可以移动的类

Keep in mind that **locks are slow**. Indeed, when you use locks you make sections of the code sequential. If you want an **highly parallel application**, there are other solutions than locks that are performing much better.

5.5.5 条件变量

condition_variable ->

必须与 std::unique_lock 配合使用

条件变量 condition_variable 也可以进行线程之间的通信，当一个线程要等待另一个线程完成某个操作时，可以使用条件变量进行实现。条件变量可以将一个或多个线程进入阻塞状态，直到收到另外一个线程的通知，或者超时才能退出阻塞状态。

一个线程等待某个条件满足，其首先获得一个unique_lock 锁。该锁将会传递给wait() 方法，然后wait() 方法会释放互斥量并将该线程暂停，直到条件变量得到相应的信号。当接受到信号，线程被唤醒后，该锁就又被重新获得了。

另外一个线程发送信号使得条件满足。其通过调用notify_one() 来发送通知，会将处于阻塞状态的等待该条件获得信号的线程中的某一个线程（任意一个线程）恢复执行；还可以通过调用notify_all() 将等待该条件的所有线程唤醒。

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex x;
std::condition_variable cond;
bool ready = false;

bool IsReady() { return ready; }

void Run( int no )
{
    std::unique_lock<std::mutex> locker( x );
    while( !ready ) // 若标志位非true，阻塞当前线程
        cond.wait( locker ); // 解锁并睡眠，被唤醒后重新加锁
    // 以上两行代码等价于cond.wait( locker , &IsReady );
```

```

    // 第二个参数为谓词，亦可使用函数实现
    std::cout << "thread" << no << '\n';
}

int main()
{
    std::thread threads[8];
    for( int i = 0; i < 8; ++i )
        threads[i] = std::thread( Run, i );
    std::cout << "8 threads ready...\n";
    {
        std::unique_lock<std::mutex> locker(x); // 互斥加锁
        ready = true; // 设置全局标志位为true
        cond.notify_all(); // 唤醒所有线程
    } // 离开作用域，自动解锁；可将此复合语句块实现为函数
    // 基于区间的循环结构，对属于threads数组的所有元素t，执行循环体
    for( auto & t: threads )
        t.join();
    return 0;
}

```

conditon_variable_any ->

更加通用的条件变量，可以与任意型式的互斥锁配合使用，相比前者使用时会有额外的开销

5.5.6 期许与承诺

线程的返回值 为支持跨平台，`thread`类无属性字段保存线程函数的返回值。解决方案有三种：

- 使用指针类型的函数参数
- 使用期许`std::future`类模版
- 使用承诺`std::promise`类模版

指针型式参数

使用指针作为函数参数，获取线程计算结果

```

#include <iostream>
#include <vector>
#include <tuple>
#include <thread>
#include <mutex>
std::mutex x;
// 劳工线程类模板，处理T型数据对象
template< typename T > class Worker
{
public:

```

```

    explicit Worker( int no, T a = 0, T b = 0 ) : _no(no), _a(a), _b(b) { }
    void ThreadFunc( T * r ) { x.lock(); *r = _a + _b; x.unlock(); }
private:
    int _no; // 线程编号，非线程ID
    T _a, _b; // 保存在线程中的待处理数据
};

int main()
{
    // 定义能够存储8个三元组的向量v，元组首元素为指向劳工对象的指针
    // 次元素保存该劳工对象计算后的结果数据，尾元素为指向劳工线程对象的指针
    // 向量中的每个元素都表示一个描述线程运行的线程对象，
    // 该线程对象对应的执行具体任务的劳工对象，及该劳工对象运算后的返回值
    std::vector< std::tuple<Worker<int>*, int, std::thread*> > v( 8 );

    // 构造三元组向量，三元编号顺次为0、1、2
    for( int i = 0; i < 8; i++ )
        v[i] = std::make_tuple( new Worker<int>( i, i+1, i+2 ), 0, nullptr );

    // 输出处理前结果；使用std::get<n>(v[i])获取向量的第i个元组的第n个元素
    // 三元编号顺次为0、1、2，因而1号元保存的将是劳工对象运算后的结果
    for( int i = 0; i < 8; i++ )
        std::cout << "No." << i << ":" result = << std::get<1>(v[i]) << std
            :: endl;
    // 创建8个线程分别计算
    for( int i = 0; i < 8; i++ )
    {
        // 将劳工类成员函数绑定为线程函数，对应劳工对象绑定为执行对象
        // 将构造线程对象时传递的附加参数作为被绑定的线程函数的第一个参数
        // auto表示由编译器自动推断f的型式
        auto f = std::bind( &Worker<int>::ThreadFunc,
            std::get<0>( v[i] ), std::placeholders::_1 );

        // 动态构造线程对象，并保存到向量的第i个三元组中
        // 传递三元组的1号元地址，即将该地址作为线程函数的参数
        // 线程将在执行时将结果写入该地址
        // 此性质由绑定函数std::bind()使用占位符std::placeholders::_1指定
        // 线程对象为2号元，即三元组的最后一个元素
        std::get<2>( v[i] ) = new std::thread( f, &std::get<1>( v[i] ) );
    }
    for( int i = 0; i < 8; i++ )
    {
        // 等待线程结束
        std::get<2>( v[i] )->join();
        // 销毁劳工对象
        delete std::get<0>( v[i] ), std::get<0>( v[i] ) = nullptr;
        // 销毁线程对象
    }
}

```

```

        delete std::get<2>( v[i] ),     std::get<2>( v[i] ) = nullptr;
    }

// 输出线程计算后的结果
for( int i = 0; i < 8; i++ )
    std::cout << "No." << i << ":" result = " << std::get<1>(v[i]) << std
                           :: endl;
return 0;
}

```

期许 future

->

目的: 获取异步操作结果, 延迟引发线程的异步操作异常

使用方法:

- 定义期许模板类的期许对象
- 使用std::async() 函数的返回值初始化
- 调用期许对象的成员函数get() 获取线程返回值

使用期许对象获取线程返回值

```

#include <iostream>
#include <exception>
#include <thread>
#include <future>

unsigned long int CalculateFactorial( short int n )
{
    unsigned long int r = 1;
    if( n > 20 )
        throw std::range_error( "The number is too big." );
    for( short int i = 2; i <= n; i++ )
        r *= i;
    return r;
}

int main()
{
    short int n = 20;
    // 启动异步线程, 执行后台计算任务, 并返回std::future对象
    std::future<unsigned long int> f = std::async( CalculateFactorial, n );
    try
    {
        // 获取线程返回值, 若线程已结束, 立即返回, 否则等待该线程计算完毕
        // *** (重点) 若线程引发异常, 则延迟到std::future::get()或std::future::
        // wait()调用时引发, 在调用端处理异常
    }
}

```

```

        unsigned long int r = f.get();
        std::cout << n << " != " << r << std::endl;
    }
    catch( const std::range_error & e )
    {
        std::cerr << e.what() << std::endl;
    }
    return 0;
}

```

理论 std::future 可以用来获取异步任务的结果，因此可以把它当成一种简单的线程间同步的手段。std::future 通常由某个 **Provider** 创建，你可以把 Provider 想象成一个异步任务的提供者，Provider 在某个线程中设置共享状态的值，与该共享状态相关联的 std::future 对象调用 get（通常在另外一个线程中）获取该值，如果共享状态的标志不为 ready，则调用 std::future::get 会阻塞当前的调用者，直到 Provider 设置了共享状态的值（此时共享状态的标志变为 ready），std::future::get 返回异步任务的值或异常（如果发生了异常）。

一个有效 (valid) 的 std::future 对象通常由以下三种 Provider 创建，并和某个共享状态相关联。Provider 可以是函数或者类，其实我们前面都已经提到了，他们分别是：

- **std::async** 函数
- **std::promise::get_future**, `get_future` 为 `promise` 类的成员函数
- **std::packaged_task::get_future**, 此时 `get_future` 为 `packaged_task` 的成员函数

一个 std::future 对象只有在有效 (valid) 的情况下才有用 (useful)，由 std::future 默认构造函数创建的 future 对象不是有效的（除非当前非有效的 future 对象被 move 赋值另一个有效的 future 对象）。

在一个有效的 future 对象上调用 get 会阻塞当前的调用者，直到 Provider 设置了共享状态的值或异常（此时共享状态的标志变为 ready），std::future::get 将返回异步任务的值或异常（如果发生了异常）。

```

#include <iostream>           // std::cout
#include <future>             // std::async, std::future
#include <chrono>              // std::chrono::milliseconds

// a non-optimized way of checking for prime numbers:
bool is_prime (int x)
{
    for (int i=2; i<x; ++i) if (x%i==0) return false;
    return true;
}

int main ()
{
    // call function asynchronously:
    std::future<bool> fut = std::async (is_prime,444444443);
}

```

```

// do something while waiting for function to set future:
std::cout << "checking, please wait";
std::chrono::milliseconds span(100);
while (fut.wait_for(span) == std::future_status::timeout)
    std::cout << '.' << std::flush;

bool x = fut.get();      // retrieve return value

std::cout << "\n444444443" << (x ? "is" : "is not") << " prime.\n";

return 0;
}

//checking, please wait .....
//444444443 is prime.

```

承诺 promise

->

目的: 承诺对象允许期许对象获取线程对象创建的线程返回值

使用方法:

- 创建承诺std::promise<T> 对象
- 获取该承诺对象的**相关**期许std::future<T> 对象
- 创建线程对象，并传递承诺对象
- 线程函数内部通过承诺模板类的成员函数set_value()、set_value_at_thread_exit()、set_exception() 或set_exception_at_thread_exit() 设置值或异常
- 通过期许对象等待并获取异步操作结果

使用承诺对象获取线程返回值

```

#include <iostream>
#include <exception>
#include <thread>
#include <future>

unsigned long int CalculateFactorial( short int n )
{
    unsigned long int r = 1;
    if( n > 20 )
        throw std::range_error( "The number is too big." );
    for( short int i = 2; i <= n; i++ )
        r *= i;
    return r;
}

```

```

// CalculateFactorial() 函数的包装函数原型
void DoCalculateFactorial(std::promise<unsigned long int> && promise, short
                           int n);

int main()
{
    short int n = 6;
    std::promise<unsigned long int> p;      // 创建承诺对象
    std::future<unsigned long int> f = p.get_future(); // 获取相关期许对
                                                    // 象
    // 启动线程，执行 CalculateFactorial() 函数的包装函数
    std::thread t( DoCalculateFactorial, std::move( p ), n );
    t.detach();
    try
    {
        unsigned long int r = f.get();
        std::cout << n << "!=" << r << std::endl;
    }
    catch( std::range_error & e )
    {
        std::cerr << e.what() << std::endl;
    }
    return 0;
}

// 重点
// CalculateFactorial() 函数的包装函数实现
void DoCalculateFactorial(std::promise<unsigned long int> && promise, short
                           int n)
{
    try
    {
        // 设置线程返回值，供期许对象获取
        promise.set_value( CalculateFactorial( n ) );
    }
    catch( ... )
    {
        // 捕获全部异常，并在期许获取线程返回值时重新引发
        promise.set_exception( std::current_exception() );
    }
}

```

理论 promise 对象可以保存某一类型 T 的值，该值可被 future 对象读取（可能在另外一个线程中），因此 promise 也提供了一种线程同步的手段。在 promise 对象构造时可以和一个共享状态（通常是std::future）相关联，并可以在相关联的共享状态 (std::future) 上保存一个类型为 T

的值。

可以通过 `get_future` 来获取与该 `promise` 对象相关联的 `future` 对象，调用该函数之后，两个对象共享相同的共享状态 (shared state)

- **promise** 对象是异步 Provider，它可以在某一时刻设置共享状态的值。
- **future** 对象可以异步返回共享状态的值，或者在必要的情况下阻塞调用者并等待共享状态标志变为 ready，然后才能获取共享状态的值。

下面以一个简单的例子来说明上述关系

```
void print_int(std::future<int>& fut) {
    int x = fut.get(); // 获取共享状态的值.
    std::cout << "value:" << x << '\n'; // 打印 value: 10.
}

int main ()
{
    std::promise<int> prom; // 生成一个 std::promise<int> 对象.
    std::future<int> fut = prom.get_future(); // 和 future 关联.
    std::thread t(print_int, std::ref(fut)); // 将 future 交给另外一个线程t.
    prom.set_value(10); // 设置共享状态的值，此处和线程t保持同步.
    t.join();
    return 0;
}
```

packaged_task

`std::packaged_task` 包装一个可调用的对象，并且允许异步获取该可调用对象产生的结果，从包装可调用对象意义上来说，`std::packaged_task` 与 `std::function` 类似，只不过 `std::packaged_task` 将其包装的可调用对象的执行结果传递给一个 `std::future` 对象（该对象通常在另外一个线程中获取 `std::packaged_task` 任务的执行结果）。

`std::packaged_task` 对象内部包含了两个最基本元素，一、被包装的任务 (**stored task**)，任务 (task) 是一个可调用的对象，如函数指针、成员函数指针或者函数对象，二、共享状态 (**shared state**)，用于保存任务的返回值，可以通过 `std::future` 对象来达到异步访问共享状态的效果。

可以通过 `std::packaged_task::get_future` 来获取与共享状态相关联的 `std::future` 对象。在调用该函数之后，两个对象共享相同的共享状态，具体解释如下：

- **std::packaged_task** 对象是异步 Provider，它在某一时刻通过调用被包装的任务来设置共享状态的值。
- **std::future** 对象是一个异步返回对象，通过它可以获取共享状态的值，当然在必要的时候需要等待共享状态标志变为 ready.

`std::packaged_task` 的共享状态的生命周期一直持续到最后一个与之相关联的对象被释放或者销毁为止。下面一个小例子大致讲了 `std::packaged_task` 的用法：

```
#include <iostream> // std::cout
```

```

#include <future>           // std::packaged_task, std::future
#include <chrono>            // std::chrono::seconds
#include <thread>             // std::thread, std::this_thread::sleep_for

// count down taking a second for each value:
int countdown (int from, int to) {
    for (int i=from; i!=to; --i) {
        std::cout << i << '\n';
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
    std::cout << "Finished!\n";
    return from - to;
}

int main ()
{
    std::packaged_task<int(int,int)> task(countdown); // 设置 packaged_task
    std::future<int> ret = task.get_future(); // 获得与 packaged_task 共享状态
                                                相关联的 future 对象.

    std::thread th(std::move(task), 10, 0); // 创建一个新线程完成计数任务.

    int value = ret.get(); // 等待任务完成并获取结果.

    std::cout << "The countdown lasted for " << value << " seconds.\n";

    th.join();
    return 0;
}

// Result:
/*
10
9
8
7
6
5
4
3
2
1
Finished!
The countdown lasted for 10 seconds.
*/

```

参考

http://blog.csdn.net/jiange_zh/article/details/52542084
http://www.cnblogs.com/haippy/p/3280643.html
http://www.cnblogs.com/haippy/p/3279565.html
http://www.cnblogs.com/haippy/p/3239248.html

5.5.7 线程池

应用场景

线程池通常适合下面的几个场合：

- 单位时间内处理的任务数较多，且每个任务的执行时间较短
- 对实时性要求较高的任务，如果接受到任务后在创建线程，再执行任务，可能满足不了实时要求，因此必须采用线程池进行预创建

实现

C++11 加入了线程库，从此告别了标准库不支持并发的历史。然而 c++ 对于多线程的支持还是比较低级，稍微高级一点的用法都需要自己去实现，譬如线程池、信号量等。线程池 (*thread pool*) 这个东西，在面试上多次被问到，一般的回答都是：“管理一个任务队列，一个线程队列，然后每次取一个任务分配给一个线程去做，循环往复。”貌似没有问题吧。但是写起程序来的时候就出问题了。

```
#ifndef ILOVERS_THREAD_POOL_H
#define ILOVERS_THREAD_POOL_H

#include <iostream>
#include <functional>
#include <thread>
#include <condition_variable>
#include <future>
#include <atomic>
#include <vector>
#include <queue>

// 命名空间
namespace ilovers {
    class TaskExecutor;
}

class ilovers::TaskExecutor{
    using Task = std::function<void ()>;
private:
    // 线程池
    std::vector<std::thread> pool;
```

```

// 任务队列
std::queue<Task> tasks;
// 同步
std::mutex m_task;
std::condition_variable cv_task;
// 是否关闭提交
std::atomic<bool> stop;

public:
    // 构造
    TaskExecutor(size_t size = 4): stop {false} {
        size = size < 1 ? 1 : size;
        for (size_t i = 0; i < size; ++i) {
            // 隐式调用 thread 的构造函数
            pool.emplace_back(&TaskExecutor::schedual, this); // push_back(std::thread {...})
        }
    }

    // 析构
    ~TaskExecutor() {
        for (std::thread& thread : pool) {
            thread.detach(); // 让线程“自生自灭”
            // thread.join(); // 等待任务结束，前提：线程一定会执行完
        }
    }

    // 停止任务提交
    void shutdown() {
        this->stop.store(true);
    }

    // 重启任务提交
    void restart() {
        this->stop.store(false);
    }

    // 提交一个任务
    template<class F, class ... Args>
    auto commit(F&& f, Args&&... args) -> std::future<decltype(f(args...))> {
        if (stop.load()) { // stop == true ???
            throw std::runtime_error("task_executor_have_closed_commit.");
        }

        using ResType = decltype(f(args...)); // typename std::

```

```

        result_of<F(Args...)>::type, 函数 f 的返回值类型
    auto task = std::make_shared<std::packaged_task<ResType()>>(std::
        bind(std::forward<F>(f), std::forward<Args>(args)...)); // wtf !
    {
        // 添加任务到队列
        std::lock_guard<std::mutex> lock {m_task};
        tasks.emplace([task](){ // push(Task{...})
            (*task)();
        });
    }
    cv_task.notify_all(); // 唤醒线程执行

    std::future<ResType> future = task->get_future();
    return future;
}

private:
    // 获取一个待执行的 task
    Task get_one_task(){
        std::unique_lock<std::mutex> lock {m_task};
        cv_task.wait(lock, [this](){ return !tasks.empty(); }); // wait
        // 直到有 task
        Task task {std::move(tasks.front())}; // 取一个 task
        tasks.pop();
        return task;
    }

    // 任务调度
    void schedual(){
        while(true){
            if(Task task = get_one_task()){
                task(); // 
            } else {
                // return; // done
            }
        }
    }
};

#endif

void f()
{
    std::cout << "hello , f !" << std::endl;
}

struct G{

```

```

int operator()() {
    std::cout << "hello ,ug!" << std::endl;
    return 42;
}

};

int main() {
    try {
        ilovers::TaskExecutor executor {10};

        std::future<void> ff = executor.commit(f);
        std::future<int> fg = executor.commit(G{});
        std::future<std::string> fh = executor.commit([]()>std::string {
            std::cout << "hello ,uh!" << std::endl; return "hello ,fh!";
        });

        executor.shutdown();

        ff.get();
        std::cout << fg.get() << " " << fh.get() << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(5));
        executor.restart(); // 重启任务
        executor.commit(f).get(); // 

        std::cout << "end..." << std::endl;
        return 0;
    } catch (std::exception& e) {
        std::cout << "some unhappy happened..." << e.what() << std::endl;
    }
}

```

实现原理

回调思想。

接着前面的废话。‘‘管理一个任务队列，一个线程队列，然后每次取一个任务分配给一个线程去做，循环往复。’’这个思路有神马问题？线程池一般要复用线程，所以如果是取一个 task 分配给某一个 thread，执行完之后再重新分配，在语言层面基本都是不支持的：一般语言的 thread 都是执行一个固定的 task 函数，执行完毕线程也就结束了（至少 c++ 是这样）。so 要如何实现 task 和 thread 的分配呢？

让每一个 **thread** 都去执行调度函数：循环获取一个 **task**，然后执行之。

idea 是不是很赞！保证了 thread 函数的唯一性，而且复用线程执行 task。

参考

各种版本实现：https://github.com/lizhenghn123/zl_threadpool

实现 C++11: <https://github.com/ctzhenghua/ThreadPool>

实现 C++11: <http://blog.csdn.net/zdarks/article/details/46994607>

<http://blog.csdn.net/ockie/article/details/51606468>

半同步半异步线程池: <http://www.cnblogs.com/kaishan1990/p/5273237.html>

5.6 References

5.6.1 thread

constructing Threads

```
// constructing threads
#include <iostream>           // std::cout
#include <atomic>              // std::atomic
#include <thread>              // std::thread
#include <vector>               // std::vector

std::atomic<int> global_counter (0);

void increase_global (int n) { for (int i=0; i<n; ++i) ++global_counter; }

void increase_reference (std::atomic<int>& variable, int n) { for (int i =0; i<n; ++i) ++variable; }

struct C : std::atomic<int>
{
    C() : std::atomic<int>(0) {}
    void increase_member (int n) { for (int i=0; i<n; ++i) fetch_add(1); }
};

int main ()
{
    std::vector<std::thread> threads;

    std::cout << "increase_global_counter with 10 threads ... \n";
    for (int i=1; i<=10; ++i)
        threads.push_back(std::thread(increase_global,1000));

    std::cout << "increase counter (foo) with 10 threads using reference ...
    ... \n";
    std::atomic<int> foo(0);
    for (int i=1; i<=10; ++i)
        threads.push_back(std::thread(increase_reference ,std::ref(foo)
            ,1000));
}
```

```

    std::cout << "increase_counter_(bar)_with_10_threads_using_member...\n";
}

C bar;
for (int i=1; i<=10; ++i)
    threads.push_back(std::thread(&C::increase_member, std::ref(bar),
        1000));

std::cout << "synchronizing_all_threads...\n";
for (auto& th : threads) th.join();

std::cout << "global_counter:" << global_counter << '\n';
std::cout << "foo:" << foo << '\n';
std::cout << "bar:" << bar << '\n';

return 0;
}

//Output///////////////
/*
    increase global counter using 10 threads...
    increase counter (foo) with 10 threads using reference...
    increase counter (bar) with 10 threads using member...
    synchronizing all threads...
    global_counter: 10000
    foo: 10000
    bar: 10000
*/

```

Detach thread

Detaches the thread represented by the object from the calling thread, allowing them to execute independently from each other.

Both threads continue without blocking nor synchronizing in any way. Note that when either one ends execution, its resources are released.

After a call to this function, the thread object becomes non-joinable and can be destroyed safely.

```

#include <iostream>           // std::cout
#include <thread>             // std::thread, std::this_thread::sleep_for
#include <chrono>              // std::chrono::seconds

void pause_thread(int n)
{
    std::this_thread::sleep_for (std::chrono::seconds(n));
    std::cout << "pause_of_" << n << " seconds ended\n";
}

```

```

int main()
{
    std::cout << "Spawning and detaching 3 threads...\n";
    std::thread (pause_thread,1).detach();
    std::thread (pause_thread,2).detach();
    std::thread (pause_thread,3).detach();
    std::cout << "Done spawning threads.\n";

    std::cout << "(the main thread will now pause for 5 seconds)\n";
    // give the detached threads time to finish (but not guaranteed!):
    pause_thread(5);
    return 0;
}

//////////OutPut///////////
/*
    Spawning and detaching 3 threads...
    Done spawning threads.
    (the main thread will now pause for 5 seconds)
    pause of 1 seconds ended
    pause of 2 seconds ended
    pause of 3 seconds ended
    pause of 5 seconds ended
*/

```

Get thread id

Returns the thread id.

If the thread object is joinable, the function returns a value that uniquely identifies the thread.

If the thread object is not joinable, the function returns a default-constructed object of member type `thread::id`.

```

// thread::get_id / this_thread::get_id
#include <iostream>           // std::cout
#include <thread>             // std::thread, std::thread::id, std::
                             //   this_thread::get_id
#include <chrono>              // std::chrono::seconds

std::thread::id main_thread_id = std::this_thread::get_id();

void is_main_thread()
{
    if ( main_thread_id == std::this_thread::get_id() )
        std::cout << "This is the main thread.\n";
    else
        std::cout << "This is not the main thread.\n";
}

```

```

int main()
{
    is_main_thread();
    std::thread th(is_main_thread);
    th.join();
}

//////////Output///////////
/*
    This is the main thread.
    This is not the main thread.
*/

```

Join thread

The function returns when the thread execution has completed.

This synchronizes the moment this function returns with the completion of all the operations in the thread: This blocks the execution of the thread that calls this function until the function called on construction returns (if it hasn't yet).

After a call to this function, the thread object becomes non-joinable and can be destroyed safely.

```

// example for thread::join
#include <iostream>           // std::cout
#include <thread>             // std::thread, std::this_thread::sleep_for
#include <chrono>              // std::chrono::seconds

void pause_thread(int n)
{
    std::this_thread::sleep_for(std::chrono::seconds(n));
    std::cout << "pause of " << n << " seconds ended\n";
}

int main()
{
    std::cout << "Spawning 3 threads...\n";
    std::thread t1(pause_thread, 1);
    std::thread t2(pause_thread, 2);
    std::thread t3(pause_thread, 3);
    std::cout << "Done spawning threads. Now waiting for them to join:\n";
    t1.join();
    t2.join();
    t3.join();
    std::cout << "All threads joined!\n";

    return 0;
}

```

```

    }

///////////Output(after 3 seconds)
///////////



/*
Spawning 3 threads...
Done spawning threads. Now waiting for them to join:
pause of 1 seconds ended
pause of 2 seconds ended
pause of 3 seconds ended
All threads joined!
*/

```

Check if joinable

Returns whether the thread object is joinable.

A thread object is joinable if it represents a thread of execution.

A thread object is not joinable in any of these cases:

- if it was default-constructed.
- if it has been moved from (either constructing another thread object, or assigning to it).
- if either of its members join or detach has been called.

```

// example for thread::joinable
#include <iostream>           // std::cout
#include <thread>             // std::thread

void mythread()
{
    // do stuff...
}

int main()
{
    std::thread foo;
    std::thread bar(mythread);

    std::cout << "Joinable after construction:\n" << std::boolalpha;
    std::cout << "foo:" << foo.joinable() << '\n';
    std::cout << "bar:" << bar.joinable() << '\n';

    if (foo.joinable()) foo.join();
    if (bar.joinable()) bar.join();

    std::cout << "Joinable after joining:\n" << std::boolalpha;
    std::cout << "foo:" << foo.joinable() << '\n';

```

```

        std::cout << "bar:" << bar.joinable() << '\n';

    return 0;
}

//////////////////Output(after 3 seconds)///////////////////
/*
Joinable after construction:
foo: false
bar: true
Joinable after joining:
foo: false
bar: false
*/

```

Move-assign thread::operator=

If the object is currently not joinable, it acquires the thread of execution represented by rhs (if any).

If it is joinable, terminate() is called.

After the call, rhs no longer represents any thread of execution (as if default-constructed).

thread objects cannot be copied (2).

```

// example for thread::operator=
#include <iostream>           // std::cout
#include <thread>             // std::thread, std::this_thread::sleep_for
#include <chrono>              // std::chrono::seconds

void pause_thread(int n)
{
    std::this_thread::sleep_for(std::chrono::seconds(n));
    std::cout << "pause of " << n << " seconds ended\n";
}

int main()
{
    std::thread threads[5];           // default-constructed
                                     // threads

    std::cout << "Spawning 5 threads...\n";
    for (int i=0; i<5; ++i)
        threads[i] = std::thread(pause_thread, i+1); // move-assign threads

    std::cout << "Done spawning threads. Now waiting for them to join:\n";
    for (int i=0; i<5; ++i)
        threads[i].join();

    std::cout << "All threads joined!\n";
}

```

```

        return 0;
    }

//////////////////Output(after 5 seconds)/////////////////////////////
/*
    Spawning 5 threads...
    Done spawning threads. Now waiting for them to join:
    pause of 1 seconds ended
    pause of 2 seconds ended
    pause of 3 seconds ended
    pause of 4 seconds ended
    pause of 5 seconds ended
    All threads joined!
*/

```

5.6.2 atomic

Objects of atomic types contain a value of a particular type (T).

The main characteristic of **atomic objects** is that **access to this contained value** from different threads **cannot cause data races** (i.e., doing that is well-defined behavior, with accesses properly sequenced). Generally, for all other objects, the possibility of causing a data race for accessing the same object concurrently qualifies the operation as undefined behavior.

Additionally, atomic objects have the ability to synchronize access to other non-atomic objects in their threads by specifying different memory orders.

5.6.3 mutex

Header with facilities that allow mutual exclusion (mutex) of concurrent execution of critical sections of code, allowing to explicitly **avoid data races**.

It contains mutex types, lock types and specific functions:

- **Mutex types** are lockable types used **to protect access to a critical section of code**: locking a mutex **prevents other threads from locking it** (exclusive access) until it is unlocked: mutex, recursive_mutex, timed_mutex, recursive_timed_mutex.
- **Locks** are objects that manage a mutex by associating its access to their own lifetime: lock_guard, unique_lock.
- **Functions** to lock multiple mutexes simultaneously (try_lock, lock) and to directly prevent concurrent execution of a specific function (call_once).

Mutex types

1. **mutex**: A mutex is a **lockable object** that is designed to **signal** when critical sections of code need exclusive access, preventing other threads with the same protection from executing concurrently and access the same memory locations.

mutex objects provide exclusive ownership and do not support recursivity (i.e., a thread shall not lock a mutex it already owns) – see recursive_mutex for an alternative class that does.

It is guaranteed to be a standard-layout class.

```
// mutex example
#include <iostream>           // std::cout
#include <thread>             // std::thread
#include <mutex>              // std::mutex

std::mutex mtx;               // mutex for critical section

void print_block (int n, char c)
{
    // critical section (exclusive access to std::cout signaled by
    // locking mtx):
    mtx.lock ();
    for (int i=0; i<n; ++i) { std::cout << c; }
    std::cout << '\n';
    mtx.unlock ();
}

int main ()
{
    std::thread th1 (print_block, 50, '*');
    std::thread th2 (print_block, 50, '$');

    th1.join ();
    th2.join ();

    return 0;
}

//////// Possible output (order of lines may vary, but characters are
// never mixed):///
/*
 ****
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
*/
```

2. **recursive_mutex**: A recursive mutex is a lockable object, just like mutex, but allows the same thread to acquire multiple levels of ownership over the mutex object.

This allows to lock (or try-lock) the mutex object from a thread that is already locking it, acquiring

a new level of ownership over the mutex object: the thread will remain locked until member unlock is called as many times as this level of ownership.

It is guaranteed to be a standard-layout class.

3. **timed_mutex**: A timed mutex is a time lockable object that is designed to signal when critical sections of code need exclusive access, just like a regular mutex, but additionally supporting timed try-lock requests.

As such, a timed_mutex has two additional members: try_lock_for and try_lock_until.

It is guaranteed to be a standard-layout class.

4. **recursive_timed_mutex**: A recursive timed mutex combines both the features of recursive_mutex and the features of timed_mutex into a single class: it supports both acquiring multiple lock levels by a single thread and also timed try-lock requests.

It is guaranteed to be a standard-layout class.

Locks

1. **lock_guard**: A lock guard is an object that manages a mutex object by keeping it always locked. On construction, the mutex object is locked by the calling thread, and on destruction, the mutex is unlocked. It is the simplest lock, and is specially useful as an object with automatic duration that lasts until the end of its context. In this way, it guarantees the mutex object is properly unlocked in case an exception is thrown.

Note though that the lock_guard object does not manage the lifetime of the mutex object in any way: the duration of the mutex object shall extend at least until the destruction of the lock_guard that locks it.

```
// lock_guard example
#include <iostream>           // std::cout
#include <thread>             // std::thread
#include <mutex>              // std::mutex, std::lock_guard
#include <stdexcept>          // std::logic_error

std::mutex mtx;

void print_even (int x)
{
    if (x%2==0) std::cout << x << " is even\n";
    else throw (std::logic_error("not even"));
}

void print_thread_id (int id)
{
    try
    {
        // using a local lock_guard to lock mtx guarantees unlocking
        // on destruction / exception:
    }
}
```

```

        std :: lock_guard<std :: mutex> lck ( mtx );
        print_even( id );
    }
    catch ( std :: logic_error& )
    {
        std :: cout << "[ exception caught ]\n";
    }
}

int main ()
{
    std :: thread threads[10];
    // spawn 10 threads:
    for ( int i=0; i<10; ++i )
        threads[ i ] = std :: thread( print_thread_id , i+1 );

    for ( auto& th : threads ) th.join();

    return 0;
}

//////////////////Output/////////////////
/*
[ exception caught ]
thread #2
[ exception caught ]
thread #4
[ exception caught ]
thread #6
[ exception caught ]
thread #8
[ exception caught ]
thread #10
*/

```

2. unique_lock: A unique lock is an object that manages a mutex object with unique ownership in both states: locked and unlocked.

On construction (or by move-assigning to it), the object acquires a mutex object, for whose locking and unlocking operations becomes responsible.

The object supports both states: locked and unlocked.

This class guarantees an unlocked status on destruction (even if not called explicitly). Therefore it is especially useful as an object with automatic duration, as it guarantees the mutex object is properly unlocked in case an exception is thrown.

Note though, that the unique_lock object does not manage the lifetime of the mutex object in any way: the duration of the mutex object shall extend at least until the destruction of the unique_lock

that manages it.

```
// unique_lock example
#include <iostream>           // std::cout
#include <thread>             // std::thread
#include <mutex>              // std::mutex, std::unique_lock

std::mutex mtx;                // mutex for critical section

void print_block (int n, char c)
{
    // critical section (exclusive access to std::cout signaled by
    // lifetime of lck):
    std::unique_lock<std::mutex> lck (mtx);
    for (int i=0; i<n; ++i) { std::cout << c; }
    std::cout << '\n';
}

int main ()
{
    std::thread th1 (print_block,50,'*');
    std::thread th2 (print_block,50,'$');

    th1.join();
    th2.join();

    return 0;
}

///Output:Possible output (order of lines may vary , but characters
// are never mixed):///
/*
 ****
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

Functions

1. **call_once:** Calls fn passing args as arguments, unless another thread has already executed (or is currently executing) a call to call_once with the same flag.

If another thread is already actively executing a call to call_once with the same flag, it causes a passive execution: Passive executions do not call fn but do not return until the active execution itself has returned, and all visible side effects are synchronized at that point among all concurrent calls to this function with the same flag.

If an active call to call_once ends by throwing an exception (which is propagated to its calling thread) and passive executions exist, one is selected among these passive executions, and called to

be the new active call instead.

Note that once an active execution has returned, all current passive executions and future calls to `call_once` (with the same flag) also return without becoming active executions.

The active execution uses decay copies of the lvalue or rvalue references of `fn` and `args`, ignoring the value returned by `fn`.

```
// call_once example
#include <iostream>           // std::cout
#include <thread>             // std::thread, std::this_thread::sleep_for
#include <chrono>              // std::chrono::milliseconds
#include <mutex>               // std::call_once, std::once_flag

int winner;
void set_winner (int x) { winner = x; }
std::once_flag winner_flag;

void wait_1000ms (int id)
{
    // count to 1000, waiting 1ms between increments:
    for (int i=0; i<1000; ++i)
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
    // claim to be the winner (only the first such call is executed):
    std::call_once (winner_flag, set_winner, id);
}

int main ()
{
    std::thread threads[10];
    // spawn 10 threads:
    for (int i=0; i<10; ++i)
        threads[i] = std::thread(wait_1000ms, i+1);

    std::cout << "waiting for the first among 10 threads to count
1000 ms...\n";

    for (auto& th : threads) th.join();
    std::cout << "winner thread:" << winner << '\n';

    return 0;
}

//////Output: Possible output (winner may vary)://////
/*
    waiting for the first among 10 threads to count 1000 ms...
    winner thread: 2
*/
```

2. **try_lock**:

3. **lock**:

5.6.4 condition_variable

A condition variable is an object able to block the calling thread until notified to resume.

It uses a unique_lock (over a mutex) to lock the thread when one of its wait functions is called. The thread remains blocked until woken up by another thread that calls a notification function on the same condition_variable object.

Objects of type condition_variable always use unique_lock<mutex> to wait: for an alternative that works with any kind of lockable type, see condition_variable_any

```
// condition_variable example
#include <iostream>           // std::cout
#include <thread>             // std::thread
#include <mutex>              // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void print_id (int id)
{
    std::unique_lock<std::mutex> lck (mtx);
    while (!ready) cv.wait(lck);
    // ...
    std::cout << "thread" << id << '\n';
}

void go()
{
    std::unique_lock<std::mutex> lck (mtx);
    ready = true;
    cv.notify_all();
}

int main ()
{
    std::thread threads[10];
    // spawn 10 threads:
    for (int i=0; i<10; ++i)
        threads[i] = std::thread(print_id, i);

    std::cout << "10 threads ready to race...\n";
    go();                      // go!
}
```

```

        for (auto& th : threads) th.join();

        return 0;
    }

///////////////////Output///////////////////
/*
10 threads ready to race...
thread 2
thread 0
thread 9
thread 4
thread 6
thread 8
thread 7
thread 5
thread 3
thread 1
*/

```

notify_one Unblocks one of the threads currently waiting for this condition.

If no threads are waiting, the function does nothing.

If more than one, it is unspecified which of the threads is selected.

```

// condition_variable::notify_one
#include <iostream>           // std::cout
#include <thread>             // std::thread
#include <mutex>              // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable

std::mutex mtx;
std::condition_variable produce, consume;

int cargo = 0;      // shared value by producers and consumers

void consumer ()
{
    std::unique_lock<std::mutex> lck(mtx);
    while (cargo==0)
        consume.wait(lck);

    std::cout << cargo << '\n';
    cargo=0;
    produce.notify_one();
}
```

```

void producer (int id)
{
    std :: unique_lock<std :: mutex> lck (mtx) ;
    while (cargo!=0)
        produce . wait (lck) ;

    cargo = id ;
    consume . notify_one () ;
}

int main ()
{
    std :: thread consumers [10] , producers [10];
    // spawn 10 consumers and 10 producers:
    for (int i=0; i<10; ++i)
    {
        consumers [i] = std :: thread (consumer) ;
        producers [i] = std :: thread (producer , i+1);
    }

    // join them back:
    for (int i=0; i<10; ++i)
    {
        producers [i] . join () ;
        consumers [i] . join () ;
    }

    return 0;
}

```

Wait for timeout or until notified The execution of the current thread (which shall have locked lck's mutex) is blocked during rel_time, or until notified (if the latter happens first).

At the moment of blocking the thread, the function automatically calls lck.unlock(), allowing other locked threads to continue.

Once notified or once rel_time has passed, the function unblocks and calls lck.lock(), leaving lck in the same state as when the function was called. Then the function returns (notice that this last mutex locking may block again the thread before returning).

Generally, the function is notified to wake up by a call in another thread either to member notify_one or to member notify_all. But certain implementations may produce spurious wake-up calls without any of these functions being called. Therefore, users of this function shall ensure their condition for resumption is met.

If pred is specified (2), the function only blocks if pred returns false, and notifications can only unblock the thread when it becomes true (which is especially useful to check against spurious wake-up calls). It behaves as if implemented as:

```
return wait_until(lck, chrono::steady_clock::now() + rel_time, std::move(pred));
```

Parameters:

- **lck**:A unique_lock object whose mutex object is currently locked by this thread. All concurrent calls to wait member functions of this object shall use the same underlying mutex object (as returned by lck.mutex()).
- **rel_time**:The maximum time span during which the thread will block waiting to be notified. duration is an object that represents a specific relative time.
- **pred**:A callable object or function that takes no arguments and returns a value that can be evaluated as a bool. This is called repeatedly until it evaluates to true.

```
// condition_variable :: wait_for example
#include <iostream>           // std :: cout
#include <thread>             // std :: thread
#include <chrono>              // std :: chrono :: seconds
#include <mutex>                // std :: mutex, std :: unique_lock
#include <condition_variable> // std :: condition_variable, std :: cv_status

std :: condition_variable cv;

int value;

void read_value()
{
    std :: cin >> value;
    cv.notify_one();
}

int main ()
{
    std :: cout << "Please , enter an integer (I'll be printing dots): ";
    std :: thread th (read_value);

    std :: mutex mtx;
    std :: unique_lock<std :: mutex> lck (mtx);
    while (cv.wait_for (lck, std :: chrono :: seconds (1)) == std :: cv_status :: timeout)
    {
        std :: cout << '.';
    }
    std :: cout << "You entered: " << value << '\n';

    th . join();

    return 0;
}
```

```
//////////////////Output///////////////////////////
/*
    Please , enter an integer (I'll be printing dots): ....1.0
    You entered: 10
*/
```

Data Races The function performs three atomic operations:

- The initial unlocking of lck and simultaneous entry into the waiting state.
- The unblocking of the waiting state.
- The locking of lck before returning.

Atomic operations on the object are ordered according to a single total order, with the three atomic operations in this function happening in the same relative order as above.

5.6.5 Futures

The standard library provides facilities to obtain values that are returned and to catch exceptions that are thrown by asynchronous tasks (i.e. functions launched in separate threads). These values are communicated in a shared state, in which the asynchronous task may write its return value or store an exception, and which may be examined, waited for, and otherwise manipulated by other threads that hold instances of std::future or std::shared_future that reference that shared state.

future

A future is an object that can retrieve a value from some provider object or function, properly synchronizing this access if in different threads.

”Valid” futures are future objects associated to a shared state, and are constructed by calling one of the following functions:

- `async`
- `promise::get_future`
- `packaged_task::get_future`

future objects are only useful when they are valid. Default-constructed future objects are not valid (unless move-assigned a valid future).

Calling `future::get` on a valid future blocks the thread until the provider makes the shared state ready (either by setting a value or an exception to it). This way, two threads can be synchronized by one waiting for the other to set a value.

The lifetime of the shared state lasts at least until the last object with which it is associated releases it or is destroyed. Therefore, if associated to a future, the shared state can survive the object from which it was obtained in the first place (if any).

```

// future example
#include <iostream>           // std::cout
#include <future>             // std::async, std::future
#include <chrono>              // std::chrono::milliseconds

// a non-optimized way of checking for prime numbers:
bool is_prime (int x)
{
    for (int i=2; i<x; ++i) if (x%i==0) return false;
    return true;
}

int main ()
{
    // call function asynchronously:
    std::future<bool> fut = std::async (is_prime,444444443);

    // do something while waiting for function to set future:
    std::cout << "checking, please wait";
    std::chrono::milliseconds span (100);
    while (fut.wait_for(span)==std::future_status::timeout)
        std::cout << '.' << std::flush;

    bool x = fut.get();      // retrieve return value

    std::cout << "\n444444443" << (x?"is ":"is not") << " prime.\n";

    return 0;
}

////Output:(may take more or less time)///////////
/*
    checking, please wait .....
    444444443 is prime.
*/

```

Get value Returns the value stored in the shared state (or throws its exception) when the shared state is ready.

If the shared state is not yet ready (i.e., the provider has not yet set its value or exception), the function blocks the calling thread and waits until it is ready.

Once the shared state is ready, the function unblocks and returns (or throws) releasing its shared state. This makes the future object no longer valid: this member function shall be called once at most for every future shared state.

All visible side effects are synchronized between the point the provider makes the shared state ready

and the return of this function.

The member of the void specialization (3) does not return any value, but still waits for the shared state to become ready and releases it.

Get shared future Returns a `shared_future` object that acquires the shared state of the future object. The future object (`*this`) is left with no shared state (as if default-constructed) and is no longer valid. A `shared_future` object that acquires the shared state of `*this`, as if constructed as:

```
shared_future<T>(std::move(*this))
```

T is the type of the value (the template parameter of future).

```
// future::share
#include <iostream>           // std::cout
#include <future>             // std::async, std::future, std::shared_future

int get_value() { return 10; }

int main()
{
    std::future<int> fut = std::async(get_value);
    std::shared_future<int> shfut = fut.share();

    // shared futures can be accessed multiple times:
    std::cout << "value:" << shfut.get() << '\n';
    std::cout << "its double:" << shfut.get()*2 << '\n';

    return 0;
}

//////////Output///////////
/*
    value: 10
    its double: 20
*/
```

Check for valid shared state Returns whether the future object is currently associated with a shared state.

For default-constructed future objects, this function returns false (unless move-assigned a valid future).

Futures can only be initially constructed with valid shared states by certain provider functions, such as `async`, `promise::get_future` or `packaged_task::get_future`.

Once the value of the shared state is retrieved with `future::get`, calling this function returns false (unless move-assigned a new valid future).

```
// future::valid
#include <iostream>           // std::cout
```

```

#include <future>           // std::async, std::future
#include <utility>          // std::move

int get_value() { return 10; }

int main()
{
    std::future<int> foo, bar;
    foo = std::async (get_value());
    bar = std::move(foo);

    if (foo.valid())
        std::cout << "foo's value:" << foo.get() << '\n';
    else
        std::cout << "foo is not valid\n";

    if (bar.valid())
        std::cout << "bar's value:" << bar.get() << '\n';
    else
        std::cout << "bar is not valid\n";

    return 0;
}

/*
    foo is not valid
    bar's value: 10
*/

```

Wait for ready Waits for the shared state to be ready.

If the shared state is not yet ready (i.e., the provider has not yet set its value or exception), the function blocks the calling thread and waits until it is ready.

Once the shared state is ready, the function unblocks and returns without reading its value nor throwing its set exception (if any).

All visible side effects are synchronized between the point the provider makes the shared state ready and the return of this function.

```

// future::wait
#include <iostream>           // std::cout
#include <future>              // std::async, std::future
#include <chrono>              // std::chrono::milliseconds

// a non-optimized way of checking for prime numbers:
bool is_prime (int x)
{
    for (int i=2; i<x; ++i) if (x%i==0) return false;

```

```

        return true;
    }

int main ()
{
    // call function asynchronously:
    std::future<bool> fut = std::async (is_prime,194232491);

    std::cout << "checking ... \n";
    fut.wait();

    std::cout << "\n194232491 ";
    if (fut.get())      // guaranteed to be ready (and not block) after
        wait returns
    std::cout << "is_prime.\n";
    else
    std::cout << "is_not_prime.\n";

    return 0;
}

//////////Output/////////
/*
    checking ...
    194232491 is prime.
*/

```

Wait for ready during time span Waits for the shared state to be ready for up to the time specified by `rel_time`.

If the shared state is not yet ready (i.e., the provider has not yet set its value or exception), the function blocks the calling thread and waits until it is ready or until `rel_time` has elapsed, whichever happens first.

When the function returns because its shared state is made ready, the value or exception set on the shared state is not read, but all visible side effects are synchronized between the point the provider makes the shared state ready and the return of this function.

If the shared state contains a deferred function (such as future objects returned by `async`), the function does not block, returning immediately with a value of `future_status::deferred`.

```

// future::wait_for
#include <iostream>           // std::cout
#include <future>             // std::async, std::future
#include <chrono>             // std::chrono::milliseconds

// a non-optimized way of checking for prime numbers:
bool is_prime (int x)
{

```

```

        for (int i=2; i<x; ++i) if (x%i==0) return false;
        return true;
    }

    int main ()
    {
        // call function asynchronously:
        std::future<bool> fut = std::async (is_prime,700020007);

        // do something while waiting for function to set future:
        std::cout << "checking , please wait";
        std::chrono::milliseconds span (100);
        while (fut.wait_for(span)==std::future_status::timeout)
            std::cout << '.';

        bool x = fut.get();

        std::cout << "\n700020007" << (x?" is ":" is not") << " prime.\n";

        return 0;
    }

//////////Output///////////
/*
    checking , please wait .....
    700020007 is prime.
*/

```

shared_future

A shared_future object behaves like a future object, except that it can be copied, and that more than one shared_future can share ownership over their end of a shared state. They also allow the value in the shared state to be retrieved multiple times once ready.

shared_future objects can be implicitly converted from future objects (see its constructor), or explicitly obtained by calling future::share. In both cases, the future object from which it is obtained transfers its association with the shared state to the shared_future and becomes itself non-valid.

The lifetime of the shared state lasts at least until the last object with which it is associated is destroyed. Retrieving the value from a shared_future (with member get) does not release its ownership over the shared state (unlike with futures). Therefore, if associated to shared_future objects, the shared state can survive the object from which it was obtained in the first place (if any).

promise

A promise is an object that can store a value of type T to be retrieved by a future object (possibly in another thread), offering a synchronization point.

On construction, promise objects are associated to a new shared state on which they can store either a value of type T or an exception derived from std::exception.

This shared state can be associated to a future object by calling member get_future. After the call, both objects share the same shared state:

- The promise object is the asynchronous provider and is expected to set a value for the shared state at some point.
- The future object is an asynchronous return object that can retrieve the value of the shared state, waiting for it to be ready, if necessary.

The lifetime of the shared state lasts at least until the last object with which it is associated releases it or is destroyed. Therefore it can survive the promise object that obtained it in the first place if associated also to a future.

```
// promise example
#include <iostream>           // std::cout
#include <functional>          // std::ref
#include <thread>              // std::thread
#include <future>               // std::promise, std::future

void print_int (std::future<int>& fut)
{
    int x = fut.get();
    std::cout << "value:" << x << '\n';
}

int main ()
{
    std::promise<int> prom;           // create promise

    std::future<int> fut = prom.get_future(); // engagement with future

    std::thread th1 (print_int, std::ref(fut)); // send future to new
                                                // thread

    prom.set_value (10);             // fulfill promise
    // (synchronizes with getting the future)
    th1.join();
    return 0;
}

//////////Output/////////
/*
    value:10
*/
```

Set value Stores val as the value in the shared state, which becomes ready.

If a future object that is associated to the same shared state is currently waiting on a call to `future::get`, it unblocks and returns val.

The member of the void specialization simply makes the shared state ready, without setting any value

第六章 多进程

参考：linux System Coding/ 系统通信一章

第七章 泛型编程

7.1 decltype

decltype(expr) 用来推断 expr 的类型，和 auto 是类似的，相当于类型占位符，占据一个类型的位置；auto f(A a, B b) -> decltype(a+b) 是一种用法，不能写作 decltype(a+b) f(A a, B b)，为啥？！c++ 就是这么规定的！

7.2 完美转发

右值引用类型是独立于值的，一个右值引用参数作为函数的形参，在函数内部再转发该参数的时候它已经变成一个左值，并不是他原来的类型。

如果我们需要一种方法能够按照参数原来的类型转发到另一个函数，这种转发类型称为完美转发

大概意思就是：不改变最初传入的类型的引用类型（左值还是左值，右值还是右值）：

```
// forward example
#include <utility>           // std::forward
#include <iostream>            // std::cout

// function with lvalue and rvalue reference overloads:
void overloaded (const int& x)
{
    std::cout << "[lvalue]";
}
void overloaded (int&& x)
{
    std::cout << "[rvalue]";
}

// function template taking rvalue reference to deduced type:
template <class T> void fn (T&& x)
{
    overloaded (x);           // always an lvalue
    overloaded (std::forward<T>(x)); // rvalue if argument is rvalue
}

int main () {
```

```

int a;

std::cout << "calling fn with lvalue:";
fn(a);
std::cout << '\n';

std::cout << "calling fn with rvalue:";
fn(0);
std::cout << '\n';

return 0;
}

//OutPut
calling fn with lvalue: [lvalue][lvalue]
calling fn with rvalue: [lvalue][rvalue]

```

7.3 前言须知

类模板是 c++ 编译器指令，说明了如何生成类和成员函数除非编译器实现了新的关键字 `export` 关键字，否则将模板成员函数放置在一个独立的实现文件中，将无法运行：因为模板不是函数，他们不能单独编译，模板必须与特定的模板实例化请求一起使用，为此 **最简单的方法是：将所有的模板信息放在一个头文件中，并在要使用这些模板的文件中包含该头文件**，否则会出现以下错误 **LNK2019**：

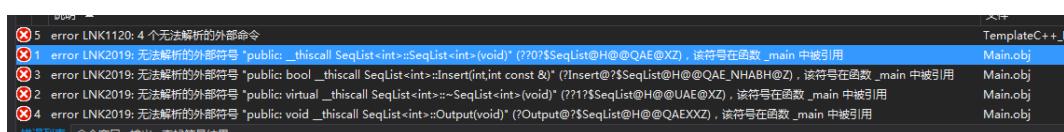


图 7.1: Error: 没包含 .cpp 文件

解决方法：同时包含.cpp 与.h 文件，或将实现写入.h 文件

```

全局范围]
1 #include <iostream>
2 #include "SeqList.h"
3 #include "SeqList.cpp"
4
5 int main()
6 {
7     SeqList<int> A;

```

图 7.2: 解决 LNK2019 问题

特点

- 抽象性：模版代码高度抽象，是函数和类的模范
- 安全性：型式检查能够发现大多数据类型失配问题
- 通用性：函数和类模版定义一次，按需生成函数和类的实体
- 易用性：接口相对直观且高度一致
- 效率：减少冗余代码，提高编程效率，通过编译优化，提升程序执行效率

用途

- 函数模板：构造函数集，实现不依赖特定数据结构的抽象算法
- 类模板：构造类集，实现抽象数据结构
- 元编程：构造在编译期执行的运算，提升程序执行效率

7.4 函数模版

7.4.1 定义及使用

```
template <typename AnyType>
void Swap(AnyType &a, AnyType &b)
{
    AnyType temp;
    temp = a;
    a = b;
    b = temp;
}
```

7.4.2 声明模版函数

```
template<typename T>
void Swap(T &a, T &b);
```

7.4.3 模版也可以重载

7.4.4 实参的演绎-模版类型推导确定

当我们调用某些模版时，模板参数可以由我们所传递的实参来决定。如下

```
template <typename T>
inline T const& max(T const& a, T const& b);

max(4,7);           //Ok: 两个实参的类型都是int
max(4,4.2);        //ERROR: 第一个T为int, 第二个T为double
```

有三种方法可以处理上面的错误：

1. 对实参进行强制类型转换，使他们可以相互匹配：`max(static_cast<double>(4),4.2);`
2. 显示指定 T 的类型：`max<double>(4,4.2);`
3. 指定两个参数可以具有不同的类型

7.4.5 定制非模版函数（重载函数模版）

函数模板功能非常强大，但是有时候可能会陷入困境，加入待比较的函数模板没有提供正确的操作符，或者提供的操作对某些特定的类型会产生不正确的操作结果，则程序不会对此进行编译。为了避免这种错误，可以使用函数模板和同名的非模板函数重载，这就是函数定制，以完成对特定的类型等完成特定的功能。

函数模板与同名的非模板函数重载必须遵守以下规定：

- 寻找一个参数完全匹配的函数，如有，则调用它
- 如果失败，寻找一个函数模板，使其实例化，产生一个匹配的模板函数，若有，则调用它
- 如果失败，再试低一级的对函数重载的方法，例如通过类型转换可产生的参数匹配等，若找到匹配的函数，调用它

```
// 定义函数模板，找出三个值中最小的值，与数据类型无关
template <class T>
T min(T ii , T jj , T kk)
{
    T temp;
    if(( ii<jj )&&(ii<kk )){           temp=ii ;      }
    else if(( jj<ii )&&(jj<kk )){       temp=jj ;      }
    else {           temp=kk ;      }
    return temp;
}
// 非模板函数重载
const char* min(const char* ch1 , const char* ch2 , const char* ch3)
{
    const char* temp;
    int result1 = strcmp(ch1,ch2);
    int result2 = strcmp(ch1,ch3);
    int result3 = strcmp(ch2,ch1);
    int result4 = strcmp(ch2,ch3);
    if(( result1<0)&&(result2<0)) {           temp = ch1;      }
    else if(( result3<0)&&(result4<0)) {       temp=ch2;      }
    else {           temp=ch3;      }
    return temp;
}
void main()
{
    cout<<min(100,20,30)<<endl;
    cout<<min(10.60,10.64,53.21)<<endl;
```

```

        cout<<min( 'c' , 'a' , 'C')<<endl;
        cout<<min( "anderson" , "Washington" , "Smith")<<endl; // 正常输出 Smith
    }

```

7.5 函数

编写函数，求某个数据集的最小元，元素型式为 T

实现策略：使用**函数指针**作为回调函数参数

实现策略：使用**函子**（**function object, functor**）作为回调函数参数

7.5.1 函数指针实现

```

template< typename T >
const T & Min( const T * a, int n, bool (*comparer)(const T&, const T&) )
{
    int index = 0;
    for( int i = 1; i < n; i++ )
    {
        if( comparer( a[ i ] , a[ index ] ) )
            index = i ;
    }
    return a[ index ];
}

```

7.5.2 函子

目的：

- 功能上：类似函数指针
- 实现上：重载函数调用操作符，必要时重载小于比较操作符

优点：

- 函数指针不能内联，而函子可以，效率更高
- 函子可以拥有任意数量的额外数据，可以保存结果和状态，提高代码灵活性
- 编译时可对函子进行型式检查

例子 ->

```

template< typename T > class Comparer
{
public:
    // 确保型式T已存在或重载operator<
    bool operator()( const T & a, const T & b ) { return a < b; }
}

```

```

};

template< typename T, typename Comparer >
const T & Min( const T * a, int n, Comparer comparer )
{
    int index = 0;
    for( int i = 1; i < n; i++)
    {
        if( comparer( a[ i ] , a[ index ] ) ) index = i ;
    }
    return a[ index ];
}

// 使用方法
int a[8] = { 9, 2, 3, 4, 5, 6, 7, 8 };
int min = Min( a, 8, Comparer<int>() ); // 构造匿名函数作为函数参数

```

7.6 类模板

7.6.1 定义类模板

```

//.h file
template <Class Type> //First imoprtant thing

class Stack
{
private:
    enum {MAX = 10}; //constant specific to class
    Type items[MAX];
    int top;
public:
    Stack();
    bool isEmpty();
    bool push( const Type& item );
    bool pop( Type &item );
};

```

7.6.2 使用类模板

```

//.cpp file
template <class Type>
bool Stack<Type>::isEmpty()
{

```

```
    return top==0;  
}
```

7.6.3 类模板别名

```
typedef std::array<double,12> arrd;  
  
arrd gallons;
```

7.6.4 类模板显示特化

使用特定的型或值显式特化类模板，以定制类模板代码

```
template<> class A<char> { ... };
```

- 显式特化版本覆盖体化版本
- 显式特化并不要求与原始模板相同，特化版本可以具有不同的数据成员或成员函数
- 类模板可以部分特化，结果仍是类模板，以支持类模板的部分定制

7.7 模版参数

7.7.1 非类型模版参数

```
template<class T, size_t N> class Stack{};  
  
template<class T, size_t N> class Stack{  
    T data[N]; // Fixed capacity is N  
    size_t count;  
  
public:  
    void push(const T& t);  
};  
  
template<class T>  
inline void Stack<T>::push(const T& t){// Something}
```

7.7.2 默认模版参数

```
template<class T, size_t N = 100> class Stack{};  
  
template<class T = int, size_t N = 100>  
class Stack{  
    T data[N];  
    size_t count;
```

```

public :
    void push( const T& t );
};

Stack<> myStack;      // Same as Stack<int , 100>

```

7.7.3 模版类型的模版参数

```

template<class T, template<class U> class Seq> class Container{};
template<class T, template<class U> class Seq>
class Container{
    Seq<T> seq;
public :
    void append( const T& t ){seq.push_back(t);}
    // Other Stuffs
};

Container<int , vector> container;

```

7.7.4 typename

必须对 模版中的嵌套类型使用 typename 进行说明，否则模版会编译不过

```

template<class T, template<class U, class = allocate<U> > class Seq>
void printSeq( const Seq<T>& seq)
{
    for( typename Seq<T>::iterator b = seq.begin(); b != seq.end(); )
    {
        cout << *b++ << endl;
    }
}

```

7.7.5 模板类中再有模版成员

```

template<typename T> class complex{
    template<class X> complex( const complex<X>& );
};

complex<float> z(1,2);
complex<double> w(z);

int data[5] = {1, 2, 3, 4, 5};
vector<int> v1(data, data+5);
vector<double> v2(v1.begin(), v1.end());

```

7.8 模版特化

特化 将类型抽取出来固定,那么前面就不再具有泛化类型,template<被抽取走> class specialLize<用特化可以理解为模版针对不同参数的重载。

7.8.1 全特化 Full Specialization

template<>

```
// 函数特化
template<class T> const T& min(const T& a, const T& b)
{
    return (a < b) ? a : b;
}

// An explicit specialization of the min template
template<>
const char* const& min<const char*> (const char* const& a, const char*
    const& b)
{
    return (strcmp(a, b) < 0) ? a : b;
}

// 类特化
template<class T, class Allocator = allocator<T>> class vector {};
// 全特化
template<> class vector<bool, allocator<bool>> {};
```

7.8.2 偏特化 Partial Specialization

改变多模版参数的一个为固定值,或同类型的指针类型,然后特化函数

```
// 偏特化 第一个参数特化, 第二参数不特化
template<class Alloc> class vector<bool, Alloc> {};
```

7.9 模版友元

```
// Necessary forward Declarations;
template<class T> class Friend;
template<class T> void f(const Friend<T> &);

template<class T> class Friend
{
    T t;
public:
```

```

    Friendly( const T& theT):t (theT){}
    // f后的<>必不可少，要不就是普通函数而不是模版函数了
    friend void f<>(const Friendly<T>&);
    void g(){ f(*this); }
}

// 特化 友元

template<class T> class Friendly
{
    template<class U> friend void f<>(const Friendly<U>&);
    /*
        由于友元声明的模版参数独立于T，因此任意的T和U的组合都允许使用，形成友元关系

        像成员模版一样，友元模版也可以出现在 非模版类中。
    */
}

```

7.10 元编程

7.11 `shared_ptr` 实现原理

<https://blog.csdn.net/ithiker/article/details/51532484>

7.11.1 源码

```

template<typename _Tp>
class shared_ptr
: public __shared_ptr<_Tp>
{
public:
    shared_ptr()
    : __shared_ptr<_Tp>() { }

    template<typename _Tp1>
    explicit
    shared_ptr(_Tp1* __p)
    : __shared_ptr<_Tp>(__p) { }

    template<typename _Tp1, typename _Deleter>
    shared_ptr(_Tp1* __p, _Deleter __d)
    : __shared_ptr<_Tp>(__p, __d) { }
}

```

```

template<typename _Tp1>
shared_ptr( const shared_ptr<_Tp1>& __r)
: __shared_ptr<_Tp>(__r) { }

template<typename _Tp1>
explicit
shared_ptr( const weak_ptr<_Tp1>& __r)
: __shared_ptr<_Tp>(__r) { }

template<typename _Tp1>
shared_ptr( const shared_ptr<_Tp1>& __r, __static_cast_tag)
: __shared_ptr<_Tp>(__r, __static_cast_tag()) { }

template<typename _Tp1>
shared_ptr( const shared_ptr<_Tp1>& __r, __const_cast_tag)
: __shared_ptr<_Tp>(__r, __const_cast_tag()) { }

template<typename _Tp1>
shared_ptr( const shared_ptr<_Tp1>& __r, __dynamic_cast_tag)
: __shared_ptr<_Tp>(__r, __dynamic_cast_tag()) { }

template<typename _Tp1>
shared_ptr&
operator=( const shared_ptr<_Tp1>& __r) // never throws
{
    this->__shared_ptr<_Tp>::operator=(__r);
    return *this;
}

// 2.2.3.8 shared_ptr specialized algorithms.

template<typename _Tp>
inline void
swap( __shared_ptr<_Tp>& __a, __shared_ptr<_Tp>& __b)
{
    __a.swap(__b);
}

template<typename _Tp, typename _Tp1>
inline shared_ptr<_Tp>
static_pointer_cast( const shared_ptr<_Tp1>& __r)
{
    return shared_ptr<_Tp>(__r, __static_cast_tag());
}

template<typename _Tp, typename _Tp1>
inline shared_ptr<_Tp>

```

```

const_pointer_cast(const shared_ptr<_Tp1>& __r)
{
    return shared_ptr<_Tp>(__r, __const_cast_tag());
}

template<typename _Tp, typename _Tp1>
inline shared_ptr<_Tp>
dynamic_pointer_cast(const shared_ptr<_Tp1>& __r)
{
    return shared_ptr<_Tp>(__r, __dynamic_cast_tag());
}

```

7.11.2 分析

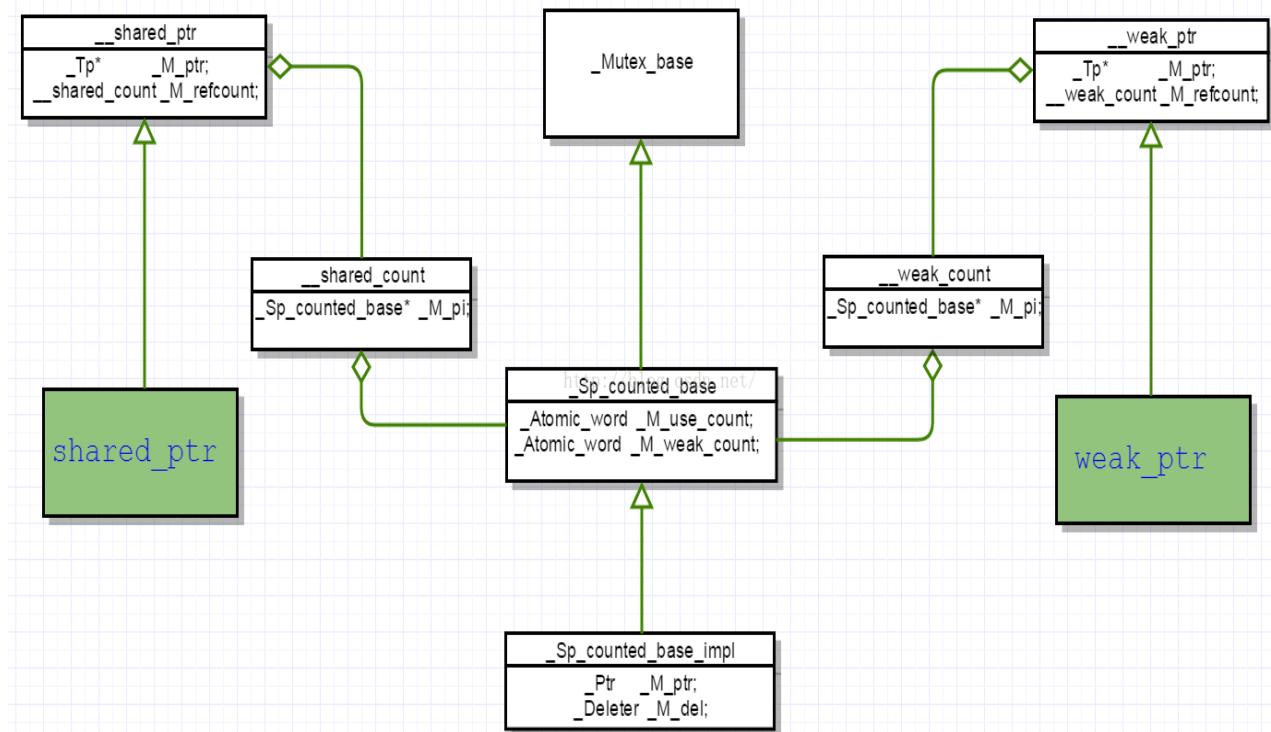


图 7.3: weakptr And sharedptr

从上面的类图可以清楚的看出`shared_ptr` 内部含有一个指向被管理对象 (**managed object**)T 的指针以及一个`__shared_count` 对象，`__shared_count` 对象包含一个指向管理对象 (**manager object**) 的基类指针，管理对象 (**manager object**) 由具有原子属性的use_count 和weak_count、指向被管理对象 (**managed object**)T 的指针、以及用来销毁被管理对象的 deleter 组成，以下均将用 new 创建后托管给`shared_ptr` 等智能指针管理的对象叫做被管理对象 (**managed object**)；`shared_ptr` 等智能指针内部创建的用来维护被管理对象生命周期的实例叫做管理对象 (**manager object**)：

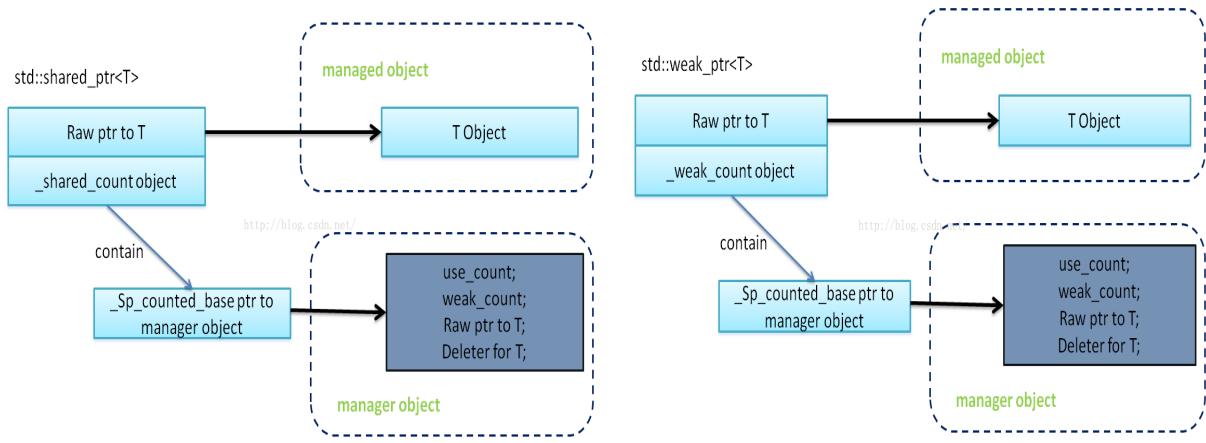


图 7.4: weakptr And sharedptr

从上图对比可以看出,shared_ptr 与weak_ptr 的差异主要是由__shared_ptr 与__weak_ptr 体现出来的,而__shared_ptr 与__weak_ptr 的差异则主要是由__shared_count 与__weak_count 体现出来。

shared_ptr 内部包含一个指向被管理对象的指针_M_ptr, _Sp_counted_base_impl 内部也含有一个指向被管理对象的指针_M_ptr, 它们是不是重复多余了呢?

实际上不多余,它们有各自的功能。这首先要从shared_ptr 的拷贝构造或者赋值构造说起,当一个shared_ptr 对象 sp2 是由 sp1 拷贝构造或者赋值构造得来的时候,实际上构造完成后 sp1 内部的__shared_count 对象包含的指向管理对象的指针与 sp2 内部的__shared_count 对象包含的指向管理对象的指针是相等的,也就是说当多个shared_ptr 对象来管理同一个对象时,它们共同使用同一个动态分配的管理对象。

这可以从下面的__share_ptr 的构造函数和__shared_count 的构造函数清楚的看出。

```
template<typename _Tp1>
__shared_ptr( const __shared_ptr<_Tp1, _Lp>& __r)
: _M_ptr(__r._M_ptr), _M_refcount(__r._M_refcount) // never throws
{ __glibcxx_function_requires(_ConvertibleConcept<_Tp1*, _Tp*>) }

__shared_count&
operator=(const __shared_count& __r) // noexcept
{
    _Sp_counted_base<_Lp>* __tmp = __r._M_pi;
    if ( __tmp != _M_pi)
    {
        if ( __tmp != 0)
            __tmp->_M_add_ref_copy();
        if ( _M_pi != 0)
            _M_pi->_M_release();

        _M_pi = __tmp;
    }
}
```

从第一张类图可以看出，`__share_count` 是计数的管理对象，其中的`_M_pi` 表示实际计数对象的指针。

上面说说当多个`shared_ptr` 对象来管理同一个对象时，它们共同使用同一个动态分配的管理对象，为什么上面给出的`__shared_count` 的构造函数中出现了`__tmp != _M_pi` 的情形呢？这在`sp2` 未初始化时（`_M_pi` 为 0，`_r._M_pi` 非 0）便是这样的情形。

更一般的，也可以考虑这样的情形：`shared_ptr` 实例`sp1` 开始指向类 A 的实例对象 **a1**，另外一个`shared_ptr` 实例`sp2` 指向类 A 的实例对象 **a2** (`a1 != a2`)，当把`sp2` 赋值给`sp1` 时便会出现上面的情形。假设初始时有且仅有一个`sp1` 指向`a1`，有且仅有一个`sp2` 指向`a2`；则赋值结束时`sp1` 与`sp2` 均指向`a2`，没有指针指向`a1`，`sp1` 指向的`a1` 以及其对应的管理对象均应该被析构。这在上面的代码中我们可以很清楚的看到：因为`__tmp != _M_pi`，`__tmp->_M_add_ref_copy()` 将会增加`a2` 的`use_count` 的引用计数；由于`a1` 内部的`_M_pi != 0`，将会调用其`_M_release()` 函数

```
//*****_Sp_counted_base*****//
void
_M_add_ref_copy()
{
    __gnu_cxx::__atomic_add_dispatch(&_M_use_count, 1);
}

//*****_Sp_counted_base*****//
void
_M_release() // noexcept
{
    _GLIBCXX_SYNCHRONIZATION_HAPPENS_BEFORE(&_M_use_count);
    if (__gnu_cxx::__exchange_and_add_dispatch(&_M_use_count, -1) == 1)
    {
        _GLIBCXX_SYNCHRONIZATION_HAPPENS_AFTER(&_M_use_count);
        _M_dispose();

        if (_Mutex_base<_Lp>::__S_need_barriers)
        {
            __atomic_thread_fence (__ATOMIC_ACQ_REL);
        }

        _GLIBCXX_SYNCHRONIZATION_HAPPENS_BEFORE(&_M_weak_count);
        if (__gnu_cxx::__exchange_and_add_dispatch(&_M_weak_count, -1) == 1)
        {
            _GLIBCXX_SYNCHRONIZATION_HAPPENS_AFTER(&_M_weak_count);
            _M_destroy();
        }
    }
}

//*****_Sp_counted_base*****//
// Called when _M_use_count drops to zero, to release the resources
// managed by *this.
```

```

virtual void _M_dispose() = 0; // noexcept

// Called when _M_weak_count drops to zero.
virtual void _M_destroy() // noexcept
{
    delete this;
}

//*****_Sp_counted_base_impl*****
virtual void _M_dispose() // noexcept
{
    _M_del(_M_ptr);
}

```

`_M_release()` 函数首先对 `a1` 的 `use_count` 减去 1，并对比减操作之前的值，如果减之前是 1，说明减后是 0，`a1` 没有任何 `shared_ptr` 指针指向它了，应该将 `a1` 对象销毁，于是调用 `_M_dispose()` 函数销毁 `a1`；同时对 `a1` 的 `weak_count` 减去 1，也对比减操作之前的值，如果减之前是 1，说明减后是 0，`a1` 没有 `weak_ptr` 指向它了，应该将管理对象销毁，于是调用 `_M_destroy()` 销毁了管理对象。这就可以解答为什么图 2 所示中 `shared_ptr` 内部含有两个指向被管理对象的指针了：
`shared_ptr` 直接包含的裸指针是为了实现一般指针的 `->`, `*` 等操作，通过 `shared_count` 间接包含的指针是为了管理对象的生命周期，回收相关资源。

换句话说，`shared_count` 内部的 `use_count` 主要用来标记被管理对象的生命周期，`weak_count` 主要用来标记管理对象的生命周期。

当一个 `shared_ptr` 超出作用域被销毁时，它会调用其 `shared_count` 的 `_M_release()` 对 `use_count` 和 `weak_count` 进行自减并判断是否需要释放管理对象和被管理对象，这是 **RAII** 原理的核心体现：

```

~__shared_count() // noexcept
{
    if (_M_pi != 0)
        _M_pi->_M_release();
}

```

7.12 weak_ptr 实现原理

https://blog.csdn.net/dong_beijing/article/details/79504591

7.12.1 使用示例

- `expired()` 用于检测所管理的对象是否已经释放，如果已经释放，返回 `true`；否则返回 `false`。
- `lock()` 用于获取所管理的对象的强引用 (`shared_ptr`)。如果 `expired` 为 `true`，返回一个空的 `shared_ptr`；否则返回一个 `shared_ptr`，其内部对象指向与 `weak_ptr` 相同。
- `use_count()` 返回与 `shared_ptr` 共享的对象的引用计数。

- reset() 将 weak_ptr 置空.
- weak_ptr 支持拷贝或赋值, 但不会影响对应的 shared_ptr 内部对象的计数.

```

std :: shared_ptr<int> sp( new int(10));
std :: weak_ptr<int> wp(sp);
wp = sp;
printf("%d\n", wp.use_count()); // 1
wp.reset();
printf("%d\n", wp); // 0

// 检查 weak_ptr 内部对象的合法性.
if ( std :: shared_ptr<int> sp = wp.lock())
{
}

```

7.12.2 实现原理

```

template<typename _Tp, _Lock_policy _Lp>
class __weak_ptr
{
public:
    typedef _Tp element_type;

    __weak_ptr()
    : _M_ptr(0), _M_refcount() // never throws
    {}

    // Generated copy constructor, assignment, destructor are fine.
    template<typename _Tp1>
    __weak_ptr( const __weak_ptr<_Tp1, _Lp>& __r)
    : _M_refcount(__r._M_refcount) // never throws
    {
        __glibcxx_function_requires(_ConvertibleConcept<_Tp1*, _Tp*>)
        _M_ptr = __r.lock().get();
    }

    template<typename _Tp1>
    __weak_ptr( const __shared_ptr<_Tp1, _Lp>& __r)
    : _M_ptr(__r._M_ptr), _M_refcount(__r._M_refcount) // never throws
    {
        __glibcxx_function_requires(_ConvertibleConcept<_Tp1*, _Tp*>)
    }

    template<typename _Tp1>
    __weak_ptr&
    operator=(const __weak_ptr<_Tp1, _Lp>& __r) // never throws

```

```

{
    _M_ptr = __r.lock().get();
    _M_refcount = __r._M_refcount;
    return *this;
}

template<typename _Tp1>
__weak_ptr&
operator=(const __shared_ptr<_Tp1, _Lp>& __r) // never throws
{
    _M_ptr = __r._M_ptr;
    _M_refcount = __r._M_refcount;
    return *this;
}

__shared_ptr<_Tp, _Lp>
lock() const // never throws
{
    #ifdef __GTHREADS
        // Optimization: avoid throw overhead.
        if (expired())
            return __shared_ptr<element_type, _Lp>();

        __try
    {
        return __shared_ptr<element_type, _Lp>(*this);
    }
    __catch( const bad_weak_ptr&)
    {
        // Q: How can we get here?
        // A: Another thread may have invalidated r after the
        //     use_count test above.
        return __shared_ptr<element_type, _Lp>();
    }
}

#else
    // Optimization: avoid try/catch overhead when single threaded.
    return expired() ? __shared_ptr<element_type, _Lp>()
                      : __shared_ptr<element_type, _Lp>(*this);
#endif
} // XXX MT

long
use_count() const // never throws
{
    return _M_refcount._M_get_use_count(); //shared Count
}

```

```

}

bool
expired() const // never throws
{
    return __M_refcount.__M_get_use_count() == 0;      //shared Count
}

void
reset() // never throws
{
    __weak_ptr().swap(*this);
}

void
swap(__weak_ptr& __s) // never throws
{
    std::swap(__M_ptr, __s.__M_ptr);
    __M_refcount.__M_swap(__s.__M_refcount);
}

private:
// Used by __enable_shared_from_this.
void
__M_assign(_Tp* __ptr, const __shared_count<_Lp>& __refcount)
{
    __M_ptr = __ptr;
    __M_refcount = __refcount;
}

template<typename _Tp1>
bool
__M_less(const __weak_ptr<_Tp1, _Lp>& __rhs) const
{
    return __M_refcount < __rhs.__M_refcount;
}

template<typename _Tp1, _Lock_policy _Lp1> friend class __shared_ptr;
template<typename _Tp1, _Lock_policy _Lp1> friend class __weak_ptr;
friend class __enable_shared_from_this<_Tp, _Lp>;
friend class enable_shared_from_this<_Tp>;

// Friend injected into namespace and found by ADL.
template<typename _Tp1>
friend inline bool
operator<(const __weak_ptr& __lhs, const __weak_ptr<_Tp1, _Lp>& __rhs)
{

```

```

        return __lhs.__M_less(__rhs);
    }

    _Tp*           __M_ptr;           // Contained pointer.
    __weak_count<_Tp> __M_refcount; // Reference counter.
};


```

对于weak_ptr, 其对应的__weak_count 的拷贝构造函数如下

```

//*****_Sp_Counted_Base*****
void
__M_weak_add_ref() // noexcept
{ __gnu_cxx::__atomic_add_dispatch(&__M_weak_count, 1); }

//*****_Sp_Counted_Base*****
void
__M_weak_release() // noexcept
{
    // Be race-detector-friendly. For more info see bits/c++config.
    _GLIBCXX_SYNCHRONIZATION_HAPPENS_BEFORE(&__M_weak_count);
    if (__gnu_cxx::__exchange_and_add_dispatch(&__M_weak_count, -1) == 1)
    {
        _GLIBCXX_SYNCHRONIZATION_HAPPENS_AFTER(&__M_weak_count);
        if (_Mutex_base<_Tp>::__S_need_barriers)
        {
            // See __M_release(),
            // destroy() must observe results of dispose()
            __atomic_thread_fence(__ATOMIC_ACQ_REL);
        }
        __M_destroy();
    }
}

__weak_count<_Tp>&
operator=(const __shared_count<_Tp>& __r) // noexcept
{
    __Sp_Counted_Base<_Tp>* __tmp = __r.__M_pi;
    if (__tmp != 0)
        __tmp->__M_weak_add_ref();

    if (__M_pi != 0)
        __M_pi->__M_weak_release();

    __M_pi = __tmp;

    return *this;
}

```

```

__weak_count<_Lp>&
operator=(const __weak_count<_Lp>& __r) // noexcept
{
    _Sp_counted_base<_Lp>* __tmp = __r._M_pi;
    if (__tmp != 0)
        __tmp->_M_weak_add_ref();
    if (_M_pi != 0)
        _M_pi->_M_weak_release();
    _M_pi = __tmp;

    return *this;
}

__weak_count<_Lp>&
operator=(const __shared_count<_Lp>& __r) // noexcept
{
    _Sp_counted_base<_Lp>* __tmp = __r._M_pi;
    if (__tmp != 0)
        __tmp->_M_weak_add_ref();
    if (_M_pi != 0)
        _M_pi->_M_weak_release();
    _M_pi = __tmp;
    return *this;
}

~__weak_count() // noexcept
{
    if (_M_pi != 0)
        _M_pi->_M_weak_release();
}

```

从上面可以看出：

- `__weak_count` 相关的赋值拷贝以及析构函数均只会影响到`weak_count` 的值, 对`use_count` 没有影响；当`weak_count` 为 0 时，释放管理对象。也就是说`__weak_ptr` 不影响被管理对象的生命周期。同时由于`__weak_ptr` 没有像`__shared_ptr` 那样实现*, `->` 等常见指针相关操作符, `__weak_ptr` 不能直接操作被管理对象；
- `__weak_count` 自身间的赋值以及`__shared_count` 对`__weak_count` 的赋值时，它们都具有同样的指向管理对象的指针；也就是说当多个`__weak_ptr` 和`__shared_ptr` 指向同一个被管理对象时，它们共享同一个管理对象，这就保证了可以通过`__weak_ptr` 可以判断`__shared_ptr` 指向的被管理对象是否存在以及获取到被管理对象的指针。

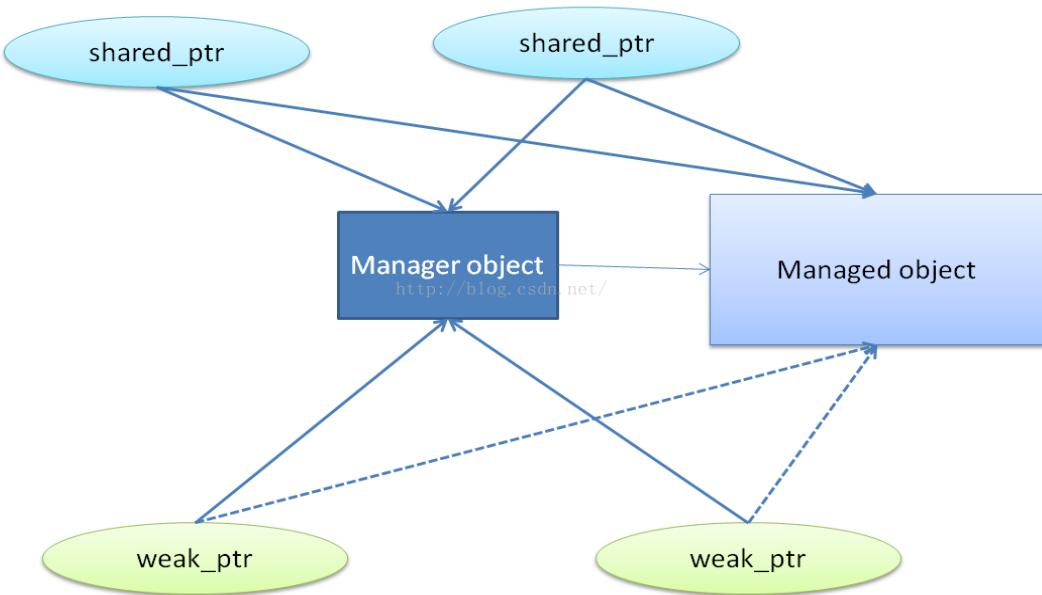


图 7.5: sharedPtr 与 weakPtr 管理同一对象时

由于weak_ptr 不能直接操作被管理对象但其仍然持有指向被管理对象的指针（用来初始化内部的__weak_count 对象），weak_ptr 与被管理对象用虚线联接。

_weak_ptr 有几个重要的成员函数：通过expired() 方法来判断对象是否过期（已经被释放）；通过use_count() 方法返回目前有多少个__shared_ptr 对象指向被管理对象；通过lock() 方法返回一个指向被管理对象的__shared_ptr 指针，调用者可以通过这个__shared_ptr 指针来操纵被管理对象而不用担心资源泄漏；

```

/***************** __weak_ptr *****/
long
use_count() const // never throws
{
    return __M_refcount.__M_get_use_count();
}

bool
expired() const // never throws
{
    return __M_refcount.__M_get_use_count() == 0;
}

__shared_ptr<_Tp, _Lp>
lock() const // never throws
{
#ifdef __GTHR_THREADS
    // Optimization: avoid throw overhead.
    if (expired())
        return __shared_ptr<element_type, _Lp>();
}

```

```

    __try
    {
        return __shared_ptr<element_type, _Lp>(*this);
    }
    __catch( const bad_weak_ptr& )
    {
        // Q: How can we get here?
        // A: Another thread may have invalidated r after the
        //     use_count test above.
        return __shared_ptr<element_type, _Lp>();
    }

#else
    // Optimization: avoid try/catch overhead when single threaded.
    return expired() ? __shared_ptr<element_type, _Lp>()
                      : __shared_ptr<element_type, _Lp>(*this);
#endif
} // XXX MT

```

7.13 enable_shared_from_this

<https://blog.csdn.net/caoshangpa/article/details/79392878>

<http://www.cnblogs.com/yang-wen/p/8573269.html>

当然shared_ptr也不是万能的,使用的时候也要注意到它给程序员挖的一个大坑:shared_ptr能够管理对象的生命周期,负责对象资源释放,其前提条件是所有shared_ptr共用同一个管理对象。如果多个shared_ptr使用多个管理对象来管理同一个被管理对象,这些管理对象在use_count为0时均会释放被管理对象,将会造成多个管理对象多次释放被管理对象,造成twice delete的堆错误。

7.13.1 使用场合

当类A被share_ptr管理,且在类A的成员函数里需要把当前类对象作为参数传给其他函数时,就需要传递一个指向自身的share_ptr。

为何不直接传递 this 指针 使用智能指针的初衷就是为了方便资源管理,如果在某些地方使用智能指针,某些地方使用原始指针,很容易破坏智能指针的语义,从而产生各种错误。

可以直接传递 share_ptr<this> 么? 答案是不能,因为这样会造成2个非共享的share_ptr指向同一个对象,未增加引用计数导致对象被析构两次。

在getptr()函数构造智能指针时,我们无法确定这个对象是不是被shared_ptr管理着,因此这样构造的shared_ptr并不是与其他shared_ptr共享一个计数器,那么,在析构时就会导致对象

被重复释放, 从而引发错误.

```
#include <memory>
#include <iostream>

class Bad
{
public:
    std::shared_ptr<Bad> getptr() {
        return std::shared_ptr<Bad>(this);
    }
    ~Bad() { std::cout << "Bad::~Bad() called" << std::endl; }
};

int main()
{
    // 错误的示例, 每个shared_ptr都认为自己是对象仅有的所有者
    std::shared_ptr<Bad> bp1(new Bad());
    std::shared_ptr<Bad> bp2 = bp1->getptr();
    // 打印bp1和bp2的引用计数
    std::cout << "bp1.use_count() = " << bp1.use_count() << std::endl;
    std::cout << "bp2.use_count() = " << bp2.use_count() << std::endl;
} // Bad 对象将会被删除两次
```

当然, 一个对象被删除两次会导致崩溃。

7.13.2 使用场景分析

现在明确一下我们的需求: 在一个对象内部构造该对象的 shared_ptr 时, 即使该对象已经被 shared_ptr 管理着, 也不会造成对象被两个独立的智能指针管理.

这就要求我们在对象内构造对象的智能指针时, 必须能识别有对象是否已经由其他智能指针管理, 智能指针的数量, 并且我们创建智能指针后也能让之前的智能指针感知到.

那么上述问题如何解决呢, 就是通过 enable_shared_from_this.

enable_shared_from_this 是一个模板类, 定义于头文件<memory>, 其原型为:

```
template< class T > class enable_shared_from_this;
```

```
#include <memory>
#include <iostream>

struct Good : std::enable_shared_from_this<Good> // 注意: 继承
{
public:
    std::shared_ptr<Good> getptr() {
        return shared_from_this();
    }
    ~Good() { std::cout << "Good::~Good() called" << std::endl; }
};
```

```

int main()
{
    // 大括号用于限制作用域，这样智能指针就能在system("pause")之前析构
    {
        std::shared_ptr<Good> gp1(new Good());
        std::shared_ptr<Good> gp2 = gp1->getptr();
        // 打印gp1和gp2的引用计数
        std::cout << "gp1.use_count() = " << gp1.use_count() << std::endl;
        std::cout << "gp2.use_count() = " << gp2.use_count() << std::endl;
    }
    system("pause");
}

```

为何会出现这种使用场合？

因为在异步调用中，存在一个保活机制，异步函数执行的时间点我们是无法确定的，然而异步函数可能会使用到异步调用之前就存在的变量。为了保证该变量在异步函数执期间一直有效，我们可以传递一个指向自身的 `share_ptr` 给异步函数，这样在异步函数执行期间 `share_ptr` 所管理的对象就不会析构，所使用的变量也会一直有效了（保活）。

7.13.3 实现原理分析

当一个对象M 创建后，如果一个函数`getThisPtr()`（另一个类的成员函数或是其它自由函数）的形参为 M 类型的智能指针，如何在对象 M 内部将对象 M 的指针作为实参传递给该函数 f？C++ 引入了`enable_shared_from_this` 利用`weak_ptr` 的特性解决了这一问题。

其基本思想是通过M 继承模板类`enable_shared_from_this`，这样对象 M 内部将会有一个`_weak_ptr` 指针`_M_weak_this`，在第一次创建指向M 的`shared_ptr` Ptr 时，通过模板特化，将会初始化`_M_weak_this`，这样M 内部也会产生一个指向自身的`weak_ptr`，并且该`weak_ptr` 内部的管理对象与Ptr 的管理对象是相同的。

（这可以从`weak_ptr` 内部的`_M_assign` 函数看出）

```

// Friend of enable_shared_from_this.
template<typename _Tp1, typename _Tp2>
void __enable_shared_from_this_helper(_shared_count<>&, const enable_shared_from_this<_Tp1>*, const _Tp2*) ;

template<typename _Tp1>
explicit __shared_ptr(_Tp1* __p)
: _M_ptr(__p), _M_refcount(__p)
{
    __glibcxx_function_requires(_ConvertibleConcept<_Tp1*, _Tp*>) typedef int
    _IsComplete[ sizeof(_Tp1) ];
    __enable_shared_from_this_helper(_M_refcount, __p, __p);
}

```

```

}

template<typename _Tp>
class enable_shared_from_this
{
protected:
    enable_shared_from_this() { }

    enable_shared_from_this(const enable_shared_from_this&) { }

    enable_shared_from_this&
    operator=(const enable_shared_from_this&)
    {
        return *this;
    }

    ~enable_shared_from_this() { }

public:
    shared_ptr<_Tp>
    shared_from_this()
    {
        return shared_ptr<_Tp>(&M_weak_this);
    }

    shared_ptr<const _Tp>
    shared_from_this() const
    {
        return shared_ptr<const _Tp>(&M_weak_this);
    }

private:
    template<typename _Tp1>
    void
    _M_weak_assign(_Tp1* __p, const __shared_count<>& __n) const
    {
        M_weak_this._M_assign(__p, __n);
    }

    template<typename _Tp1>
    friend void
    __enable_shared_from_this_helper(const __shared_count<>& __pn, const
        enable_shared_from_this* __pe, const _Tp1* __px)
    {
        if (__pe != 0)
            __pe->_M_weak_assign(const_cast<_Tp1*>(__px), __pn);
    }
}

```

```

    mutable weak_ptr<_Tp> _M_weak_this;
};

_M_assign(_Tp* __ptr, const __shared_count<_Lp>& __refcount)
{
    _M_ptr = __ptr;
    _M_refcount = __refcount;
}

```

7.14 sharedCount、weakCount

7.14.1 count base

```

template<_Lock_policy _Lp>
class _Mutex_base
{
protected:
// The atomic policy uses fully-fenced builtins, single doesn't care.
enum { _S_need_barriers = 0 };
};

template<>
class _Mutex_base<_S_mutex>
: public __gnu_cxx::__mutex
{
protected:
// This policy is used when atomic builtins are not available.
// The replacement atomic operations might not have the necessary
// memory barriers.
enum { _S_need_barriers = 1 };
};

template<_Lock_policy _Lp = __default_lock_policy>
class _Sp_counted_base
: public _Mutex_base<_Lp>
{
public:
_Sp_counted_base()
: _M_use_count(1), _M_weak_count(1) { }

virtual
~_Sp_counted_base() // noexcept
{ }

// Called when _M_use_count drops to zero, to release the resources

```

```

// managed by *this .
virtual void
__M_dispose() = 0; // noexcept

// Called when __M_weak_count drops to zero .
virtual void
__M_destroy() // noexcept
{ delete this; }

virtual void*
__M_get_deleter(const std::type_info&) = 0;

void
__M_add_ref_copy()
{
    __gnu_cxx::__atomic_dispatch(&__M_use_count, 1);
}

void
__M_add_ref_lock();

void
__M_release() // noexcept
{
    // Be race-detector-friendly. For more info see bits/c++config .
    __GLIBCXX_SYNCHRONIZATION_HAPPENS_BEFORE(&__M_use_count);
    if (__gnu_cxx::__exchange_and_add_dispatch(&__M_use_count, -1) == 1)
    {
        __GLIBCXX_SYNCHRONIZATION_HAPPENS_AFTER(&__M_use_count);
        __M_dispose();
        // There must be a memory barrier between dispose() and destroy()
        // to ensure that the effects of dispose() are observed in the
        // thread that runs destroy().
        // See http://gcc.gnu.org/ml/libstdc++/2005-11/msg00136.html
        if (_Mutex_base<_Lp>::_S_need_barriers)
        {
            __atomic_thread_fence (__ATOMIC_ACQ_REL);
        }
    }
    // Be race-detector-friendly. For more info see bits/c++config .
    __GLIBCXX_SYNCHRONIZATION_HAPPENS_BEFORE(&__M_weak_count);
    if (__gnu_cxx::__exchange_and_add_dispatch(&__M_weak_count, -1) ==
        1)
    {
        __GLIBCXX_SYNCHRONIZATION_HAPPENS_AFTER(&__M_weak_count);
        __M_destroy();
    }
}

```

```

        }

    }

    void
    _M_weak_add_ref() // noexcept
{
    __gnu_cxx::__atomic_add_dispatch(&_M_weak_count, 1);
}

void
_M_weak_release() // noexcept
{
    // Be race-detector-friendly. For more info see bits/c++config.h
    __GLIBCXX_SYNCHRONIZATION_HAPPENS_BEFORE(&_M_weak_count);
    if (__gnu_cxx::__exchange_and_add_dispatch(&_M_weak_count, -1) == 1)
    {
        __GLIBCXX_SYNCHRONIZATION_HAPPENS_AFTER(&_M_weak_count);
        if (_Mutex_base<_Lp>::_S_need_barriers)
        {
            // See _M_release(),
            // destroy() must observe results of dispose()
            __atomic_thread_fence(__ATOMIC_ACQ_REL);
        }
        _M_destroy();
    }
}

long
_M_get_use_count() const // noexcept
{
    // No memory barrier is used here so there is no synchronization
    // with other threads.
    return const_cast<const volatile __Atomic_word&>(_M_use_count);
}

private:
    _Sp_counted_base(_Sp_counted_base const&);
    _Sp_counted_base& operator=(_Sp_counted_base const&);

    __Atomic_word    _M_use_count;      // #shared
    __Atomic_word    _M_weak_count;     // #weak + (#shared != 0)
};

template<typename S>
inline void
_Sp_counted_base<S>::_M_add_ref_lock()
{
}

```

```

{
    if ( __gnu_cxx::__exchange_and_add_dispatch(&_M_use_count, 1) == 0)
    {
        _M_use_count = 0;
        __throw_bad_weak_ptr();
    }
}

template<typename _P, typename _D, typename _L>
inline void
_Sp_counted_base<_S_mutex>::
_M_add_ref_lock()
{
    __gnu_cxx::__scoped_lock_sentry(*this);
    if ( __gnu_cxx::__exchange_and_add_dispatch(&_M_use_count, 1) == 0)
    {
        _M_use_count = 0;
        __throw_bad_weak_ptr();
    }
}

template<typename _P, typename _D, typename _L>
inline void
_Sp_counted_base<_S_atomic>::
_M_add_ref_lock()
{
    // Perform lock-free add-if-not-zero operation.
    _Atomic_word __count = _M_use_count;
    do
    {
        if (__count == 0)
            __throw_bad_weak_ptr();
        // Replace the current counter value with the old value + 1, as
        // long as it's not changed meanwhile.
    }
    while (!__atomic_compare_exchange_n(&_M_use_count, &__count, __count + 1,
        true, __ATOMIC_ACQ_REL, __ATOMIC_RELAXED));
}

template<typename _P, typename _D, typename _L>
class _Sp_counted_base_impl
: public _Sp_counted_base<_L>
{
public:
    // Precondition: __d(__p) must not throw.
    _Sp_counted_base_impl(_P __p, _D __d)

```

```

        : _M_ptr(__p) , _M_del(__d) { }

    virtual void
    _M_Dispose() // noexcept
    {
        _M_del(_M_ptr);
    }

    virtual void*
    _M_get_deleter(const std::type_info& __ti)
    {
#if __cpp_rtti
        return __ti == typeid(_Deleter) ? &_M_del : 0;
#else
        return 0;
#endif
    }

private:
    _Sp_counted_base_impl(const _Sp_counted_base_impl&);

    _Sp_counted_base_impl& operator=(const _Sp_counted_base_impl&);

    _Ptr      _M_ptr; // copy constructor must not throw
    _Deleter  _M_del; // copy constructor must not throw
};

template<typename _Tp>
struct _Sp_deleter
{
    typedef void result_type;
    typedef _Tp* argument_type;
    void operator()(_Tp* __p) const { delete __p; }
};

```

7.14.2 shared count

```

template<_Lock_policy _Lp = __default_lock_policy>
class __shared_count
{
public:
    __shared_count()
    : _M_pi(0) // noexcept
    { }

```

```

template<typename _Ptr>
__shared_count(_Ptr __p) : _M_pi(0)
{
    __try
    {
        typedef typename std::tr1::remove_pointer<_Ptr>::type _Tp;
        _M_pi = new _Sp_counted_base_impl<_Ptr, _Sp_deleter<_Tp>, _Lp>(
            __p, _Sp_deleter<_Tp>());
    }
    __catch (...)
    {
        delete __p;
        __throw_exception_again;
    }
}

template<typename _Ptr, typename _Deleter>
__shared_count(_Ptr __p, _Deleter __d) : _M_pi(0)
{
    __try
    {
        _M_pi = new _Sp_counted_base_impl<_Ptr, _Deleter, _Lp>(__p, __d);
    }
    __catch (...)
    {
        __d(__p); // Call _Deleter on __p.
        __throw_exception_again;
    }
}

// Special case for auto_ptr<_Tp> to provide the strong guarantee.
template<typename _Tp>
explicit
__shared_count(std::auto_ptr<_Tp>& __r)
: _M_pi(new _Sp_counted_base_impl<_Tp*>,
        _Sp_deleter<_Tp>, _Lp >(__r.get(), _Sp_deleter<_Tp>()))
{
    __r.release();
}

// Throw bad_weak_ptr when __r._M_get_use_count() == 0.
template<typename _Lp>
explicit
__shared_count(const __weak_count<_Lp>& __r);

~__shared_count() // nothrow
{
    if (_M_pi != 0)

```

```

        _M_pi->_M_release() ;
    }

__shared_count( const __shared_count& __r)
: _M_pi(__r._M_pi) // nothrow
{
    if (_M_pi != 0)
        _M_pi->_M_add_ref_copy() ;
}

__shared_count&
operator=(const __shared_count& __r) // nothrow
{
    _Sp_counted_base<_Lp>* __tmp = __r._M_pi;
    if (__tmp != _M_pi)
    {
        if (__tmp != 0)
            __tmp->_M_add_ref_copy() ;
        if (_M_pi != 0)
            _M_pi->_M_release() ;
        _M_pi = __tmp;
    }
    return *this;
}

void
_M_swap(__shared_count& __r) // nothrow
{
    _Sp_counted_base<_Lp>* __tmp = __r._M_pi;
    __r._M_pi = _M_pi;
    _M_pi = __tmp;
}

long
_M_get_use_count() const // nothrow
{
    return _M_pi != 0 ? _M_pi->_M_get_use_count() : 0;
}

bool
_M_unique() const // nothrow
{
    return this->_M_get_use_count() == 1;
}

friend inline bool
operator==(const __shared_count& __a, const __shared_count& __b)

```

```

{
    return __a._M_pi == __b._M_pi;
}

friend inline bool
operator<(const __shared_count& __a, const __shared_count& __b)
{
    return std::less<_Sp_counted_base<_Lp>*>()(__a._M_pi, __b._M_pi);
}

void*
__M_get_deleter(const std::type_info& __ti) const
{
    return __M_pi ? __M_pi->__M_get_deleter(__ti) : 0;
}

private:
friend class __weak_count<_Lp>;

__Sp_counted_base<_Lp>* __M_pi;
};

```

7.14.3 weak Count

```

template<_Lock_policy _Lp>
class __weak_count
{
    public:
    __weak_count()
    : __M_pi(0) // nothrow
    { }

    __weak_count(const __shared_count<_Lp>& __r)
    : __M_pi(__r.__M_pi) // nothrow
    {
        if (__M_pi != 0)
            __M_pi->__M_weak_add_ref();
    }

    __weak_count(const __weak_count<_Lp>& __r)
    : __M_pi(__r.__M_pi) // nothrow
    {
        if (__M_pi != 0)
            __M_pi->__M_weak_add_ref();
    }
}

```

```

~__weak_count() // nothrow
{
    if (_M_pi != 0)
        _M_pi->_M_weak_release();
}

__weak_count<_Lp>&
operator=(const __shared_count<_Lp>& __r) // nothrow
{
    _Sp_counted_base<_Lp>* __tmp = __r._M_pi;
    if (__tmp != 0)
        __tmp->_M_weak_add_ref();
    if (_M_pi != 0)
        _M_pi->_M_weak_release();
    _M_pi = __tmp;
    return *this;
}

__weak_count<_Lp>&
operator=(const __weak_count<_Lp>& __r) // nothrow
{
    _Sp_counted_base<_Lp>* __tmp = __r._M_pi;
    if (__tmp != 0)
        __tmp->_M_weak_add_ref();
    if (_M_pi != 0)
        _M_pi->_M_weak_release();
    _M_pi = __tmp;
    return *this;
}

void
_M_swap(__weak_count<_Lp>& __r) // nothrow
{
    _Sp_counted_base<_Lp>* __tmp = __r._M_pi;
    __r._M_pi = _M_pi;
    _M_pi = __tmp;
}

long
_M_get_use_count() const // nothrow
{
    return _M_pi != 0 ? _M_pi->_M_get_use_count() : 0;
}

friend inline bool
operator==(const __weak_count<_Lp>& __a, const __weak_count<_Lp>& __b)
{

```

```

        return __a._M_pi == __b._M_pi;
    }

    friend inline bool
    operator<(const __weak_count<_Lp>& __a, const __weak_count<_Lp>& __b)
    {
        return std :: less<_Sp_counted_base<_Lp>*>()(__a._M_pi, __b._M_pi);
    }

private:
friend class __shared_count<_Lp>;
_Sp_counted_base<_Lp>* _M_pi;
};

template<_Lock_policy _Lp>
inline
__shared_count<_Lp>::
__shared_count( const __weak_count<_Lp>& __r)
: _M_pi(__r._M_pi)
{
    if (_M_pi != 0)
        _M_pi->_M_add_ref_lock();
    else
        __throw_bad_weak_ptr();
}

```

7.15 SFINAE and enable_if

https://eli.thegreenplace.net/2014/sfinae-and-enable_if/

```

void foo(unsigned i) {
    std :: cout << "unsigned" << i << "\n";
}

template <typename T>
void foo(const T& t) {
    std :: cout << "template" << t << "\n";
}

```

What do you think a call to `foo(42)` would print? **The answer is "template 42"**, and the reason for this is that integer literals are signed by default (they only become unsigned with the U suffix). When the compiler examines the overload candidates to choose from for this call, *it sees that the first function needs a conversion, while the second one matches perfectly, so that is the one it picks.*

7.16 参考

<http://developer.51cto.com/art/201208/351569.htm>

第八章 Effective

8.1 Effective C++

8.1.1 C++ 基本相关性能提升

1. 尽量以 **const, enum, inline** 替换 **#define** 编译过程: .c 文件–预处理–> i 文件–编译–>.o 文件–链接–>bin 文件

多了类型检查，因为#define 只是单纯的替换

2. 尽可能使用 **const** const 允许你告诉编译器和其他程序员某值应保持不变，只要“某值”确实是不该被改变的，那就该确实说出来。

3. 确定对象被使用前已先被初始化

赋值与初始化 C++ 规定，对象的成员变量的初始化动作发生在进入构造函数本体之前。所以应将成员变量的初始化置于构造函数的初始化列表中

```
ABEntry::ABEntry( const std::string& name, const std::string& address,  
      const std::list<PhoneNumber>& phones)  
{  
    theName = name;           // 这些都是赋值，而非初始化  
    theAddress = address;     // 这些成员变量在进入函数体之前已调用默认构  
    造函数，接着又调用赋值函数，  
    thePhones = phones;       // 即要经过两次的函数调用。  
    numTimesConsulted = 0;  
}  
  
ABEntry::ABEntry( const std::string& name, const std::string& address,  
      const std::list<PhoneNumber>& phones)  
:theName(name),           // 这些才是初始化  
  theAddress(address),     // 这些成员变量只用相应的值进行拷贝构造函  
  thePhones(phones),  
  numTimesConsulted(0){}
```

Note C++ 有着十分固定的“成员初始化次序”。基类总是在派生类之前被初始化，而类的成员变量总是以其说明次序被初始化。所以：当在成员初始化列表中列各成员时，最好总是以其声明次序为次。

8.1.2 C++ 构造/析构/赋值性能提升

4.C++ 默认编写并调用哪些函数 如果你自己没声明，编译器就会为类声明（编译器版本的）一个拷贝构造函数，一个拷贝赋值操作符和一个析构函数。此外如果你没有声明任何构造函数，编译器也会成为你声明一个默认构造函数。所有这些函数都是public 且inline，编译器产生规则如下：

- 调用准则：惟有当这些函数被需要（被调用），它们才会被编译器创建出来。即有需求，编译器才会创建它们。
- 析构函数：编译器产生的析构函数是个non-virtual，除非这个类的基类自身声明有virtual 析构函数
- 拷贝构造：至于拷贝构造函数和拷贝赋值操作符，编译器创建的版本只是单纯地将来源对象的每一个非静态成员变量拷贝到目标对象（浅拷贝）
- 构造函数：如一个类声明了一个构造函数（无论有没参数），编译器就不再为它创建默认构造函数

5. 若不想使用编译器自动生成的函数，就该明确拒绝 为驳回编译器自动（暗自）提供的机能，可将相应的成员函数声明为private 并且不予实现。使用像noncopyable 这样的基类也是一种做法，除此 C++11 提供=delete 关键字可以完成该操作。

6. 为多态基类声明 virtual 析构函数 当基类的指针指向派生类的对象的时候，当我们使用完，对其调用delete 的时候，其结果将是未有定义——基类成分通常会被销毁，而派生类的充分可能还留在堆里。这可是形成资源泄漏、败坏之数据结构、在调试器上消费许多时间

消除以上问题的做法很简单：给基类一个virtual 析构函数。此后删除派生类对象就会如你想要的那般。

STL容器都不带virtual 析构函数，所以最好别派生它们

// Note:

- 1、带有多态性质的基类应该声明一个virtual 析构函数。如果一个类带有任何virtual 函数，它就应该拥有一个virtual 析构函数
- 2、一个类的设计目的不是作为基类使用，或不是为了具备多态性，就不该声明virtual 析构函数

7. 别让异常逃离析构函数 析构函数绝对不要吐出异常。如果一个被析构函数调用的函数可能抛出异常，析构函数应该捕捉任何异常，然后吞下它们（不传播）或结束程序

8. 决不让构造和析构过程中调用 virtual 函数 基类的构造函数的执行要早于派生类的构造函数，当基类的构造函数执行时，派生类的成员变量尚未初始化。派生类的成员变量没初始化，即为指向虚函数表的指针vptr 没被初始化又怎么去调用派生类的virtual 函数呢？析构函数也相同，派生类先于基类被析构，又如何去找派生类相应的虚函数？

9. 令 operator= 返回一个 reference to *this 对于赋值操作符，我们常常要达到这种类似效果，即连续赋值

```
int x, y, z;  
x = y = z = 15;
```

赋值采用右结合律，所以上述连锁赋值被解析为

```
x = (y = (z = 15));
```

这里15先被赋值给z，然后其结果(更新后的z)再赋值给y，然后其结果(更新后的y)再赋值给x。

为了实现“连锁赋值”，赋值操作符必须返回一个“引用”指向操作符的左侧实参，这个协议适合所有的赋值运算。

```
Widget& operator = (const Widget &rhs)  
{  
    ...  
    return *this;  
}  
  
class Widget{  
public:  
    Widget& operator+=(const Widget& rhs) //这个协议使用于所有赋值相关运算  
        (+=, -=, *=等)  
    {  
        ...  
        return *this;  
    }  
  
    Widget& operator= (int rhs)  
    {  
        ...  
        return *this;  
    }  
};
```

10. 在 operator= 中处理“自我赋值” 自我赋值如下所示：

```
Widget w;  
w = w;  
a[i] = a[j]; //i == j or i != j  
*px = *py; // px, py 指向同个地址;
```

```
// 该怎么做
Widget& Widget::operator=(const Widget& rhs)
{
    Widget temp(rhs);
    swap(temp);
    return *this;
}
```

“许多时候一群精心安排的语句就可以导出异常安全 (new 新对象时出错，那么原来的东西亦将不复存在) (自我赋值) 的代码。”以下 note 给出正确做此事的规矩.

- 1、确保当对象自我赋值时 `operator =` 有良好行为。其中技术包括比较“来源对象”和“目标对象”的地址、精心周到的语句顺序、以及 `copy-and-swap`
- 2、确定任何函数如果操作一个以上的对象，而其中多个对象是同一个对象时，其行为仍然正确

11. 复制对象时勿忘其每一个成员 如果你为 class 添加一个成员变量，你必须同时修改复制构造函数和赋值操作符函数，当使用继承时，子类需重写拷贝构造，且不要忘了基类的成分

```
PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
: Customer(rhs),           // 调用 Base类的拷贝构造函数
  priority(rhs.priority)
{
    logCall("PriorityCustomer::Copy::constructor");
}

PriorityCustomer& operator=(const PriorityCustomer& rhs)
{
    Customer::operator=(rhs); // 对Base 类成分进行赋值动作
    priority = rhs.priority;
    return (*this);
}
```

8.1.3 资源管理

12. 以对象管理资源 使用 `shared_ptr` 等代理对象管理指针

13. 在资源管理类中小心拷贝行为

14. 在资源管理类中提供对原始资源的访问

15. 成对使用 `new` 和 `delete` 时要采取相同形式

16. 以独立语句将 `newed` 对象置入智能指针

8.1.4 设计与声明

17. 让接口容易被正确使用，不易被误用
18. 设计 `class` 犹如设计 `type`
19. 宁以 `pass-by-reference-to-const` 替代 `pass-by-value` 这种传递方式效率高得多：没有任何构造函数或析构函数被调用，因为没有任何新对象被创建。
20. 必须返回对象时，别妄想返回其 `reference` 函数创建新对象的途径有二：在栈空间和堆空间
 - **栈上：**即在函数内的局部变量。局部变量在函数返回后就没有存在的意义，若还对它“念念不忘”，将带来灾难性后果。所以传引用在栈上不可行
 - **堆上：**在堆上构造一个对象，并返回。看似可行，也埋下了资源泄漏的危险。谁该对这对对象实施 `delete` 呢？别把这种对资源的管理寄托完全托于用户。所以传引用在堆上不可行
 - **静态对象：**出于我们对多线程安全性的疑虑，以及当线程中两个函数对单份对象的处理也可能带来不可测行为。所以静态对象也是不可行的一个“必须返回新对象”的函数的正确写法是：就让那个函数返回一个新对象
21. 将成员变量声明为 `private`
22. 宁以 `non-member`、`non-friend` 替换 `member` 函数 这样做可以增加封装性、包裹弹性和机能扩充性
23. 若所有参数皆需类型转换，请为此采用 `non-member` 函数
24. 考虑写出一个不抛异常的 `swap` 函数

8.1.5 实现

25. 尽可能延后变量定义式的出现时间
26. 尽量少做转型动作
27. 避免返回 `handles` 指向对象内部成分
28. 为“异常安全”而努力是值得的
29. 透彻了解 `inlining` 的里里外外
30. 将文件间的编译依存关系降至最低

8.1.6 继承与面向对象设计

31. 确定你的 **public** 继承塑模出 **is-a** 关系

32. 避免遮掩继承而来的名称

33. 区分接口继承和实现继承 **public** 继承分为:1、接口继承2、实现继承

- 成员函数的接口总是会被继承
- 声明一个纯虚函数的目的是为了让派生类继承函数接口-”变化性”
- 声明一个虚函数的目的是让派生类继承该函数的接口和缺省实现-”不变性，与变化性”
- 声明一个非虚函数的目的是为了令派生类继承函数的接口及一份强制性实现-”不变性，凌驾特异性”

34. 考虑 **virtual** 函数以外的其它选择

35. 绝不重新定义继承而来的 **non-virtual** 函数

36. 绝不重新定义继承而来的缺省参数值

37. 通过符合塑模出 **has-a** 或 “根据某物实现出”

8.2 More Effective C++

8.2.1 基础议题 (Basics)

1. 仔细区别 **pointers** 和 **references** 在任何情况下都不能使用指向空值的引用。一个引用必须总是指向某些对象。在 C ++ 里，引用应被初始化，不存在指向空值的引用这个事实意味着使用引用的代码效率比使用指针的要高。因为在使用引用之前不需要测试它的合法性。

指针与引用的另一个重要的不同是指针可以被重新赋值以指向另一个不同的对象。但是引用则总是指向在初始化时被指定的对象，以后不能改变

总的来说，在以下情况下你应该使用指针，一是你考虑到存在不指向任何对象的可能（在这种情况下，你能够设置指针为空），二是你需要能够在不同的时刻指向不同的对象（在这种情况下，你能改变指针的指向）。如果总是指向一个对象并且一旦指向一个对象后就不会改变指向，那么你应该使用引用。还有一种情况，就是当你重载某个操作符时，你应该使用引用

2. 最好使用 C++ 转型操作符 `static_cast<double>(value)`

3. 绝对不要以多态方式处理数组 继承后仍然使用基类数组，导致数组的下标运算出现问题，因为数组是根据元素的 `sizeof` 进行计算下一个元素出现的位置，如果派生类的 `sizeof` 值不同，将会出现内存错误。

```

#include <iostream>
using namespace std;

struct B
{
    virtual void print() const{cout<<"base<print () "<<endl;}
};

struct D : B
{
    void print() const{cout<<"derived<print () "<<endl;}
    int id; //如果没有此句，执行将正确，因为基类对象和子类对象长度相同
};

int fun( const B array [] , int size )
{
    for( int i = 0; i < size; ++i )
    {
        array [ i ]. print();
    }
}

int main()
{
    B barray [ 5 ];
    fun( barray , 5 );
    D darray [ 5 ];
    fun( darray , 5 );
}

```

4. 避免无用的 default constructors

8.2.2 操作符 (Operators)

5. 对定制的“类型转换函数”保持警觉 定义类似功能的函数，而抛弃隐式类型转换，使得类型转换必须显示调用。例如 String 类没有定义对Char* 的隐式转换，而是用c_str 函数来实施这个转换。拥有单个参数（或除第一个参数外都有默认值的多参数）构造函数的类，很容易被隐式类型转换，最好加上 explicit 防止隐式类型转换

隐身类类型抓换

- 可以用 单个形参来调用的构造函数定义了从 形参类型到该类类型的一个隐式转换，这里应该注意的是，“可以用单个形参进行调用”并不是指构造函数只能有一个形参，而是它可以有多个形参，但那些形参都是有默认实参的。

```
#include <string>
```

```

#include <iostream>
using namespace std ;
class BOOK //定义了一个书类
{
private:
    string _bookISBN ; //书的ISBN号
    float _price ; //书的价格

public:
    //定义了一个成员函数，这个函数即是那个“期待一个实参为类类型的函数”
    //这个函数用于比较两本书的ISBN号是否相同
    bool isSameISBN( const BOOK & other ){
        return other._bookISBN==_bookISBN;
    }

    //类的构造函数，即那个“能够用一个参数进行调用的构造函数”（虽然
    //它有两个形参，但其中一个有默认实参，只用一个参数也能进行调用）
    BOOK( string ISBN , float price=0.0f ):_bookISBN(ISBN),_price(price)
    {}

};

int main()
{
    BOOK A("A-A-A");
    BOOK B("B-B-B");

    cout<<A.isSameISBN(B)<<endl; //正经地进行比较，无需发生转换

    cout<<A.isSameISBN(string("A-A-A"))<<endl; //此处即发生一个隐式转
    //换：string类型-->BOOK类型，借助BOOK的构造函数进行转换，以满足
    //isSameISBN函数的参数期待。
    cout<<A.isSameISBN(BOOK("A-A-A"))<<endl; //显式创建临时对象，
    //也即是编译器干的事情。

    system("pause");
}

```

代码中可以看到，isSameISBN 函数是期待一个BOOK 类类型形参的，但我们却传递了一个string类型的给它，这不是它想要的啊！还好，BOOK 类中有个构造函数，它使用一个string 类型实参进行调用，编译器调用了这个构造函数，隐式地将stirng 类型转换为BOOK 类型（构造了一个BOOK 临时对象），再传递给isSameISBN 函数。

隐式类类型转换还是会带来风险的，正如上面标记，隐式转换得到类的临时变量，完成操作后就消失了，我们构造了一个完成测试后被丢弃的对象。

- 隐式类型转换运算符只是一个样子奇怪的成员函数：operator **关键字**，其后跟一个类型符号。你不用定义函数的返回类型，因为返回类型就是这个函数的名字。例如为了允许Rational(有

理数)类隐式地转换为double类型(在用有理数进行混合类型运算时,可能有用),你可以如此声明Rational类

```
class Rational {  
public:  
    ...  
    operator double() const; // 转换 Rational 类成 double 类型  
};  
  
Rational r(1, 2); // r 的值是 1/2  
double d = 0.5 * r; // 隐式转换 r 到 double,
```

避免隐身转换 一般来说,越有经验的C++程序员就越喜欢避开类型转换运算符。例如在C++标准库委员会工作的人员是对此领域最有经验的,他们加在库函数中的string类型没有包括隐式地从string转换成C风格的char*的功能,而是定义了一个成员函数c_str用来完成这个转换,这是巧合么?我看不是。

- 声明装换功能的函数,如string的c_str()函数
- explicit operator Type() const;
- explicit BOOK(string ISBN, float price=0.0f);

6. 区别 increment/decrement 操作符的前置和后置形式

7. 千万不要重载 &&, || 和, 操作符

8. 了解各种不同意义的 new 和 delete

new操作符(new operator)和new操作(operator new)的区别 当你写这样的代码:string *ps = new str;你使用的new是new操作符。这个操作符就象sizeof一样是语言内置的,你不能改变它的含义,它的功能总是一样的。它要完成的功能分成两部分。**第一部分**是分配足够的内存以便容纳所需类型的对象。**第二部分**是它调用构造函数初始化内存中的对象。new操作符总是做这两件事情,你不能以任何方式改变它的行为。

你所能改变的是如何为对象分配内存。new操作符调用一个函数来完成必需的内存分配,你能够重写或重载这个函数来改变它的行为。new操作符为分配内存所调用函数的名字是operator new。函数operator new通常这样声明: void * operator new(size_t size);

就象malloc一样,operator new的职责只是分配内存。它对构造函数一无所知。

placement new

8.2.3 异常 (Exceptions)

-堆栈辗转开解 stack-unwinding 如果一个函数中出现异常,在函数内即通过try..catch捕捉的话,可以继续往下执行;如果不捕捉就会抛出(或通过throw显式抛出)到外层函数,则当前函数

会终止运行，释放当前函数内的局部对象（局部对象的析构函数就自然被调用了），外层函数如果也没有捕捉到的话，会再次抛出到更外层的函数，该外层函数也会退出，释放其局部对象……如此一直循环下去，直到找到匹配的 catch 子句，如果找到 main 函数中仍找不到，则退出程序。

9. 利用 **destructors** 避免泄漏资源

10. 在 **constructors 内阻止资源泄漏** 使用智能指针在初始化列表中进行初始化堆上资源可以防止异常情况下的资源泄漏，其实，防止内存泄漏就用智能指针的手段就可以了

11. 禁止异常流出 **destructors 之外** 这一条讲得其实是捕获析构函数里的异常的重要性。第一是防止程序调用 terminate 终止（这里有个名词叫：堆栈辗转开解 stack-unwinding）；第二是析构函数内如果发生异常，则异常后面的代码将不执行，无法确保我们完成我们想做的清理工作。

之前我们知道，析构函数被调用，会发生在对象被删除时，如栈对象超出作用域或堆对象被显式 delete (还有继承体系中，virtual 基类析构函数会在子类对象析构时调用)。除此之外，在异常传递的堆栈辗转开解 (stack-unwinding) 过程中，异常处理系统也会删除局部对象，从而调用局部对象的析构函数，而此时如果该析构函数也抛出异常，C++ 程序是无法同时处理两个异常的，就会调用 terminate() 终止程序 (会立即终止，连局部对象也不释放)。另外，如果异常被抛出，析构函数可能未执行完毕，导致一些清理工作不能完成。

所以不建议在析构函数中抛出异常，如果异常不可避免，则应在析构函数内捕获，而不应当抛出

12. 了解“抛出一个 exception”与“传递一个参数”或“调用一个虚函数”之间的差异

13. 以 **by reference 方式捕捉 exceptions 提高效率**

14. 明智运用 exception specifications 毫无疑问，异常规格是一个引人注目的特性。它使得代码更容易理解，因为它明确地描述了一个函数可以抛出什么样的异常

15. 了解异常处理的成本

8.2.4 效率 (Efficiency)

16. 谨记 80-20 法则

17. 考虑使用 lazy evaluation (缓式评估)

18. 分期摊还预期的计算成本

19. 了解临时对象的来源

20. 协助完成“返回值优化 (RVO)

21. 利用重载技术避免隐式类型转换
22. 考虑以操作符复合形式 (`op=`) 取代其独身形式 (`op`)
23. 考虑使用其它程序库
24. 了解 `virtual functions`、`multiple inheritance`、`virtual base classes`、`runtime type identification` 的成本

8.2.5 技术 (Techniques, Idioms, Patterns)

25. 将 `constructor` 和 `non-member functions` 虚化

26. 限制某个 `class` 所能产生的对象数量

27. 要求 (或禁止) 对象产生于 `heap` 中

28. Smart Pointer (智能指针)

29. Reference counting (引用计数)

30. Proxy classes (替身类、代理类)

8.2.6 杂项讨论 (Miscellany)

31. 在未来时态下发展程序

32. 将非尾端类设计为抽象类

8.3 Effective Modern C++

8.3.1 Deducing Types

8.3.2 auto

8.3.3 Moving to Modern C++

8.3.4 Smart Pointers

8.3.5 Rvalue References, Move Semantics, and Perfect Forwarding

8.3.6 Lambda Expressions

8.3.7 The Concurrency API

8.3.8 Tweaks

8.4 Effective STL

8.4.1 容器

1. 慎重选择容器类型，根据需要选择高效的容器类型

2. 不要试图编写独立于容器类型的代码

3. 确定容器中的对象拷贝正确而高效。也就是防止在存在继承关系时发生剥离

4. 调用 `empty` 而不是检查 `size()` 是否为 0 来判断容器是否为空 原因是调用 `empty` 比检查 `size()` 更加高效

5. 尽量使用区间成员，而不是多次使用与之对应的单元素成员函数，原因是这样更加高效 如尽量使用 `vector` 的 `assign` 或 `insert` 成员函数，而不是一直使用 `push_back`

6. 小心 C++ 编译器最烦人的分析机制

7. 如果在容器中包含了能过 `new` 操作创建的指针，切记在容器对象析构前将指针 `delete` 掉 使用智能指针如 `shared_ptr`(C++11) 是个比较好的选择

8. 切勿创建包含 `auto_ptr` 的容器对象 `auto_ptr` 不是采用引用计数实现的，放进容器中就等着出问题吧，特别是当对容器使用 `sort` 这类会对元素进行赋值的算法之后

9. 慎重选择删除元素的方法 这一点很重要，因为在删除一个元素之后会导致一些迭代器、指针及引用失效。所以如果手写循环删除元素一定要小心。最好的方法是vector、string、deque 采用erase-remove 这样的习惯写法，方便又安全，list 应当使用成员函数remove，关联容器使用成员函数erase。

要小心的一点是算法remove 并不是真正的删除容器中的元素，因为它没有这个能力，它只能将无用的元素移去末尾。之后你必须用成员函数erase 删除无用区间

10. 了解分配器 allocator 的约定和限制

11. 理解自定义分配器合理用法

12. 切勿对 STL 容器的线程安全性有不切实际的依赖 请手工保证线程安全，不要依赖STL

8.4.2 vector And string

13. 尽量用 vector 和 string 代替动态分配的数组

14. 使用 reserve 来避免不必要的内存重新分配 成员函数reserve 可以为容器预留一块内存，当容器大小（size）增大时，不用重新分配内存，可以提高效率。需要缩减分配的内存时可以使用后面提到的swap技术，或者使用成员函数.shrink_to_fit

15. 注意 string 实现的多样性

16. 了解如何把 vector 和 string 数据传给旧的 C API vector 请使用 &vector[0]，当然先要确定容器不为空。string 使用c_str() 方法

17. 使用“swap 技巧”除去多余的容量 与临时变量交换，然后利用临时变量的 RAII 机制，保证了在任何情况下，使用对象时先构造对象，最后析构对象。从而在析构该临时对象时将空间释放掉

18. 避免使用 vector<bool> 因为它不是容器，它是用位来存储bool 值的，所以它也没有迭代器，不能对其使用标准算法。最好使用deque <bool> 或bitset 代替之

8.4.3 关联容器

19. 理解相等 equality 和等价 equivalence 的区别 在实际中相等的概念是基于 operator== 的。find 对“相同”的定义是相等，是以 operator== 为基础的；set::insert 对“相同”的定义是等价，是以operator< 为基础的

20. 为包含指针的关联容器指定比较类型 默认情况的比较类型是 less<T>，所以只会比较指针的值，而不是指针指向的值，所以需要手工指定比较类型，以比较指针指向的值。

21. 总是让比较函数在等值情况下返回 `false` 特别是在关联容器基于比较函数判断等价的情况下。否则会导致相同的值按照定义是不等价的

22. 切勿直接修改 `set` 或 `multiset` 中的键 会破坏该数据结构。而`map` 中的键被`const` 修饰，不能更改

23. 考虑用排序的 `vector` 替代关联容器

24. 当效率至关重要时，请在 `map::operator[]` 与 `map::insert` 之间做出谨慎选择 选择的标准是：当向`map` 中添加元素时，要优先选择`insert`；当更新已经在`map` 中的元素的值时，要优先选择`operator[]`

25. 熟悉散列容器 `unordered_map`、`unordered_multimap`、`unordered_set`、`unordered_multiset`

8.4.4 迭代器

26. 尽量使用 `iterator`, 而不是 `const_iterator`、`reverse_iterator` 及 `const_reverse_iterator` 使用`iterator`会减少很多麻烦，所以使用 STL 时尽量使用`iterator`

27. 使用 `distance` 和 `advance` 将容器的 `const_iterator` 转换成 `iterator` 首先明确的是不能使用`const_cast`进行转换，因为这两个根本就是两个完全不相关的类，比`string`和`complex<double>`之间的关系还远。可以使用下面的技术转换，`i`是`iterator`类型，`ci`是`const_iterator`类型：

```
typedef IntDeque::const_iterator ConstIter;
advance(i, distance<ConstIter>(i, ci)); // <ConstIter> 不能少，否则编译通不过
```

28. 正确理解由 `reverse_iterator` 的 `base()` 成员函数所产生的 `iterator` 的用法 向右有一个位置的偏移，保证了对于插入操作而言，`ri`和`ri.base()`是等价的

对于删除而言则不是等价的，要先`++ri`，如`v.erase((++ri).base());`

29. 对非格式化的逐个字符的输入考虑使用 `istreambuf_iterator` 比`istream_iterator` 效率更高而且更方便（不用清`iOS::skipws` 标志）。相对的还有`ostreambuf_iterator`

8.4.5 算法

30. 确保目标区间足够大：

`transform(values.begin(), values.end(), results.end(), transmogrify);`

可以更改为

`transform(values.begin(), values.end(), back_inserter(results), transmogrify);`

应该使用插入迭代器。插入迭代器有`inserter`、`back_inserter`、`front_inserter`、`ostream_iterator`

31. 了解各种与排序有关的选择

非稳定排序 都要求随机访问迭代器 RandomAccessIterator

- sort: 基于快排的排序
- partial_sort: 部分排序, 指定排序区间
- nth_element: 将排序后位于前 n 个位置的元素放在前 n 个位置, 但不对它们进行排序。

稳定排序

- stable_sort: 稳定排序, 排序后等价元素的相对位置不会变
- list::sort

分割序列的算法 只需要双向迭代器

- partition
- stable_partition: 不改变相同元素顺序

32. 如果确实需要删除元素, 则需要在 remove 这一类算法之后调用成员函数 erase 算法remove不能真正删除元素, 因为它没有这个能力, 必须调用类似erase 这样的成员函数才行, list 的成员remove 可以真正删除元素。类似remove 这样的算法有remove_if、unique

33. 对包含指针的容器使用 remove 这一类算法时要特别小心 因为被“删除”的值可能也不存在于序列最后面, 而是被覆盖了。与remove 算法的实现有关

34. 了解哪些算法要求使用排序区间作为参数

35. 通过 mismatch 或 lexicographical_compare 实现简单忽略大小写的字符串比较

36. 理解 copy_if 算法的正确实现

37. 使用 accumulate 或者 for_each 进行区间统计

8.4.6 函数子、函数子类、函数及其他

38. 遵循按值传递的原则来设计函数子类 在 STL 中, 函数对象往往会按值传递和返回, 所以在设计函数对象时要保证经过了值传递之后还能正常工作

39. 确保判别式 predicate 是纯函数 pure function “纯函数”是指返回值仅仅依赖于其参数的函数

40. 若一个类是函数子, 则就使它可配接 adaptable

41. 理解 `ptr_fun`、`mem_fun`、`mem_fun_ref` 的来由

42. 确保 `less<T>` 与 `operator<` 具有相同的语义

8.4.7 在程序中使用 STL

43. 算法调用优先于手写的循环

44. 容器的成员函数优于同名的算法 因为成员函数更高效，成员函数可以根据特定的容器及实现作出优化

45. 正确区分 `count`、`find`、`binary_search`、`lower_bound`、`upper_bound` 和 `equal_range` 根据需要选择最高效的算法

What You Want to Know	Algorithm to Use		Member Function to Use	
	On an Unsorted Range	On a Sorted Range	With a set or map	With a multiset or multimap
Does the desired value exist?	find	binary_search	count	find
Does the desired value exist? If so, where is the first object with that value?	find	equal_range	find lower_bound (see below)	find or lower_bound
Where is the first object with a value not preceding the desired value?	find_if	lower_bound	lower_bound	lower_bound
Where is the first object with a value succeeding the desired value?	find_if	upper_bound	upper_bound	upper_bound
How many objects have the desired value?	count	equal_range	count	count
Where are all the objects with the desired value?	find (iteratively)	equal_range	equal_range	equal_range

46. 考虑使用函数对象而不是函数作为 STL 算法参数 一个事实是 STL 的 `sort` 比 C 语言标准中的 `qsort` 更快。原因就是 `sort` 使用函数对象，而 `qsort` 使用函数指针作为参数。如果一个函数对象的 `operator()` 已经被声明为内联的，那么其在函数体内也是内联的，但对于函数指针就不一样了，通过函数指针传递的函数不大可能会被优化成内联的，即使该函数被声明为内联的。

47. 避免产生“直写型（write-only）”代码 “直写型”代码复杂而难以理解，不便于维护。”直写型“代码就是那种一句话完成 n 多功能的代码，晦涩难懂。最好将”直写型“代码分解为易读的代码

48. 总是包含（#include）正确的头文件

49. 学会分析与 STL 相关的编译器诊断信息

8.5 提升 C++ 编程性能的技术

8.5.1 跟踪范例

关注点 本章引入的实际问题为：定义一个简单的 Trace 类，将当前函数名输出到日志文件中。Trace 对象会带来一定的开销，因此在默认情况下不会开启 **Trace** 功能。

问题是：怎么设计 Trace 类，使得在不开启 Trace 功能时引入的开销最小

解决方案

1. 宏：用宏来开关 Trace 功能很简单，在不开启时开销完全没有

```
#ifdef TRACE  
Trace trace("aaa");  
#endif
```

2. 静态状态变量：使用状态变量的话有一定的运行时开销，但能保证灵活性，是一种比较合理的选择

```
class Trace {  
public:  
    ...  
    static bool isTraceEnabled;  
    void Debug() {  
        if (isTraceEnabled) {  
            ...  
        }  
    }  
}
```

涉及技术- 延迟创建 原本的 Trace 类中内置 string 成员，这样在不开启 Trace 时也要承担构造和析构的开销。可以将其改为 string*，并在真正需要开启时再创建该成员。

如果 Trace 的开启时间远小于总时间，则此方法很有效，否则当动态创建的开销大于固定的 1 次构造和析构的开销时，原方法更好一些

8.5.2 虚函数

8.5.3 临时对象

关注点 如何避免产生不必要的临时对象

类型不匹配 在不同类型间的赋值容易无意中导致临时对象的创建。可以通过在单参数构造函数前加explicit 来避免这种隐式的转换产生

避免重复创建相同的临时对象 临时对象有一个就够了.. 相同的对象的反复创建无形的增加了工作量

```
Complex a;
for (int i = 0; i < 10; ++i) {
    a += 1.0;
}
```

其中每次循环都会创建一个值为1.0的Complex对象。可以在循环外创建一个值为1.0的Complex对象，来减少这种开销：

```
Complex one(1.0);
for (int i = 0; i < 10; ++i) {
    a += one;
}
```

8.5.4 内存池

单线程内存池

关注点 默认的通用内存管理器的性能在特定场景下会造成一定的性能瓶颈。本章讨论的是在单线程环境下，每次分配固定大小和不固定大小的内存时，实现比通用new/delete 性能更好的内存池管理器

Rational 专用内存池 每次分配Rational 大小的内存块，用一个空闲链表维护已分配的空闲内存，在释放时重新将此内存块放回到链表中

```
class Rational {
    ...
    static list<char *> freeList;
    void *operator new(size_t size) {
        if (freeList.empty()) {
            return new char[sizeof(Rational)];
        } else {
            void *buf = freeList.back();
            freeList.pop_back();
            return buf;
        }
    }
}
```

```

    }
    void operator delete(void *ptr, size_t size) {
        freeList.push_back(ptr);
    }
};

```

此版本的内存池从不收缩，如果需要释放内存，则需要新增一个接口。

此版本的内存池与通用内存管理器相比，收益在于：

- 每次分配的大小为固定值，不用在空闲列表中进行大量的查找（直接返回末端指针）。
- 不用处理并发情况，没有临界区

固定大小内存池实现 不只针对 Rational，而是扩展为支持任意固定大小的类：

```

template <typename T>
class FixedSizeMemoryPool {
public:
    FixedSizeMemoryPool(): size_(sizeof(T))
    {}
    ~FixedSizeMemoryPool() {
        for (char *&p: freeList_) {
            delete [] p;
        }
    }
    void *Alloc() {
        if (freeList_.empty()) {
            return new char[size_];
        } else {
            void *buf = freeList_.back();
            freeList_.pop_back();
            return buf;
        }
    }
    void Free(void *buf) {
        freeList_.push_back(buf);
    }
private:
    list<char *> freeList_;
    const size_t size_;
};

// Rational则需要改为：
class Rational {
public:
    void *operator new(size_t size) {
        return pool.Alloc();
    }
    void operator delete(void *ptr, size_t size) {

```

```

        pool.Free(ptr);
    }
private:
    static FixedSizeMemoryPool<Rational> pool;
};

```

不定大小内存池 不定大小的内存池的管理方法与上面的版本不同，因为没有办法直接从链表中返回一个内存块（大小不同）。这里我们在需要时分配一个大的固定大小的内存块，每次分配单个对象的内存时就从这个内存块上分配，空间不够时就分配大的内存块。

随着通用性的增加，性能也在逐渐下降。因此，在非常需要性能时，牺牲一些灵活性通用性也许会有很好的效果。

多线程内存池

在 Alloc 和 Free 时加锁，其它保持不变

8.5.5 引用计数

关注点 C++ 使用了引用计数来解决垃圾回收问题，基本思想是把对象清除的责任从客户端代码转移给对象本身。

引用计数可以减少内存使用、避免内存泄漏，但在执行速度方面却可能会有坏处，尤其是在多线程环境中

引用计数的实现 方案 1：

```

class RefCountBase {
public:
    Attach() {
        ++refCount_;
    }
    Detach() {
        if (--refCount_ == 0) {
            delete this;
        }
    }
protected:
    RefCountBase(): refCount_(0) {}
    RefCountBase( const RefCountBase &rc): refCount_(0) {}
    RefCountBase &operator=( const RefCountBase &rc) {
        return *this;
    }
    virtual ~RefCountBase() {}

    size_t refCount_;
};

```

如类A继承自RefCountBase，为了实现引用计数，还需要一个代理类SmartPtr充当A的智能指针

```
template <typename T>
class SmartPtr {
public:
    SmartPtr(T *ptr = nullptr): ptr_(ptr) {}
    SmartPtr(const SmartPtr &sprt): ptr_(sprt.ptr_) {
        if (ptr_) {
            ptr_->Attach();
        }
    }
    SmartPtr &operator=(const SmartPtr &sprt) {
        if (sprt.ptr_) {
            sprt.ptr_->Attach();
        }
        if (ptr_)
            ptr_->Detach();

        ptr_ = sprt.ptr_;
        return *this;
    }
    T *operator->() { return ptr_; }
    T &operator*() ( return *ptr_ );
private:
    T *ptr_;
};
```

实现B：将计数功能放入SmartPtr中。去掉RefCountBase，而是在SmartPtr中增加一个size_t *count_，对ptr_的Attach操作变为++*count_，而Detach操作则变为--*count_。其它相同

并发引用计数 SmartPtr中需要同时对count_和ptr_进行操作，在并发环境下这就意味着需要在操作前后加锁，来保证对两个对象的原子操作，从而避免数据竞争

引用计数的性能 实现A中需要对原类进行修改，如果不能进行这种修改，则只能使用实现B。实现B中因为需要操作两个堆上的成员（count_和ptr_），创建和清除性能会比实现A差一些

优点：

- 1、防止内存泄露
- 2、高效的赋值操作。尤其是作为写时复制(COW)的重要环节，如果赋值后很少有修改操作的话，相比于深复制，引用计数的收益非常明显
- 3、节省内存空间。尤其是体积非常大的对象
- 4、可以方便的实现RAII。将引用计数的Detach操作变为某种关闭操作，则可很方便地实现RAII

缺点：

-
- 1、 COW中如果修改较多，那么性能相比深复制不一定有提升
 - 2、 在并发环境下对它的操作还有锁的开销，可能会影响性能比较多
-

下列条件会增加引用计数的收益：

- 1、 目标对象消耗大量资源
- 2、 资源的分配和释放很昂贵
- 3、 目标对象高度共享
- 4、 引用的创建和清除很廉价

8.5.6 循环引用

参考文献：<http://www.cnblogs.com/TianFang/archive/2008/09/20/1294590.html>

引用计数是一种便利的内存管理机制，但它有一个很大的缺点，那就是不能管理循环引用的对象。一个简单的例子如下：

```
#include <string>
#include <iostream>
#include <boost/shared_ptr.hpp>
#include <boost/weak_ptr.hpp>

class parent;
class children;

typedef boost::shared_ptr<parent> parent_ptr;
typedef boost::shared_ptr<children> children_ptr;

class parent
{
public:
~parent() { std::cout << "destroying parent\n"; }

public:
children_ptr children;
};

class children
{
public:
~children() { std::cout << "destroying children\n"; }

public:
parent_ptr parent;
};

void test()
```

```

{
    parent_ptr father(new parent());
    children_ptr son(new children);

    father->children = son;
    son->parent = father;
}

void main()
{
    std::cout << "begin test ... \n";
    test();
    std::cout << "end test .\n";
}

```

运行该程序可以看到，即使退出了test 函数后，由于parent和children 对象互相引用，它们的引用计数都是 1，不能自动释放，并且此时这两个对象再无法访问到。这就引起了 c++ 中那臭名昭著的内存泄漏。

一般来讲，解除这种循环引用有下面有三种可行的方法：

1. 当只剩下最后一个引用的时候需要手动打破循环引用释放对象。
2. 当 parent 的生存期超过 children 的生存期的时候，children 改为使用一个普通指针指向 parent。
3. 使用弱引用的智能指针打破这种循环引用。

虽然这三种方法都可行，但方法 1 和方法 2 都需要程序员手动控制，麻烦且容易出错。这里主要介绍一下第三种方法和 boost 中的弱引用的智能指针boost::weak_ptr。

强引用弱引用 一个强引用当被引用的对象活着的话，这个引用也存在（就是说，当至少有一个强引用，那么这个对象就不能被释放）。boost::shared_ptr 就是强引用。

相对而言，弱引用当引用的对象活着的时候不一定存在。仅仅是当它存在的时候的一个引用。弱引用并不修改该对象的引用计数，这意味着这弱引用它并不对对象的内存进行管理，在功能上类似于普通指针，然而一个比较大的区别是，弱引用能检测到所管理的对象是否已经被释放，从而避免访问非法内存。

通过弱引用打破循环引用 由于弱引用不更改引用计数，类似普通指针，只要把循环引用的一方使用弱引用，即可解除循环引用。对于上面的那个例子来说，只要把 children 的定义改为如下方式，即可解除循环引用：

```

class children
{
public:
    ~children() { std::cout << "destroying children\n"; }

public:

```

```
    boost::weak_ptr<parent> parent;  
};
```

最后值得一提的是，虽然通过弱引用指针可以有效的解除循环引用，但这种方式必须在程序员能预见会出现循环引用的情况下才能使用，也可以说这个仅仅是一种编译期的解决方案，如果程序在运行过程中出现了循环引用，还是会造内存泄漏的。因此，不要认为只要使用了智能指针便能杜绝内存泄漏。毕竟，对于 C++ 来说，由于没有垃圾回收机制，内存泄漏对每一个程序员来说都是一个非常头痛的问题。

实践操作示例 参考文献：<https://blog.csdn.net/xtzmm1215/article/details/45868835>

```
class B;  
class A  
{  
public:// 为了省去一些步骤这里 数据成员也声明为 public  
    weak_ptr<B> pb;  
    //shared_ptr<B> pb;  
    void doSomthing()  
    {  
        shared_ptr<B> pp = pb.lock();  
        if(pp)//通过 lock() 方法来判断它所管理的资源是否被释放  
        {  
            cout<<"sb_use_count :"<<pp.use_count()<<endl;  
        }  
    }  
  
    ~A()  
    {  
        cout << "kill_A\n";  
    }  
};  
  
class B  
{  
public:  
    //weak_ptr<A> pa;  
    shared_ptr<A> pa;  
    ~B()  
    {  
        cout <<"kill_B\n";  
    }  
};  
  
int main(int argc, char** argv)  
{  
    shared_ptr<A> sa(new A());  
    shared_ptr<B> sb(new B());
```

```
if (sa && sb)
{
    sa->pb=sb;
    sb->pa=sa;
}
sa->doSomthing();
cout<<"sb use count :"<<sb.use_count()<<endl;
return 0;
}
```

8.5.7 代码优化

8.5.8 设计优化

8.5.9 可伸缩性

8.5.10 系统体系结构相关性

8.6 深入探索 C++ 对象模型

8.7 STL 源码剖析

8.8 参考

Effective C++ <http://blog.csdn.net/shenzi/article/details/5601038>

More Effective C++ <http://www.cnblogs.com/tianyajuanke/archive/2012/11/29/2795131.html>

Effective Modern C++ <http://blog.csdn.net/cteng/article/details/41912179>

Effective STL http://blog.csdn.net/xzz_hust/article/details/9624613

提高 C++ 性能的编程技术 <http://www.cnblogs.com/fuzhe1989/p/3546598.html>

STL 源码剖析 <http://blog.csdn.net/shenya1314/article/details/54923558>

第九章 OpenMP 并行技术

第十章 GPU 并行技术

10.1 OpenCL

10.2 CUDA