
C++ 内存管理

从基础到高级的完整指南

涵盖内容

- 内存布局与分配机制
 - 智能指针详解 (unique_ptr, shared_ptr, weak_ptr)
 - 内存对齐原理与优化
 - Placement New 与内存池
 - 内存泄漏检测与防范
 - RAII 与异常安全
 - 最佳实践与常见陷阱
-

面试准备 · 技术进阶 · 实战指南

作者：Aweo

2025 年 10 月

Contents

1 C++ 内存管理	3
1.1 内存布局概述	3
1.2 智能指针详解	4
1.2.1 unique_ptr - 独占所有权	4
1.2.2 shared_ptr - 共享所有权	6
1.2.3 weak_ptr - 弱引用	7
1.3 内存对齐	9
1.3.1 为什么需要内存对齐?	9
1.3.2 内存对齐示例	13
1.4 Placement New	14
1.5 内存池	16
1.6 内存泄漏详解	19
1.6.1 什么是内存泄漏?	19
1.6.2 常见内存泄漏场景	19
1.6.3 内存泄漏的危害	23
1.6.4 如何防止内存泄漏	24
1.6.5 内存泄漏检测工具	29
1.6.6 内存泄漏防范检查清单	32
1.7 内存管理最佳实践	36
1.8 总结	37


1 C++ 内存管理

1.1 内存布局概述

C++程序的内存主要分为以下几个区域：

1	// 内存布局示意（从低地址到高地址）		
2	<div><div></div><div>低地址</div></div>		
3	代码段(.text)	存放程序的机器码	
4	<div><div></div></div>		
5	常量区(.rodata)	存放字符串常量等	
6	<div><div></div></div>		
7	全局/静态区		
8	- .data段	已初始化的全局/静态变量	
9	- .bss段	未初始化的全局/静态变量	
10	<div><div></div></div>		
11	堆(Heap) ↓	动态分配，向下增长	
12	<div><div></div></div>		
13	... 自由空间 ...		
14	<div><div></div></div>		
15	栈(Stack) ↑	局部变量，向上增长	
16	<div><div></div></div>		
17	内核空间		
18	<div><div></div><div>高地址</div></div>		

各区域特点：

1	// 1. 栈 (Stack)		
2	void func() {		
3	int local = 42;	// 栈上分配	
4	char buffer[100];	// 栈上分配	
5	// 函数返回时自动释放		
6	}		
7	// 特点：		
8	// - 自动管理，LIFO		
9	// - 速度快（只需移动栈指针）		
10	// - 大小有限（通常1-8MB）		
11	// - 编译期确定大小		
12			
13	// 2. 堆 (Heap)		
14	void func() {		
15	int* p = new int(42);	// 堆上分配	
16	delete p;	// 手动释放	
17	}		
18	// 特点：		
19	// - 手动管理		
20	// - 速度较慢（需要分配器管理）		
21	// - 大小受限于物理内存		
22	// - 运行时确定大小		
23			
24	// 3. 全局/静态区		
25	int global_var = 10;	// .data段	
26	static int static_var = 20;	// .data段	
27	int uninit_global;	// .bss段（零初始化）	

```

28
29 // 特点：
30 // - 程序启动时分配
31 // - 程序结束时释放
32 // - 整个程序运行期间存在
33
34 // 4. 常量区
35 const char* str = "Hello"; // "Hello"存储在.rodata段
36 // 特点：
37 // - 只读，不可修改
38 // - 字符串字面量可能被合并

```

内存大小示例：

```

1  #include <iostream>
2
3  struct Empty {};
4  struct Padding {
5      char c;    // 1字节
6      int i;     // 4字节
7  };
8
9  int main() {
10     std::cout << "栈上对象:\n";
11     int stack_var;
12     std::cout << "sizeof(int): " << sizeof(int) << "\n";
13     std::cout << "sizeof(Empty): " << sizeof(Empty) << "\n"; // 1 (最小)
14     std::cout << "sizeof(Padding): " << sizeof(Padding) << "\n"; // 8 (对齐)
15
16     std::cout << "\n堆上对象:\n";
17     int* heap_var = new int;
18     std::cout << "指针大小: " << sizeof(heap_var) << "\n"; // 8 (64位系统)
19     delete heap_var;
20 }

```

1.2 智能指针详解

智能指针通过 RAII 自动管理内存，是现代 C++ 的核心工具。

1.2.1 unique_ptr - 独占所有权

```

1  #include <memory>
2
3  // 1. 基本使用
4  std::unique_ptr<int> p1(new int(42));
5  std::unique_ptr<int> p2 = std::make_unique<int>(42); // C++14, 推荐
6
7  // 访问
8  *p2 = 100;
9  int value = *p2;
10
11 // 自动释放 (作用域结束时)
12 {
13     auto p = std::make_unique<int>(42);
14 } // 自动delete
15

```

```

16 // 2. 数组版本
17 std::unique_ptr<int[]> arr(new int[10]);
18 arr[0] = 1; // 可以使用下标
19
20 auto arr2 = std::make_unique<int[]>(10); // C++14
21
22 // 3. 所有权转移 (移动)
23 std::unique_ptr<int> p3 = std::make_unique<int>(10);
24 std::unique_ptr<int> p4 = std::move(p3); // p3变为nullptr
25 // std::unique_ptr<int> p5 = p4; // 错误! 不能拷贝
26
27 // 4. 自定义删除器
28 auto deleter = [](int* p) {
29     std::cout << "Custom delete: " << *p << "\n";
30     delete p;
31 };
32 std::unique_ptr<int, decltype(deleter)> p5(new int(42), deleter);
33
34 // 文件句柄示例
35 auto file_deleter = [](FILE* fp) {
36     if (fp) fclose(fp);
37 };
38 std::unique_ptr<FILE, decltype(file_deleter)> file(
39     fopen("data.txt", "r"),
40     file_deleter
41 );
42
43 // 5. 工厂函数
44 template<typename T, typename... Args>
45 std::unique_ptr<T> create(Args&&... args) {
46     return std::make_unique<T>(std::forward<Args>(args)...);
47 }
48
49 // 6. 作为类成员
50 class Widget {
51     std::unique_ptr<Resource> resource; // 自动管理资源
52 public:
53     Widget() : resource(std::make_unique<Resource>()) {}
54     // 析构时自动释放resource
55 };
56
57 // 7. 释放所有权
58 auto p6 = std::make_unique<int>(42);
59 int* raw = p6.release(); // p6不再拥有, 返回裸指针
60 delete raw; // 需要手动删除
61
62 // 8. 重置
63 auto p7 = std::make_unique<int>(42);
64 p7.reset(); // 释放当前对象, p7变为nullptr
65 p7.reset(new int(100)); // 释放当前对象, 指向新对象

```

unique_ptr 的优势：

- 零开销（相比裸指针）
- 明确所有权语义

- 异常安全
- 不能拷贝，避免多次删除

1.2.2 shared_ptr - 共享所有权

```

1  #include <memory>
2
3  // 1. 基本使用
4  std::shared_ptr<int> sp1 = std::make_shared<int>(42); // 推荐
5  std::shared_ptr<int> sp2(new int(42)); // 不推荐 (两次内存分配)
6
7  // 2. 共享所有权
8  auto sp3 = std::make_shared<int>(100);
9  auto sp4 = sp3; // 拷贝，引用计数+1
10 auto sp5 = sp3; // 引用计数=3
11 std::cout << sp3.use_count() << "\n"; // 输出: 3
12
13 sp4.reset(); // 引用计数-1
14 // 当最后一个shared_ptr销毁时，对象才被删除
15
16 // 3. 从unique_ptr转换
17 auto up = std::make_unique<int>(42);
18 std::shared_ptr<int> sp6 = std::move(up); // 转移所有权
19
20 // 4. 自定义删除器
21 auto deleter = [](int* p) {
22     std::cout << "Delete: " << *p << "\n";
23     delete p;
24 };
25 std::shared_ptr<int> sp7(new int(42), deleter);
26
27 // 5. 数组支持 (C++17起)
28 std::shared_ptr<int[]> sp_arr(new int[10]); // C++17
29 sp_arr[0] = 1;
30
31 // C++20推荐
32 auto sp_arr2 = std::make_shared<int[]>(10);
33
34 // 6. 别名构造 (aliasing constructor)
35 struct Data {
36     int x, y;
37 };
38 auto sp_data = std::make_shared<Data>();
39 std::shared_ptr<int> sp_x(sp_data, &sp_data->x); // 指向x，但共享Data的生命周期
40
41 // 7. enable_shared_from_this
42 class Node : public std::enable_shared_from_this<Node> {
43 public:
44     std::shared_ptr<Node> getPtr() {
45         return shared_from_this(); // 安全地返回shared_ptr
46     }
47 };
48
49 auto node = std::make_shared<Node>();
50 auto ptr = node->getPtr(); // 正确

```

```

51
52 // 8. 循环引用问题 (需要weak_ptr解决)
53 struct Bad {
54     std::shared_ptr<Bad> next;
55 };
56
57 auto b1 = std::make_shared<Bad>();
58 auto b2 = std::make_shared<Bad>();
59 b1->next = b2;
60 b2->next = b1; // 循环引用, 内存泄漏!

```

shared_ptr 实现原理：

```

1  // 简化的shared_ptr实现
2  template<typename T>
3  class SimpleSharedPtr {
4      T* ptr;
5      size_t* ref_count; // 引用计数
6
7  public:
8      SimpleSharedPtr(T* p) : ptr(p), ref_count(new size_t(1)) {}
9
10     SimpleSharedPtr(const SimpleSharedPtr& other)
11         : ptr(other.ptr), ref_count(other.ref_count) {
12         ++(*ref_count); // 增加引用计数
13     }
14
15     ~SimpleSharedPtr() {
16         if (--(*ref_count) == 0) { // 减少引用计数
17             delete ptr;
18             delete ref_count;
19         }
20     }
21
22     T& operator*() { return *ptr; }
23     T* operator->() { return ptr; }
24     size_t use_count() const { return *ref_count; }
25 };
26
27 // make_shared的优势：一次内存分配
28 // new：两次分配 (对象+控制块)
29 // make_shared：一次分配 (对象和控制块在一起)
30
31 |-----|
32 | 对象数据 | 控制块 |
33 | T object | ref_count |
34 |          | weak_count |
35 |          | deleter    |

```

1.2.3 weak_ptr - 弱引用

```

1  #include <memory>
2
3  // 1. 基本使用
4  auto sp = std::make_shared<int>(42);

```

```

5  std::weak_ptr<int> wp = sp; // 不增加引用计数
6
7  std::cout << sp.use_count() << "\n"; // 1 (weak_ptr不影响)
8  std::cout << wp.use_count() << "\n"; // 1
9
10 // 2. 检查对象是否存在
11 if (wp.expired()) {
12     std::cout << "对象已被删除\n";
13 } else {
14     std::cout << "对象仍存在\n";
15 }
16
17 // 3. 访问对象 (需要转换为shared_ptr)
18 if (auto sp2 = wp.lock()) { // 临时提升为shared_ptr
19     std::cout << *sp2 << "\n";
20 } else {
21     std::cout << "对象已被删除\n";
22 }
23
24 // 4. 解决循环引用
25 struct Good {
26     std::shared_ptr<Good> next;
27     std::weak_ptr<Good> prev; // 使用weak_ptr打破循环
28 };
29
30 auto g1 = std::make_shared<Good>();
31 auto g2 = std::make_shared<Good>();
32 g1->next = g2;
33 g2->prev = g1; // 不会造成循环引用
34
35 // 5. 观察者模式
36 class Subject;
37
38 class Observer {
39     std::weak_ptr<Subject> subject; // 弱引用, 不控制生命周期
40 public:
41     void setSubject(std::shared_ptr<Subject> s) {
42         subject = s;
43     }
44
45     void notify() {
46         if (auto s = subject.lock()) {
47             // 使用subject
48         }
49     }
50 };
51
52 // 6. 缓存实现
53 class Cache {
54     std::map<std::string, std::weak_ptr<Resource>> cache;
55
56 public:
57     std::shared_ptr<Resource> get(const std::string& key) {
58         auto it = cache.find(key);

```



```

59     if (it != cache.end()) {
60         if (auto sp = it->second.lock()) {
61             return sp; // 缓存命中
62         }
63         cache.erase(it); // 已过期，删除
64     }
65
66     // 创建新资源
67     auto resource = std::make_shared<Resource>();
68     cache[key] = resource;
69     return resource;
70 }
71 };

```

智能指针选择指南：

```

1 // 1. 独占所有权 → unique_ptr
2 std::unique_ptr<Widget> widget = std::make_unique<Widget>();
3
4 // 2. 共享所有权 → shared_ptr
5 std::shared_ptr<Resource> shared = std::make_shared<Resource>();
6
7 // 3. 不控制生命周期 → weak_ptr
8 std::weak_ptr<Resource> observer = shared;
9
10 // 4. 工厂函数返回 → unique_ptr (可转为shared_ptr)
11 std::unique_ptr<Base> createObject() {
12     return std::make_unique<Derived>();
13 }
14
15 // 5. 容器中存储 → shared_ptr或unique_ptr
16 std::vector<std::unique_ptr<Widget>> widgets; // 独占
17 std::vector<std::shared_ptr<Resource>> resources; // 共享

```

1.3 内存对齐

内存对齐影响性能和跨平台兼容性。

1.3.1 为什么需要内存对齐？

1. 硬件访问效率

现代CPU访问内存不是逐字节访问，而是按“字(word)”访问（通常4或8字节）。

```

1 // CPU访问内存的方式
2 // 32位系统：每次读取4字节（地址0, 4, 8, 12...）
3 // 64位系统：每次读取8字节（地址0, 8, 16, 24...）
4
5 // 未对齐访问（假设int=4字节）
6
7 

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|

内存地址
8
9 └───int(addr=1)───┘
10
11 // CPU需要：
12 // 1. 读取地址0-3的4字节
13 // 2. 读取地址4-7的4字节

```

```

14 // 3. 拼接提取中间的int
15 // → 两次内存访问！
16
17 // 对齐访问
18
19 

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|


20
21 
22
23 // CPU只需一次内存访问！

```

2. 性能差异

```

1  #include <chrono>
2  #include <iostream>
3
4  struct Unaligned {
5      char c;
6      int i;    // 偏移1，未对齐
7  } __attribute__((packed));
8
9  struct Aligned {
10     char c;
11     // 3字节填充
12     int i;    // 偏移4，对齐
13 };
14
15 // 性能测试
16 void performance_test() {
17     const int N = 100000000;
18
19     // 未对齐访问
20     Unaligned* arr1 = new Unaligned[N];
21     auto start = std::chrono::high_resolution_clock::now();
22     for (int i = 0; i < N; ++i) {
23         arr1[i].i = i; // 未对齐写入
24     }
25     auto end = std::chrono::high_resolution_clock::now();
26     auto unaligned_time = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
27
28     // 对齐访问
29     Aligned* arr2 = new Aligned[N];
30     start = std::chrono::high_resolution_clock::now();
31     for (int i = 0; i < N; ++i) {
32         arr2[i].i = i; // 对齐写入
33     }
34     end = std::chrono::high_resolution_clock::now();
35     auto aligned_time = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
36
37     std::cout << "未对齐: " << unaligned_time << "ms\n";
38     std::cout << "对齐: " << aligned_time << "ms\n";
39     std::cout << "性能提升: " << (double)unaligned_time / aligned_time << "x\n";
40     // 典型结果：对齐访问快2-10倍

```

```

41
42     delete[] arr1;
43     delete[] arr2;
44 }

```

3. 跨平台兼容性

某些架构对未对齐访问的处理不同：

```

1  // x86/x64架构
2  // - 允许未对齐访问，但性能下降
3  int* p = (int*)((char*)buffer + 1); // 未对齐
4  *p = 42; // 可以运行，但慢
5
6  // ARM架构（早期版本）
7  // - 未对齐访问可能触发硬件异常
8  // - 或者读取错误的数据（静默失败）
9  int* p = (int*)((char*)buffer + 1);
10 *p = 42; // 可能崩溃！
11
12 // MIPS架构
13 // - 未对齐访问直接抛出异常
14 int* p = (int*)((char*)buffer + 1);
15 *p = 42; // 崩溃！

```

4. 原子操作要求

```

1  #include <atomic>
2
3  struct Data {
4      char c;
5      std::atomic<int> counter; // 必须对齐，否则原子操作失败
6  };
7
8  // 错误示例
9  struct __attribute__((packed)) Bad {
10     char c;
11     std::atomic<int> counter; // 未对齐，原子性无法保证！
12 };
13
14 // 在多线程环境下：
15 // - 未对齐的原子操作可能不是真正的原子操作
16 // - 导致数据竞争和未定义行为

```

5. SIMD 指令要求

```

1  #include <immintrin.h>
2
3  void simd_example() {
4      // SSE要求16字节对齐
5      alignas(16) float data1[4];
6      __m128 vec1 = _mm_load_ps(data1); // ✓ 对齐加载
7
8      float data2[4];
9      // __m128 vec2 = _mm_load_ps(data2); // 可能崩溃！未对齐
10     __m128 vec2 = _mm_loadu_ps(data2); // ✓ 使用未对齐加载（慢）

```

```

11
12 // AVX要求32字节对齐
13 alignas(32) float data3[8];
14 __m256 vec3 = _mm256_load_ps(data3); // ✓ 对齐加载
15
16 // 性能差异：
17 // _mm_load_ps (对齐) vs _mm_loadu_ps (未对齐)
18 // 快约2倍
19 }

```

6. 缓存行效率

```

1 // 现代CPU缓存行通常64字节
2 // 每次从内存加载数据到缓存时，以缓存行为单位
3
4 struct Data {
5     int x; // 4字节
6 };
7
8 // 未对齐到缓存行
9 Data arr[100]; // 可能跨越多个缓存行
10
11 // 对齐到缓存行
12 struct alignas(64) CacheAlignedData {
13     int x;
14     char padding[60];
15 };
16
17 CacheAlignedData arr2[100]; // 每个元素独占一个缓存行
18
19 // 多线程场景的false sharing
20 struct Counter {
21     int count1; // 缓存行1
22     int count2; // 同一缓存行
23 };
24
25 // 线程1修改count1 → 整个缓存行失效
26 // 线程2修改count2 → 需要重新加载缓存行
27 // → False Sharing，性能下降！
28
29 // 解决方案：对齐到独立缓存行
30 struct alignas(64) GoodCounter {
31     int count1;
32     char padding1[60];
33 };
34 GoodCounter counters[2]; // 每个counter独立缓存行

```

总结：内存对齐的好处

方面	未对齐	对齐	访问速度	慢（多次内存访问）	快（单次访问）	跨平台	可能崩溃
兼容所有平台	是	否	原子操作	不保证原子性	保证原子性	SIMD	不可用或慢
缓存利用	差	优	内存占用	少（无填充）	多（有填充）	高效	

权衡：对齐以空间换时间，绝大多数情况下是正确的选择。

1.3.2 内存对齐示例

```

1  #include <iostream>
2
3  // 1. 自然对齐
4  struct Natural {
5      char c;    // 1字节, 偏移0
6      // 3字节填充
7      int i;     // 4字节, 偏移4
8      short s;   // 2字节, 偏移8
9      // 2字节填充 (保证数组元素对齐)
10 };
11 // sizeof(Natural) = 12 (不是7)
12
13 // 2. 手动排列减少空间
14 struct Optimized {
15     int i;      // 4字节, 偏移0
16     short s;    // 2字节, 偏移4
17     char c;     // 1字节, 偏移6
18     // 1字节填充
19 };
20 // sizeof(Optimized) = 8
21
22 // 3. 查看对齐要求
23 std::cout << "int对齐: " << alignof(int) << "\n"; // 4
24 std::cout << "double对齐: " << alignof(double) << "\n"; // 8
25
26 // 4. 指定对齐 (C++11)
27 struct alignas(16) Aligned16 {
28     int x;
29 };
30 std::cout << alignof(Aligned16) << "\n"; // 16
31
32 // 5. 对齐分配
33 void* aligned_alloc_example() {
34     // C++17: aligned_alloc
35     void* ptr = std::aligned_alloc(64, 128); // 64字节对齐, 128字节大小
36     std::free(ptr);
37
38     // C++11: alignas + new
39     struct alignas(64) AlignedData {
40         char data[128];
41     };
42     auto* p = new AlignedData;
43     delete p;
44
45     return ptr;
46 }
47
48 // 6. SIMD要求对齐
49 #include <immintrin.h>
50
51 alignas(32) float data[8]; // AVX需要32字节对齐
52 __m256 vec = _mm256_load_ps(data); // 对齐加载

```

```

53
54 // 7. 缓存行对齐 (避免false sharing)
55 struct alignas(64) CacheLine { // 64字节 = 典型缓存行大小
56     int counter;
57     char padding[60]; // 填充到64字节
58 };
59
60 // 多线程场景
61 CacheLine counters[4]; // 每个counter在独立缓存行, 避免false sharing

```

内存对齐规则：

```

1 // 规则1：成员对齐
2 // 每个成员的偏移必须是其大小的倍数
3
4 // 规则2：结构体对齐
5 // 结构体大小必须是最大成员对齐的倍数
6
7 // 规则3：数组对齐
8 // 确保数组元素正确对齐
9
10 struct Example {
11     char c1; // 偏移0
12     // 7字节填充
13     double d; // 偏移8 (double要求8字节对齐)
14     char c2; // 偏移16
15     // 7字节填充
16 };
17 // sizeof(Example) = 24

```



1.4 Placement New

在指定内存位置构造对象。

```

1 #include <new>
2 #include <iostream>
3
4 // 1. 基本用法
5 char buffer[sizeof(int)];
6 int* p = new (buffer) int(42); // 在buffer中构造int
7 std::cout << *p << "\n";
8 p->~int(); // 手动调用析构函数 (不释放内存)
9
10 // 2. 数组
11 char arr_buffer[sizeof(int) * 10];
12 int* arr = new (arr_buffer) int[10]; // 构造数组
13 for (int i = 0; i < 10; ++i) {
14     arr[i].~int(); // 逐个析构
15 }
16
17 // 3. 对象池实现
18 template<typename T, size_t N>
19 class ObjectPool {
20     alignas(T) char storage[N * sizeof(T)];
21     bool used[N] = {};

```



```

22
23 public:
24     template<typename... Args>
25     T* allocate(Args&&... args) {
26         for (size_t i = 0; i < N; ++i) {
27             if (!used[i]) {
28                 used[i] = true;
29                 T* ptr = reinterpret_cast<T*>(&storage[i * sizeof(T)]);
30                 return new (ptr) T(std::forward<Args>(args)...); // placement new
31             }
32         }
33         return nullptr;
34     }
35
36     void deallocate(T* ptr) {
37         if (!ptr) return;
38
39         size_t index = (reinterpret_cast<char*>(ptr) - storage) / sizeof(T);
40         if (index < N && used[index]) {
41             ptr->~T(); // 调用析构
42             used[index] = false;
43         }
44     }
45 };
46
47 // 使用
48 ObjectPool<std::string, 10> pool;
49 std::string* s1 = pool.allocate("Hello");
50 std::string* s2 = pool.allocate("World");
51 pool.deallocate(s1);
52 pool.deallocate(s2);
53
54 // 4. 自定义容器
55 template<typename T>
56 class MyVector {
57     T* data;
58     size_t capacity;
59     size_t size;
60
61 public:
62     void push_back(const T& value) {
63         if (size == capacity) {
64             reserve(capacity * 2);
65         }
66         new (&data[size]) T(value); // placement new
67         ++size;
68     }
69
70     void pop_back() {
71         if (size > 0) {
72             data[--size].~T(); // 显式析构
73         }
74     }
75
76     void reserve(size_t new_cap) {

```

```

77     if (new_cap <= capacity) return;
78
79     T* new_data = static_cast<T*>(::operator new(new_cap * sizeof(T)));
80
81     // 移动/拷贝现有元素
82     for (size_t i = 0; i < size; ++i) {
83         new (&new_data[i]) T(std::move(data[i]));
84         data[i].~T();
85     }
86
87     ::operator delete(data);
88     data = new_data;
89     capacity = new_cap;
90 }
91
92 ~MyVector() {
93     for (size_t i = 0; i < size; ++i) {
94         data[i].~T();
95     }
96     ::operator delete(data);
97 }
98 };
99
100 // 5. 内存映射文件
101 class MappedObject {
102 public:
103     static void* operator new(size_t size) {
104         // 使用mmap分配内存
105         void* ptr = mmap(nullptr, size, PROT_READ | PROT_WRITE,
106             MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
107         return ptr;
108     }
109
110     static void operator delete(void* ptr, size_t size) {
111         munmap(ptr, size);
112     }
113 };

```

1.5 内存池

提高小对象分配效率，减少内存碎片。

```

1  // 1. 固定大小内存池
2  template<size_t BlockSize, size_t BlockCount>
3  class FixedMemoryPool {
4      struct FreeNode {
5          FreeNode* next;
6      };
7
8      alignas(std::max_align_t) char storage[BlockSize * BlockCount];
9      FreeNode* free_list;
10
11 public:
12     FixedMemoryPool() {
13         // 初始化空闲链表
14         free_list = reinterpret_cast<FreeNode*>(storage);

```




```

15     FreeNode* current = free_list;
16
17     for (size_t i = 0; i < BlockCount - 1; ++i) {
18         char* next_block = storage + (i + 1) * BlockSize;
19         current->next = reinterpret_cast<FreeNode*>(next_block);
20         current = current->next;
21     }
22     current->next = nullptr;
23 }
24
25 void* allocate() {
26     if (!free_list) return nullptr;
27
28     void* ptr = free_list;
29     free_list = free_list->next;
30     return ptr;
31 }
32
33 void deallocate(void* ptr) {
34     if (!ptr) return;
35
36     FreeNode* node = static_cast<FreeNode*>(ptr);
37     node->next = free_list;
38     free_list = node;
39 }
40 };
41
42 // 使用
43 FixedMemoryPool<32, 100> pool; // 100个32字节的块
44 void* p1 = pool.allocate();
45 void* p2 = pool.allocate();
46 pool.deallocate(p1);
47
48 // 2. 线程安全的内存池
49 template<typename T>
50 class ThreadSafePool {
51     struct Node {
52         alignas(T) char storage[sizeof(T)];
53         Node* next;
54     };
55
56     std::mutex mutex;
57     Node* free_list = nullptr;
58     std::vector<Node*> blocks; // 跟踪所有分配的块
59
60     static constexpr size_t BLOCK_SIZE = 1024;
61
62     void grow() {
63         Node* new_block = new Node[BLOCK_SIZE];
64         blocks.push_back(new_block);
65
66         for (size_t i = 0; i < BLOCK_SIZE; ++i) {
67             new_block[i].next = free_list;
68             free_list = &new_block[i];
69         }

```

```

70     }
71
72     public:
73         template<typename... Args>
74         T* allocate(Args&&... args) {
75             std::lock_guard<std::mutex> lock(mutex);
76
77             if (!free_list) {
78                 grow();
79             }
80
81             Node* node = free_list;
82             free_list = free_list->next;
83
84             T* ptr = reinterpret_cast<T*>(node->storage);
85             return new (ptr) T(std::forward<Args>(args)...);
86         }
87
88         void deallocate(T* ptr) {
89             if (!ptr) return;
90
91             ptr->~T();
92
93             std::lock_guard<std::mutex> lock(mutex);
94             Node* node = reinterpret_cast<Node*>(ptr);
95             node->next = free_list;
96             free_list = node;
97         }
98
99         ~ThreadSafePool() {
100             for (Node* block : blocks) {
101                 delete[] block;
102             }
103         }
104     };
105
106     // 3. STL兼容的分配器
107     template<typename T>
108     class PoolAllocator {
109     public:
110         ThreadSafePool<T>* pool;
111
112         using value_type = T;
113
114         PoolAllocator(ThreadSafePool<T>* p) : pool(p) {}
115
116         template<typename U>
117         PoolAllocator(const PoolAllocator<U>& other)
118             : pool(reinterpret_cast<ThreadSafePool<T>*>(other.pool)) {}
119
120         T* allocate(size_t n) {
121             if (n != 1) {
122                 return static_cast<T*>(::operator new(n * sizeof(T)));
123             }
124             return pool->allocate();

```

```

125     }
126
127     void deallocate(T* ptr, size_t n) {
128         if (n != 1) {
129             ::operator delete(ptr);
130         } else {
131             pool->deallocate(ptr);
132         }
133     }
134 };
135
136 // 使用自定义分配器
137 ThreadSafePool<int> int_pool;
138 std::vector<int, PoolAllocator<int>> vec(PoolAllocator<int>(&int_pool));

```

1.6 内存泄漏详解

1.6.1 什么是内存泄漏？

定义：程序动态分配的内存没有被正确释放，导致内存无法被重新使用。

```

1 // 典型的内存泄漏
2 void leak_example() {
3     int* p = new int(42);
4     // 忘记delete p;
5 } // p离开作用域，指针销毁，但内存未释放！
6
7 // 调用1000次后
8 for (int i = 0; i < 1000; ++i) {
9     leak_example(); // 泄漏1000个int的内存
10 }

```

内存泄漏 vs 悬垂指针：

```

1 // 内存泄漏：内存未释放，但无法访问
2 void memory_leak() {
3     int* p = new int(42);
4     p = nullptr; // 内存泄漏！原来的内存无法访问也无法释放
5 }
6
7 // 悬垂指针：内存已释放，但指针仍在被使用
8 void dangling_pointer() {
9     int* p = new int(42);
10    delete p;
11    *p = 100; // 危险！使用已释放的内存
12 }
13
14 // 双重释放
15 void double_delete() {
16     int* p = new int(42);
17     delete p;
18     delete p; // 未定义行为！
19 }

```

1.6.2 常见内存泄漏场景

1. 忘记释放

```

1  // 场景1：简单遗忘
2  void simple_leak() {
3      char* buffer = new char[1024];
4      // ... 使用buffer
5      // 忘记：delete[] buffer;
6  }
7
8  // 场景2：提前返回
9  bool process_data(const std::string& filename) {
10     char* buffer = new char[1024];
11
12     if (filename.empty()) {
13         return false; // 泄漏！忘记释放buffer
14     }
15
16     // ... 处理数据
17     delete[] buffer;
18     return true;
19 }
20
21 // 场景3：异常抛出
22 void exception_leak() {
23     int* data = new int[1000];
24
25     // 如果这里抛出异常，data不会被释放
26     process(data); // 可能抛出异常
27
28     delete[] data; // 异常发生时不会执行
29 }

```

2. 容器中存储裸指针

```

1  // 错误：容器中存储裸指针
2  void container_leak() {
3      std::vector<int*> vec;
4
5      for (int i = 0; i < 10; ++i) {
6          vec.push_back(new int(i)); // 分配内存
7      }
8
9      // vec销毁时，只删除指针，不删除指向的内存
10 } // 泄漏10个int！
11
12 // 正确方式1：手动释放
13 void correct1() {
14     std::vector<int*> vec;
15     for (int i = 0; i < 10; ++i) {
16         vec.push_back(new int(i));
17     }
18
19     // 释放所有内存
20     for (int* p : vec) {
21         delete p;
22     }

```

```

23     vec.clear();
24 }
25
26 // 正确方式2：使用智能指针
27 void correct2() {
28     std::vector<std::unique_ptr<int>> vec;
29     for (int i = 0; i < 10; ++i) {
30         vec.push_back(std::make_unique<int>(i));
31     }
32     // 自动释放，无泄漏
33 }

```

3. 循环引用

```

1  // shared_ptr的循环引用
2  class Node {
3  public:
4      std::shared_ptr<Node> next;
5      std::shared_ptr<Node> prev; // 问题所在
6      ~Node() { std::cout << "~Node\n"; }
7  };
8
9  void circular_reference_leak() {
10     auto node1 = std::make_shared<Node>();
11     auto node2 = std::make_shared<Node>();
12
13     node1->next = node2; // node2引用计数 = 2
14     node2->prev = node1; // node1引用计数 = 2
15
16     // 函数结束时：
17     // node1引用计数 = 2 - 1 = 1 (还有node2->prev指向)
18     // node2引用计数 = 2 - 1 = 1 (还有node1->next指向)
19     // 两个对象都不会被删除！
20 } // 泄漏！析构函数不会被调用
21
22 // 正确方式：使用weak_ptr
23 class GoodNode {
24 public:
25     std::shared_ptr<GoodNode> next;
26     std::weak_ptr<GoodNode> prev; // 弱引用，打破循环
27     ~GoodNode() { std::cout << "~GoodNode\n"; }
28 };
29
30 void no_leak() {
31     auto node1 = std::make_shared<GoodNode>();
32     auto node2 = std::make_shared<GoodNode>();
33
34     node1->next = node2; // node2引用计数 = 2
35     node2->prev = node1; // node1引用计数 = 1 (weak_ptr不增加)
36
37     // 函数结束时：
38     // node1引用计数 = 1 - 1 = 0 → 删除
39     // node2引用计数 = 2 - 1 - 1 = 0 → 删除
40 } // 正确释放

```

4. this 指针与 shared_ptr

```

1  // 错误：从this创建shared_ptr
2  class Widget {
3  public:
4      std::shared_ptr<Widget> getPtr() {
5          return std::shared_ptr<Widget>(this); // 危险！
6      }
7  };
8
9  void dangerous_this() {
10     auto w1 = std::make_shared<Widget>();
11     auto w2 = w1->getPtr(); // 创建了第二个独立的shared_ptr！
12
13     // w1和w2都管理同一个对象，但引用计数分别独立
14     // 当w1或w2之一为0时，会删除对象
15     // 当另一个为0时，会再次删除 → 双重删除！
16 }
17
18 // 正确方式：enable_shared_from_this
19 class SafeWidget : public std::enable_shared_from_this<SafeWidget> {
20 public:
21     std::shared_ptr<SafeWidget> getPtr() {
22         return shared_from_this(); // 正确！共享引用计数
23     }
24 };
25
26 void safe_this() {
27     auto w1 = std::make_shared<SafeWidget>();
28     auto w2 = w1->getPtr(); // w1和w2共享引用计数
29     // 正确释放
30 }

```

5. 资源管理类忘记释放

```

1  // 错误：资源管理类没有正确释放
2  class ResourceManager {
3      int* data;
4  public:
5      ResourceManager() : data(new int[1000]) {}
6      // 忘记写析构函数！
7  };
8
9  void resource_leak() {
10     ResourceManager rm;
11 } // 泄漏！data未释放
12
13 // 正确方式：遵循Rule of Three/Five
14 class GoodResourceManager {
15     int* data;
16 public:
17     GoodResourceManager() : data(new int[1000]) {}
18
19     ~GoodResourceManager() {
20         delete[] data; // 析构时释放

```

```

21     }
22
23     // 禁止拷贝，或实现深拷贝
24     GoodResourceManager(const GoodResourceManager&) = delete;
25     GoodResourceManager& operator=(const GoodResourceManager&) = delete;
26
27     // 可选：实现移动
28     GoodResourceManager(GoodResourceManager&& other) noexcept
29         : data(other.data) {
30         other.data = nullptr;
31     }
32
33     GoodResourceManager& operator=(GoodResourceManager&& other) noexcept {
34         if (this != &other) {
35             delete[] data;
36             data = other.data;
37             other.data = nullptr;
38         }
39         return *this;
40     }
41 };

```

6. 多线程中的泄漏

```

1  #include <thread>
2  #include <mutex>
3
4  // 场景：线程局部存储泄漏
5  class ThreadManager {
6      std::map<std::thread::id, int*> thread_data;
7      std::mutex mtx;
8
9  public:
10     void register_thread() {
11         std::lock_guard<std::mutex> lock(mtx);
12         thread_data[std::this_thread::get_id()] = new int(0);
13     }
14
15     // 问题：线程结束时没有清理
16     ~ThreadManager() {
17         // 应该释放所有thread_data中的指针
18         for (auto& [id, ptr] : thread_data) {
19             delete ptr;
20         }
21     }
22 };

```

1.6.3 内存泄漏的危害

```

1  // 1. 内存耗尽
2  void memory_exhaustion() {
3      std::vector<int*> leaks;
4      while (true) {
5          leaks.push_back(new int[1024 * 1024]); // 每次4MB
6          // 最终：系统内存耗尽，程序崩溃或被系统杀死
7      }

```

```

8  }
9
10 // 2. 性能下降
11 // - 可用内存减少 → 频繁换页 (swapping)
12 // - 系统响应变慢
13 // - GC语言: GC压力增大
14
15 // 3. 长期运行的服务器
16 void server_leak() {
17     // 假设服务器每秒泄漏1KB
18     // 1天 = 86400秒 → 84MB
19     // 1周 → 588MB
20     // 1月 → 2.5GB
21     // 最终服务器崩溃
22 }
23
24 // 4. 难以调试
25 // - 症状不明显: 内存缓慢增长
26 // - 难以重现: 可能需要运行数天才出现
27 // - 难以定位: 泄漏点可能在代码的任何地方

```

1.6.4 如何防止内存泄漏

1. 使用 RAII (Resource Acquisition Is Initialization)

```

1  // 原则: 资源获取即初始化, 资源释放通过析构函数自动完成
2
3  // 差的做法
4  void bad_raii() {
5      FILE* fp = fopen("data.txt", "r");
6      if (!fp) return;
7
8      // ... 处理文件
9      if (error_condition) {
10         return; // 泄漏! 忘记fclose
11     }
12
13     fclose(fp);
14 }
15
16 // 好的做法: RAII
17 class FileHandle {
18     FILE* fp;
19 public:
20     FileHandle(const char* filename, const char* mode)
21         : fp(fopen(filename, mode)) {
22         if (!fp) throw std::runtime_error("Cannot open file");
23     }
24
25     ~FileHandle() {
26         if (fp) fclose(fp);
27     }
28
29     FILE* get() { return fp; }
30

```



```

31 // 禁止拷贝
32 FileHandle(const FileHandle&) = delete;
33 FileHandle& operator=(const FileHandle&) = delete;
34 };
35
36 void good_raii() {
37     FileHandle file("data.txt", "r");
38     // ... 使用file.get()
39     // 任何情况下（正常返回、异常、提前返回），文件都会自动关闭
40 }
41
42 // 更好：使用现有的RAII类
43 void best_raii() {
44     auto deleter = [](FILE* fp) { if(fp) fclose(fp); };
45     std::unique_ptr<FILE, decltype(deleter)> file(
46         fopen("data.txt", "r"), deleter
47     );
48     // 自动管理
49 }

```

2. 优先使用智能指针

```

1 // 规则：永远不要使用裸指针管理内存
2
3 // ✗ 差
4 void use_raw_pointer() {
5     Widget* w = new Widget();
6     // ...
7     delete w; // 容易忘记，异常时不会执行
8 }
9
10 // ✓ 好：unique_ptr（独占所有权）
11 void use_unique_ptr() {
12     auto w = std::make_unique<Widget>();
13     // 自动释放，异常安全
14 }
15
16 // ✓ 好：shared_ptr（共享所有权）
17 void use_shared_ptr() {
18     auto w = std::make_shared<Widget>();
19     // 引用计数管理，自动释放
20 }
21
22 // ✓ 好：容器中存储智能指针
23 void use_container() {
24     std::vector<std::unique_ptr<Widget>> widgets;
25     widgets.push_back(std::make_unique<Widget>());
26     // vector销毁时，自动释放所有Widget
27 }

```

3. 避免循环引用

```

1 // 设计数据结构时考虑所有权
2
3 // 树结构：父节点拥有子节点

```

```

4  class TreeNode {
5      std::vector<std::unique_ptr<TreeNode>> children; // 拥有
6      TreeNode* parent; // 不拥有，裸指针
7
8  public:
9      void addChild(std::unique_ptr<TreeNode> child) {
10         child->parent = this;
11         children.push_back(std::move(child));
12     }
13 };
14
15 // 双向链表：使用weak_ptr
16 class ListNode {
17 public:
18     std::shared_ptr<ListNode> next; // 强引用
19     std::weak_ptr<ListNode> prev; // 弱引用，打破循环
20 };
21
22 // 图结构：明确所有权
23 class Graph {
24     std::vector<std::unique_ptr<Node>> nodes; // Graph拥有所有节点
25
26     // 边使用裸指针或weak_ptr（不拥有）
27     struct Edge {
28         Node* from;
29         Node* to;
30     };
31     std::vector<Edge> edges;
32 };

```

4. 容器和异常安全

```

1  // 使用容器自动管理资源
2  void exception_safe() {
3      std::vector<std::unique_ptr<Widget>> widgets;
4
5      try {
6          for (int i = 0; i < 10; ++i) {
7              auto w = std::make_unique<Widget>();
8              w->initialize(); // 可能抛出异常
9              widgets.push_back(std::move(w));
10         }
11     } catch (...) {
12         // widgets会自动释放所有已创建的Widget
13         throw;
14     }
15     // 正常情况也自动释放
16 }
17
18 // 事务性操作
19 void transactional_resource() {
20     std::vector<std::unique_ptr<Resource>> resources;
21
22     try {
23         // 第1步：分配资源

```

 C++

```

24     resources.push_back(std::make_unique<Resource>(1));
25     resources.push_back(std::make_unique<Resource>(2));
26
27     // 第2步：可能失败的操作
28     if (!validate()) {
29         throw std::runtime_error("Validation failed");
30     }
31
32     // 第3步：提交
33     for (auto& r : resources) {
34         r->commit();
35     }
36 } catch (...) {
37     // 自动回滚：resources析构会清理所有资源
38     throw;
39 }
40 }

```

5. 遵循 Rule of Three/Five/Zero

```

1  // Rule of Zero：不管理资源，让编译器生成默认函数
2  class RuleOfZero {
3      std::string name;
4      std::vector<int> data;
5      std::unique_ptr<Widget> widget;
6      // 不需要自定义析构、拷贝、移动 - 编译器生成的版本正确
7  };
8
9  // Rule of Three：如果需要自定义析构、拷贝构造或拷贝赋值之一，
10 // 则三者都需要定义（C++11之前）
11 class RuleOfThree {
12     int* data;
13     size_t size;
14
15 public:
16     // 析构函数
17     ~RuleOfThree() {
18         delete[] data;
19     }
20
21     // 拷贝构造函数
22     RuleOfThree(const RuleOfThree& other)
23         : size(other.size), data(new int[size]) {
24         std::copy(other.data, other.data + size, data);
25     }
26
27     // 拷贝赋值运算符
28     RuleOfThree& operator=(const RuleOfThree& other) {
29         if (this != &other) {
30             delete[] data;
31             size = other.size;
32             data = new int[size];
33             std::copy(other.data, other.data + size, data);
34         }
35         return *this;

```

```

36     }
37 };
38
39 // Rule of Five: C++11, 加上移动构造和移动赋值
40 class RuleOfFive {
41     int* data;
42     size_t size;
43
44 public:
45     ~RuleOfFive() { delete[] data; }
46
47     RuleOfFive(const RuleOfFive& other); // 拷贝构造
48     RuleOfFive& operator=(const RuleOfFive& other); // 拷贝赋值
49
50     // 移动构造函数
51     RuleOfFive(RuleOfFive&& other) noexcept
52         : data(other.data), size(other.size) {
53         other.data = nullptr;
54         other.size = 0;
55     }
56
57     // 移动赋值运算符
58     RuleOfFive& operator=(RuleOfFive&& other) noexcept {
59         if (this != &other) {
60             delete[] data;
61             data = other.data;
62             size = other.size;
63             other.data = nullptr;
64             other.size = 0;
65         }
66         return *this;
67     }
68 };

```

6. 工厂函数返回智能指针

```

1 // 返回智能指针, 明确所有权转移
2 std::unique_ptr<Widget> createWidget(int id) {
3     auto w = std::make_unique<Widget>();
4     w->setId(id);
5     return w; // 移动, 无拷贝
6 }
7
8 void use_factory() {
9     auto widget = createWidget(42);
10    // 自动管理
11 }
12
13 // 多态工厂
14 std::unique_ptr<Base> createObject(const std::string& type) {
15     if (type == "A") return std::make_unique<DerivedA>();
16     if (type == "B") return std::make_unique<DerivedB>();
17     return nullptr;
18 }

```

7. 使用标准容器

```

1 // x 差：手动管理数组
2 void manual_array() {
3     int* arr = new int[1000];
4     // ... 使用
5     delete[] arr; // 容易忘记
6 }
7
8 // ✓ 好：使用vector
9 void use_vector() {
10    std::vector<int> arr(1000);
11    // 自动管理
12 }
13
14 // x 差：手动管理字符串
15 void manual_string() {
16     char* str = new char[100];
17     strcpy(str, "Hello");
18     // ...
19     delete[] str;
20 }
21
22 // ✓ 好：使用string
23 void use_string() {
24     std::string str = "Hello";
25     // 自动管理
26 }

```

8. 注意第三方库和 API

```

1 // 了解所有权语义
2 void third_party_api() {
3     // 示例1：API返回的指针需要用户释放
4     char* data = some_library_alloc(); // 分配内存
5     // ... 使用
6     some_library_free(data); // 必须用对应的free函数
7
8     // 更好：立即封装到智能指针
9     std::unique_ptr<char, decltype(&some_library_free)>
10        smart_data(some_library_alloc(), some_library_free);
11
12     // 示例2：API接管所有权
13     Widget* w = new Widget();
14     api_takes_ownership(w); // API会负责释放
15     // 不要再delete w!
16
17     // 示例3：API不接管所有权
18     Widget* w2 = new Widget();
19     api_borrows(w2); // API只是借用
20     delete w2; // 仍需手动释放
21 }

```

1.6.5 内存泄漏检测工具

1. Valgrind (Linux)

```

1  # 编译时保留调试信息
2  g++ -g -O0 program.cpp -o program
3
4  # 运行Valgrind
5  valgrind --leak-check=full \
6           --show-leak-kinds=all \
7           --track-origins=yes \
8           --verbose \
9           --log-file=valgrind-out.txt \
10          ./program
11
12 # 输出示例：
13 # ==12345== LEAK SUMMARY:
14 # ==12345==    definitely lost: 40 bytes in 1 blocks
15 # ==12345==    indirectly lost: 0 bytes in 0 blocks
16 # ==12345==    possibly lost: 0 bytes in 0 blocks

```

2. AddressSanitizer (ASan)

```

1  # 编译时启用ASan
2  g++ -fsanitize=address -fno-omit-frame-pointer -g program.cpp -o program
3
4  # 运行
5  ./program
6
7  # 自动检测：
8  # - 内存泄漏
9  # - 使用已释放的内存
10 # - 缓冲区溢出
11 # - 双重释放

```

3. 自定义内存跟踪

```

1  // 1. 重载全局new/delete进行跟踪
2  #ifdef DEBUG_MEMORY
3
4  #include <map>
5  #include <mutex>
6
7  struct AllocationInfo {
8      size_t size;
9      const char* file;
10     int line;
11 };
12
13 std::map<void*, AllocationInfo> allocations;
14 std::mutex alloc_mutex;
15
16 void* operator new(size_t size, const char* file, int line) {
17     void* ptr = malloc(size);
18     std::lock_guard<std::mutex> lock(alloc_mutex);
19     allocations[ptr] = {size, file, line};
20     return ptr;
21 }
22
23 void operator delete(void* ptr) noexcept {

```

```

24     std::lock_guard<std::mutex> lock(alloc_mutex);
25     allocations.erase(ptr);
26     free(ptr);
27 }
28
29 #define new new(__FILE__, __LINE__)
30
31 // 检测泄漏
32 void check_leaks() {
33     std::lock_guard<std::mutex> lock(alloc_mutex);
34     for (const auto& [ptr, info] : allocations) {
35         std::cerr << "Leak: " << info.size << " bytes at "
36                 << info.file << ":" << info.line << "\n";
37     }
38 }
39
40 #endif
41
42 // 2. RAII封装检测
43 class MemoryTracker {
44     size_t allocations = 0;
45     size_t deallocations = 0;
46
47 public:
48     ~MemoryTracker() {
49         if (allocations != deallocations) {
50             std::cerr << "Memory leak detected!\n";
51             std::cerr << "Allocations: " << allocations << "\n";
52             std::cerr << "Deallocations: " << deallocations << "\n";
53         }
54     }
55
56     void on_allocate() { ++allocations; }
57     void on_deallocate() { ++deallocations; }
58 };
59
60 // 3. 使用工具
61 // Valgrind: valgrind --leak-check=full ./program
62 // AddressSanitizer: g++ -fsanitize=address program.cpp
63 // 内存分析器: heaptrack, massif

```

4. Visual Studio 内存泄漏检测

```

1 // Windows平台
2 #ifdef _DEBUG
3 #define _CRTDBG_MAP_ALLOC
4 #include <stdlib.h>
5 #include <crtdbg.h>
6
7 int main() {
8     _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
9
10    // 程序代码
11    int* leak = new int(42); // 故意泄漏
12

```

```

13     // 程序结束时自动报告泄漏
14     return 0;
15 }
16 #endif

```

5. 其他工具

```

1  # Dr. Memory (跨平台)
2  drmemory -- ./program
3
4  # Heaptrack (Linux, 可视化)
5  heaptrack ./program
6  heaptrack_gui heaptrack.program.gz
7
8  # Massif (Valgrind的一部分, 分析堆使用)
9  valgrind --tool=massif ./program
10 ms_print massif.out.12345
11
12 # GPerfTools
13 LD_PRELOAD=/usr/lib/libtcmalloc.so HEAPCHECK=normal ./program

```

1.6.6 内存泄漏防范检查清单

代码审查清单：

```

1  // ✓ 检查项
2  // [ ] 所有new都有对应的delete
3  // [ ] 所有new[]都有对应的delete[]
4  // [ ] 异常安全：异常抛出时资源能正确释放
5  // [ ] 提前返回：所有return路径都释放了资源
6  // [ ] 容器存储：使用智能指针而非裸指针
7  // [ ] 循环引用：使用weak_ptr打破循环
8  // [ ] 资源类：实现或禁用拷贝/移动操作
9  // [ ] 第三方API：了解所有权语义
10 // [ ] 多线程：线程局部存储正确清理
11 // [ ] 单例模式：考虑是否需要释放

```

代码模式：

```

1  // 1. 每个new都应该考虑：谁负责delete?
2  void allocation_ownership() {
3      // 问题：谁负责释放？
4      Widget* w = new Widget();
5
6      // 答案1：我负责
7      std::unique_ptr<Widget> w(new Widget());
8
9      // 答案2：调用者负责
10     std::unique_ptr<Widget> create() {
11         return std::make_unique<Widget>();
12     }
13
14     // 答案3：共享所有权
15     std::shared_ptr<Widget> w = std::make_shared<Widget>();
16 }
17

```



```

18 // 2. 容器中的资源
19 void container_resources() {
20     // x 需要手动管理
21     std::vector<int*> bad;
22
23     // ✓ 自动管理
24     std::vector<int> good1;
25     std::vector<std::unique_ptr<int>> good2;
26 }
27
28 // 3. 异常安全的资源获取
29 void exception_safe_acquisition() {
30     // x 不安全
31     Resource* r1 = new Resource();
32     Resource* r2 = new Resource(); // 如果这里抛出异常，r1泄漏
33
34     // ✓ 安全
35     auto r1 = std::make_unique<Resource>();
36     auto r2 = std::make_unique<Resource>(); // 安全！
37 }
38
39 // 4. 成对操作
40 void paired_operations() {
41     // 资源获取和释放应该在同一抽象层次
42
43     // x 差：获取和释放分离
44     class Bad {
45     public:
46         int* data;
47         void allocate() { data = new int[100]; }
48         void deallocate() { delete[] data; } // 容易忘记调用
49     };
50
51     // ✓ 好：通过构造和析构配对
52     class Good {
53     public:
54         std::vector<int> data;
55         Good() : data(100) {} // 自动管理
56     };
57 }
58
59 // 5. 明确的所有权转移
60 void ownership_transfer() {
61     // x 不清晰
62     Widget* transfer_bad() {
63         Widget* w = new Widget();
64         return w; // 谁负责删除？不清楚！
65     }
66
67     // ✓ 清晰
68     std::unique_ptr<Widget> transfer_good() {
69         auto w = std::make_unique<Widget>();
70         return w; // 明确：调用者获得所有权
71     }

```

```
72 }
```

常见陷阱：

```
1 // 陷阱1：忘记虚析构函数
2 class Base {
3 public:
4     // 缺少virtual析构函数！
5     ~Base() { /* ... */ }
6 };
7
8 class Derived : public Base {
9     int* data;
10 public:
11     Derived() : data(new int[1000]) {}
12     ~Derived() { delete[] data; }
13 };
14
15 void polymorphic_delete() {
16     Base* b = new Derived();
17     delete b; // 只调用Base::~~Base()，Derived::~~Derived()不会调用！
18     // data泄漏！
19 }
20
21 // 正确：虚析构
22 class GoodBase {
23 public:
24     virtual ~GoodBase() = default;
25 };
26
27 // 陷阱2：拷贝资源类
28 class ResourceHolder {
29     int* data;
30 public:
31     ResourceHolder() : data(new int[100]) {}
32     ~ResourceHolder() { delete[] data; }
33     // 没有禁用拷贝！
34 };
35
36 void copy_trap() {
37     ResourceHolder r1;
38     ResourceHolder r2 = r1; // 浅拷贝！r1.data == r2.data
39 } // 双重释放！
40
41 // 正确：禁用或实现深拷贝
42 class SafeResourceHolder {
43     int* data;
44 public:
45     SafeResourceHolder() : data(new int[100]) {}
46     ~SafeResourceHolder() { delete[] data; }
47
48     // 方式1：禁用拷贝
49     SafeResourceHolder(const SafeResourceHolder&) = delete;
50     SafeResourceHolder& operator=(const SafeResourceHolder&) = delete;
51 }
```

```

52     // 方式2：或使用智能指针
53     // std::unique_ptr<int[]> data;
54 };
55
56 // 陷阱3：返回局部变量的指针/引用
57 int* local_pointer() {
58     int x = 42;
59     return &x; // 危险！返回局部变量地址
60 }
61
62 std::string& local_reference() {
63     std::string s = "Hello";
64     return s; // 危险！返回局部变量引用
65 }
66
67 // 陷阱4：单例模式的内存
68 class Singleton {
69     static Singleton* instance;
70     Singleton() {}
71 public:
72     static Singleton* getInstance() {
73         if (!instance) {
74             instance = new Singleton(); // 永不释放
75         }
76         return instance;
77     }
78 };
79
80 // 更好的方式
81 class GoodSingleton {
82 public:
83     static GoodSingleton& getInstance() {
84         static GoodSingleton instance; // 静态局部变量，程序结束时自动析构
85         return instance;
86     }
87 private:
88     GoodSingleton() {}
89     GoodSingleton(const GoodSingleton&) = delete;
90     GoodSingleton& operator=(const GoodSingleton&) = delete;
91 };

```

内存泄漏调试流程：

```

1 // 步骤1：重现问题
2 // - 在测试环境中运行程序
3 // - 监控内存使用（top, htop, Task Manager等）
4 // - 确认内存持续增长
5
6 // 步骤2：定位泄漏点
7 // - 使用Valgrind或ASan运行程序
8 // - 分析输出，找到泄漏的分配点
9 // - 查看调用栈
10
11 // 步骤3：分析原因

```

```

12 // - 是否忘记delete?
13 // - 是否有循环引用?
14 // - 异常安全问题?
15 // - 所有权不明确?
16
17 // 步骤4: 修复
18 // - 使用智能指针
19 // - 添加RAII封装
20 // - 打破循环引用
21 // - 确保异常安全
22
23 // 步骤5: 验证
24 // - 重新运行程序和检测工具
25 // - 确认泄漏消失
26 // - 添加单元测试防止回归
27
28 // 示例: 调试循环引用
29 void debug_circular_reference() {
30     // 问题代码
31     auto node1 = std::make_shared<Node>();
32     auto node2 = std::make_shared<Node>();
33     node1->next = node2;
34     node2->prev = node1; // 循环引用
35
36     // 调试: 检查引用计数
37     std::cout << "node1 count: " << node1.use_count() << "\n"; // 2
38     std::cout << "node2 count: " << node2.use_count() << "\n"; // 2
39
40     // 修复: 使用weak_ptr
41     // 在Node类中: std::weak_ptr<Node> prev;
42 }

```

1.7 内存管理最佳实践

```

1 // 1. 优先使用栈
2 void good() {
3     std::string s = "Hello"; // 栈上, 自动管理
4     std::vector<int> vec(100); // 内部堆分配, 但vec本身在栈上
5 }
6
7 void bad() {
8     std::string* s = new std::string("Hello"); // 不必要的堆分配
9     delete s;
10 }
11
12 // 2. 使用智能指针, 避免裸指针
13 // ✓ 好
14 std::unique_ptr<Widget> widget = std::make_unique<Widget>();
15
16 // ✗ 差
17 Widget* widget = new Widget();
18 delete widget;
19
20 // 3. 避免过早优化

```



```

21 // 先写清晰的代码，必要时才优化
22 std::vector<int> data(1000); // 简单清晰
23
24 // 不要过早使用内存池等复杂技术
25
26 // 4. 注意对象生命周期
27 void dangerous() {
28     std::string* ptr;
29     {
30         std::string temp = "Hello";
31         ptr = &temp; // 危险！temp即将销毁
32     }
33     // *ptr; // 悬垂指针！
34 }
35
36 // 5. 移动而非拷贝
37 std::vector<int> create_large_vector() {
38     std::vector<int> vec(1000000);
39     return vec; // 移动，不拷贝
40 }
41
42 auto v = create_large_vector(); // 高效
43
44 // 6. 就地构造
45 std::vector<std::string> vec;
46 vec.emplace_back("Hello"); // 直接在vector中构造
47 // 而非：vec.push_back(std::string("Hello")); // 构造临时对象+移动
48
49 // 7. 预分配容量
50 std::vector<int> vec;
51 vec.reserve(1000); // 预分配，避免多次重新分配
52 for (int i = 0; i < 1000; ++i) {
53     vec.push_back(i);
54 }
55
56 // 8. 小对象优化(SSO)意识
57 std::string short_str = "Hi"; // 可能不分配堆内存(SSO)
58 std::string long_str = "Very long string..."; // 分配堆内存
59
60 // 9. 自定义删除器
61 auto deleter = [](FILE* fp) { if(fp) fclose(fp); };
62 std::unique_ptr<FILE, decltype(deleter)> file(fopen("data.txt", "r"), deleter);
63
64 // 10. 避免内存碎片
65 // - 使用内存池管理小对象
66 // - 预分配连续内存
67 // - 避免频繁的分配/释放

```

1.8 总结

内存管理关键原则：

1. **RAII**：资源获取即初始化，利用对象生命周期管理资源
2. 智能指针：优先使用智能指针，避免手动管理
3. 栈优先：能用栈就用栈，性能最优

4. 移动语义：利用移动避免不必要的拷贝
5. 就地构造：使用 `emplace` 系列函数
6. 预分配：已知大小时预分配容量
7. 内存对齐：性能敏感代码注意对齐
8. 避免泄漏：使用工具检测，遵循最佳实践

智能指针选择：

- 独占所有权 → `unique_ptr`
- 共享所有权 → `shared_ptr`
- 观察者模式 → `weak_ptr`
- 工厂函数 → `unique_ptr`（可转换为 `shared_ptr`）

性能优化：

- 内存池：小对象频繁分配
- 对象池：复用对象
- 自定义分配器：特定场景
- `placement new`：精确控制内存位置