Dynamic Memory Allocation

程羽

2019 年 12 月 5 日

Overview

动态内存分配器

- 在运行时，程序利用动态内存分配器维护虚拟内存中的堆结构.
- 内核为每个进程维护一个指向堆顶部的指针 brk.
- 分配器将堆视为一系列块的集合，每个块是一个连续的虚拟内存片，分为已分配的和空闲的

malloc package

- C 标准库提供了 malloc 显式分配器.
- 如果创建成功，malloc 函数返回一个指向分配位置的指针.
- 如果创建失败，返回 NULL 并设置 errno
- 其他函数:
  - calloc：将分配的内存空间初始化为 0.
  - realloc: 调整之前分配块的 size.
  - sbrk: 改变 brk 指针来调整堆的大小.

动态内存分配器

- 为什么使用动态内存分配?
  - 实际运行到程序某处时，才知道某些数据结构的大小.
  - 编写大型程序的时候使用固定大小的数据结构维护困难.

- 分配器的要求
  - 处理任意请求序列.
  - 立即响应. 不允许缓冲请求或重新排列请求.
  - 对齐要求.
  - 不能修改已经分配的块.

动态内存分配器

分配器的目标

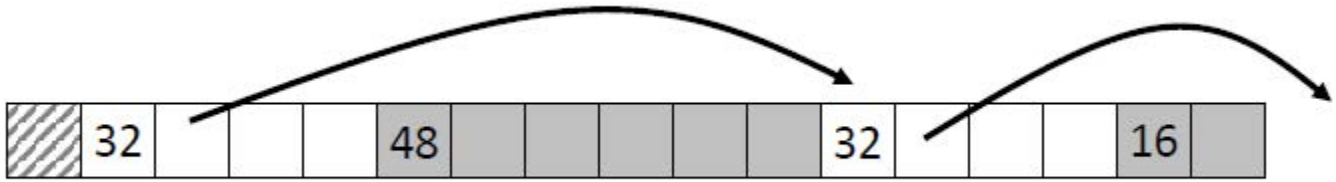
- 最大化吞吐率
  - 吞吐率：单位时间内完成的请求数 (分配操作或释放操作).
- 最大化内存利用率 (最小化开销)
  - P_k: 完成请求 R_k 后已分配的块的有效载荷之和.
  - H_k: 当前堆的大小（假设堆只有在使用 sbrk 函数的时候大小才会发生变化）.
  - U_k(峰值利用率):$U_k = \frac{max_{i \leq k} P_i}{H_k}$
  - O_k(开销):$O_k = \frac{H_k}{max_{i \leq k} P_i} - 1$

碎片

- 内部碎片
  - 已分配块大小大于负载大小.
  - 原因：对齐方式，块规定的强制最小大小等.
  - 只和之前的命令有关，量化和处理方便.
- 外部碎片
  - 因为不合理的排列方式导致空间不足.
  - 与以前和将来的请求模式都有关，难以预测和处理.
  - 要求分配器尽量维持尽可能大的空闲块.

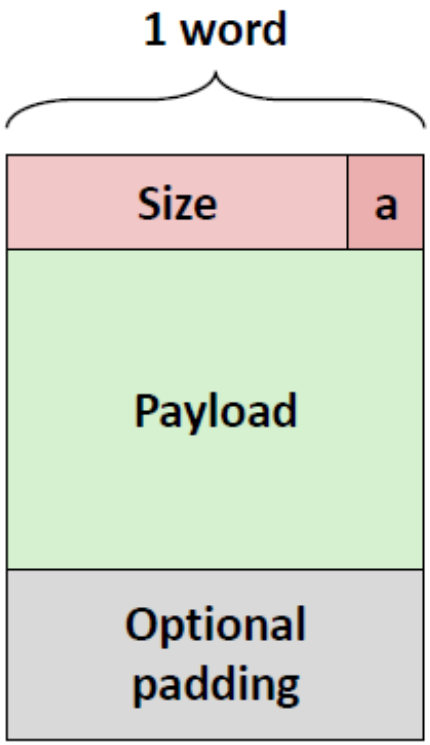
如何检索空闲块

- 方法一：隐式空闲块链表
  - 一个字记录块的大小
- 方法二：显式空闲块链表
  - 一个字记录块大小，一个块记录是指向下一个空闲块的指针



- 方法三：分离的空闲链表
  - 按照块的大小组织链表，每个链表中的块大小大致相当
- 方法四：按照块大小排序的链表

隐式空闲链表



- 头部
  - 记录信息：是否被分配，块的大小
  - 若要求是双字 (8 字节) 对齐的，最后 3 位是冗余，可以用来记录其他信息
- 有效载荷
- 填充

可以通过块大小检索到下一个块的位置.

隐式空闲链表

放置请求块：

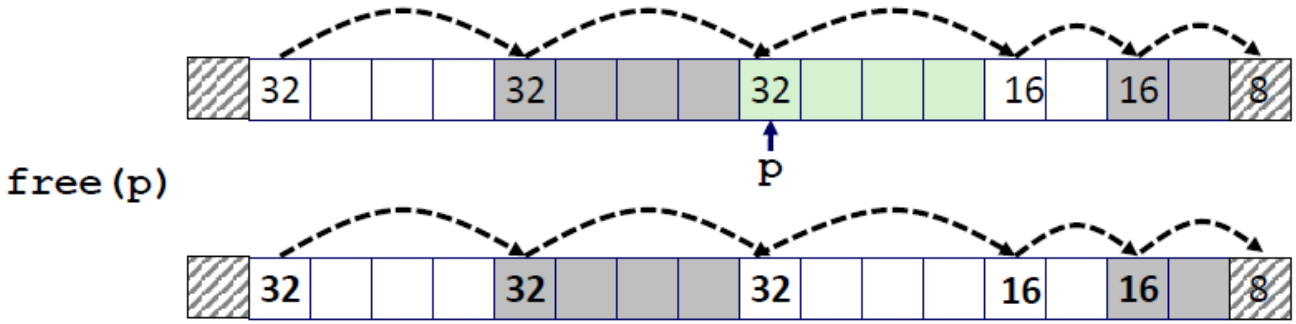
- 首次适配 (first fit)：从头搜索链表，选择第一个合适的空闲块
  - 缺点：每次分配的块都集中在开头部分，查找时间随分配的块数线性增长。
- 下一次适配 (next fit)：每次从上一次查询结束的地方开始搜索.
- 最佳适配 (best fit)：检查每个空闲块，选择最合适的空闲块.
  - 大多数情况下有较好结果，但仍然是贪心算法，不能保证最优.
  - 查找时间过长.

隐式空闲链表

合并空闲块：当释放一个和空闲块相邻的块时，会出现假碎片

- 何时进行合并：
  - 立即合并：可能会产生大量不必要的分割和合并操作。
  - 推迟合并

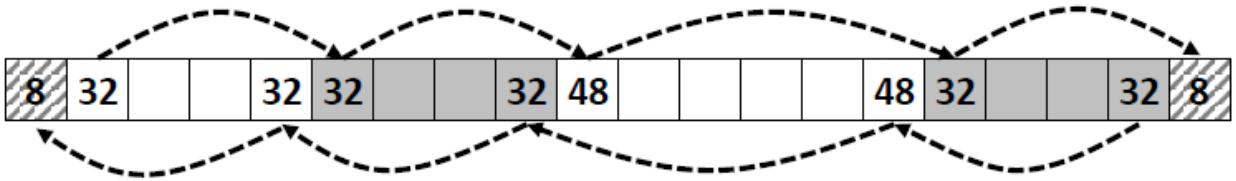
    实际情况下会选择某种形式的推迟合并。



隐式空闲链表

free(p):

- 合并后一个块：
  - 通过头部的记录寻找下一个块的头部，若是空闲块就将块大小赋为两个块大小之和.
- 合并前一个块：
  - 建立双向链表，需要脚部 (头部的一个副本) 来便于后一个块向前检索



Malloc 的代码实现

```
1  const size_t dsize = 2*sizeof(word_t);
2  //header和footer的大小之和
3  void *mm_malloc(size_t size)
4  {
5      size_t asize = round_up(size + dsize, dsize);
6      block_t *block = find_fit(asize);
7      if (block == NULL)
8          return NULL;
9      size_t block_size = get_size(block);
10     write_header(block, block_size, true);
11     write_footer(block, block_size, true);
12     split_block(block, asize);
13     return header_to_payload(block);
14 }
```

$$round\_up(n, m) = m\lfloor \frac{n+m-1}{m} \rfloor$$

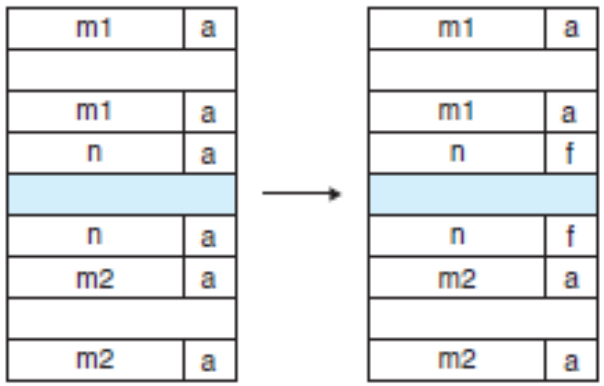
得到的 asize 是考虑了向 dsize 对齐后的最小值，即为所需的空间.

Free 的代码实现

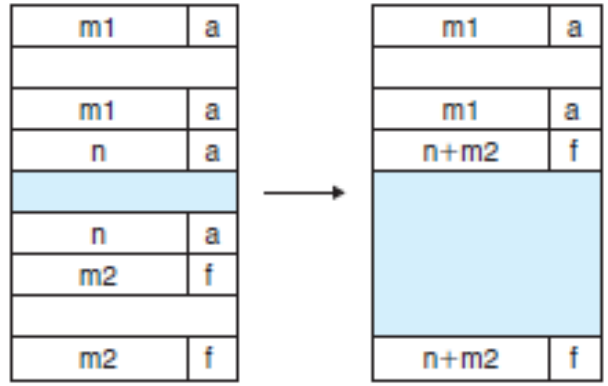
```
1  void mm_free(void *bp)
2  {
3      block_t *block = payload_to_header(bp);
4      size_t size = get_size(block);
5      write_header(block, size, false);
6      write_footer(block, size, false);
7      coalesce_block(block);
8  }
```

隐式空闲链表

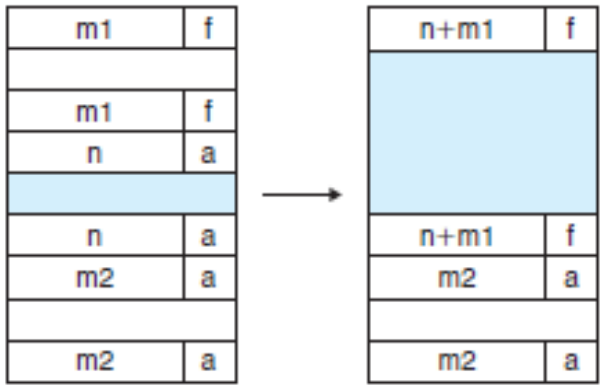
合并的 4 种情况



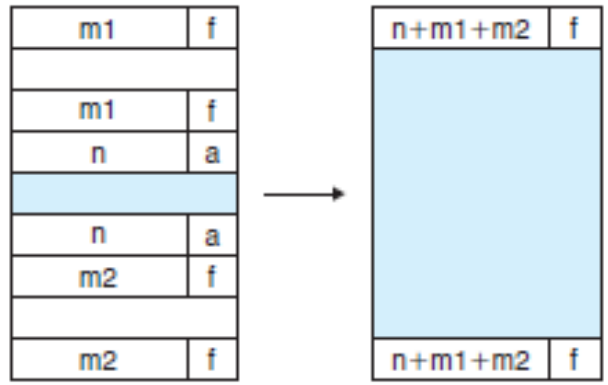
Case 1



Case 2



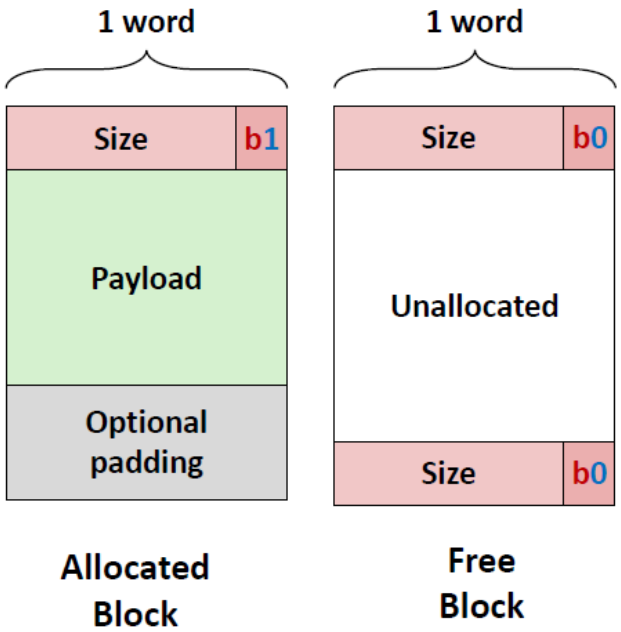
Case 3



Case 4

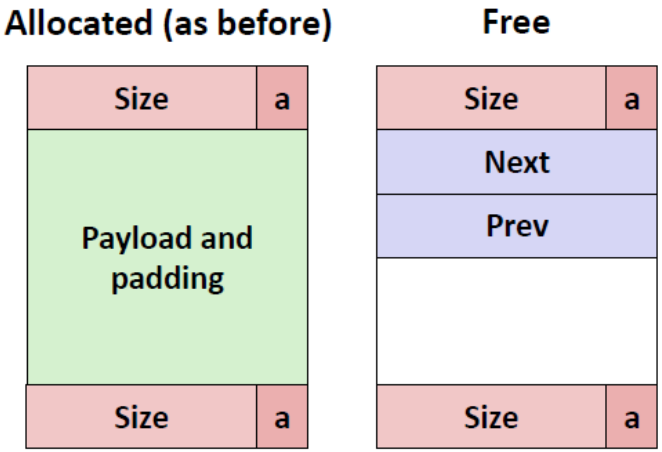
对脚部的优化

- 如果每次请求的空间很小，头部和脚部会占据大量空间
- 只有空闲块需要脚部，但是系统并不知道当前块 B 的前一个块 A 是否空闲
- 可以在块 A 后一个块 B 的头部记录 A 是否为空闲块
- 事实上，由于对齐的限制，头部的 size 很多位是冗余的。若 8 位对齐，则有 3 个冗余位可以记录其他信息



显式空闲块链表

- 只需要建立空白块的链表
- 每个空闲块需要一个前驱指针和一个后继指针
- 每个块的前驱和后继可能指向堆中的任何位置
- 若采用 first fit 查找第一个空闲块，时间复杂度减少到空闲块的线性时间
- 释放一个块的时间取决于块排序策略，可能是线性的也可能是常数



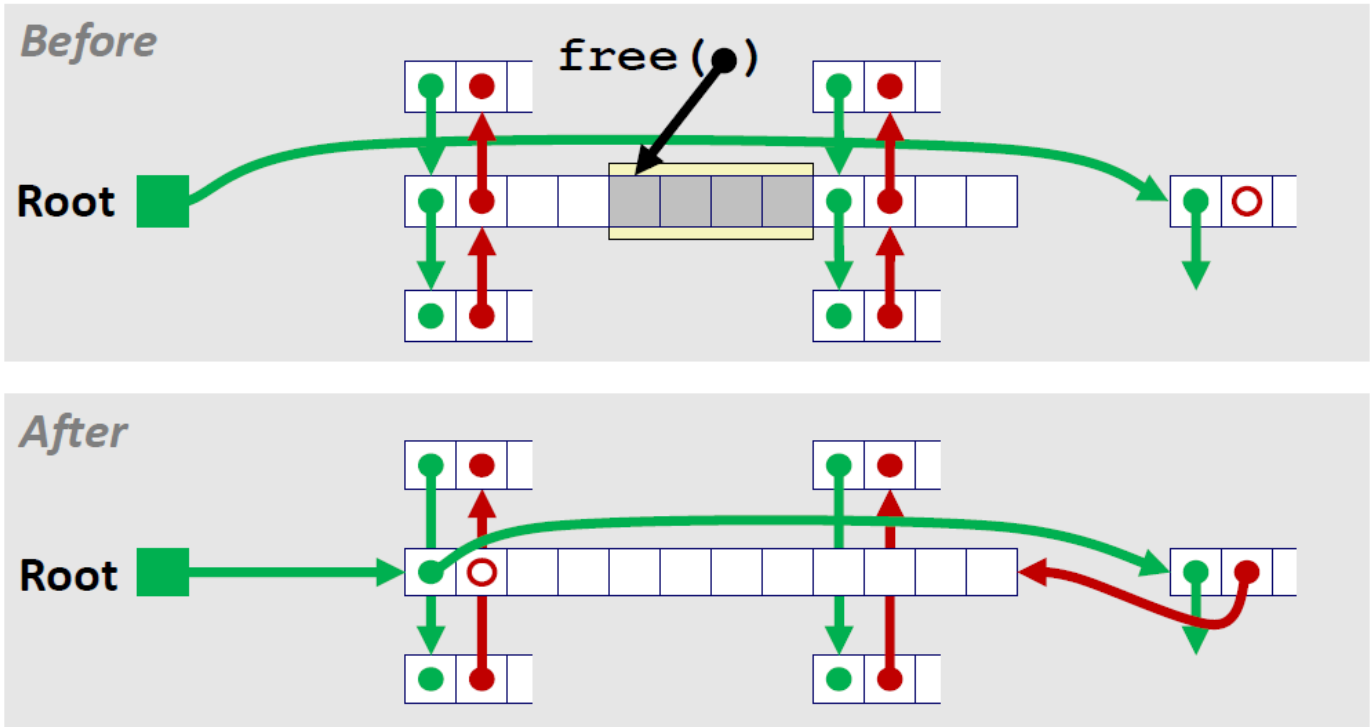
显式空闲块链表

块排序策略：

- 不考虑地址顺序
  - 后进先出 (LIFO)：新释放的块放在链表头 (先被检索到)
  - 先进先出 (FIFO)：新释放的块放在链表尾 (后被检索到)
- 按照地址顺序
  - 优势：碎片化程度低
  - 劣势：时间复杂度高，释放一个块需要线性时间

LIFO 例子

释放两块空闲块之间的块：

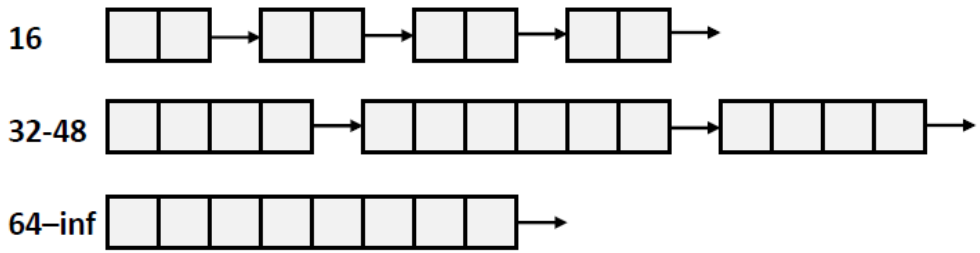


显式空闲块链表

- 在一般情况下，存储空间基本为满，显式空闲块链表比隐式空闲块链表快很多 (不需检查已分配的块)
- 由于前后指针的存在，要求了空闲块有一个更大的最小体积，潜在提高了内部碎片程度

分离的空闲块链表

- 维护多个空闲链表，每个链表中的块大小都相近
- 将所有可能的块大小分成一些等价类，叫做大小类
- 划分方法例子：
  - 按照 2 的幂次划分
  - 小块按照自身大小划分，大块按照 2 的幂次划分
- 两个基本方法
  - 简单分离存储
  - 分离适配



分离的空闲块链表

简单分离存储：

- 每个大小类的空闲链表包含大小相等的块。

  例：大小类为 $\{17\ 32\}$，则块大小为 32
- 分配方式：
  - 相应链表非空：分配第一块的全部 (空闲块不会分割)
  - 相应链表为空：向操作系统申请内存片，分割成大小相同的块，链接起来形成新链表
- 释放方式：将这个块插入到相应空闲链表的头部

分离的空闲块链表

优点：

- 分配和释放都在常数时间内完成
- 可以从地址推断一个已分配块的大小
- 没有合并操作，不需要头部和脚部，空闲块中只需要一个指向下一个块的指针，最小块大小为 1 字

缺点：

- 由于块大小固定，容易产生内部碎片
- 不会合并空闲块，可能会产生大量外部碎片

分离的空闲块链表

分离适配：

- 每个大小类的空闲链表包含的块大小可以不相同。
- 分配方式：对合适的空闲链表做 first fit，若找到合适的块，(可选地分割这个快)，剩余部分插入合适的空闲链表
  - 若找不到合适的块，搜索下一个链表
  - 若搜索完所有链表还是找不到合适的块：向操作系统申请堆内存 (sbrk())，分配出一个合适的块，剩余部分放到相应的空闲链表中
- 释放方式：将这个块插入到相空闲链表的头部

分离的空闲块链表

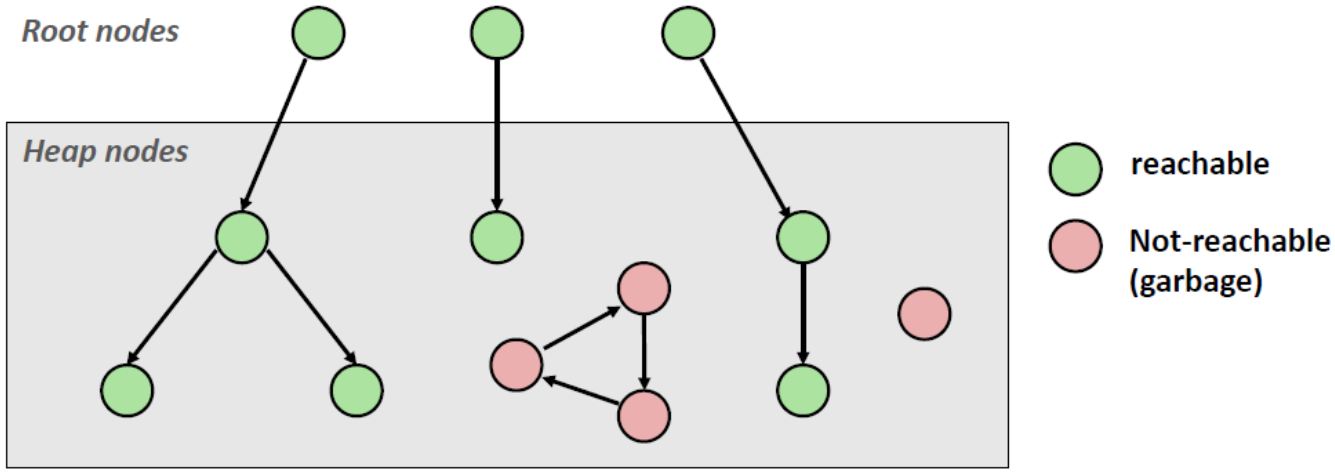
分离适配：

- 块大小可变，空间利用效率更高
- 相比于 best fit，搜索时间更短
- 分离适配的 first fit 内存利用率近似于整个堆空间的 best fit 的利用率.

C 标准库中的 malloc 包采用分离适配

垃圾回收

- 自动释放程序不再需要的分配块
- 垃圾收集器将内存视作一张有向可达图
  - 堆中的每个结点代表堆中的一个块
  - 堆中的每条边代表指向这个块的一个指针
  - 根结点指向堆内的边代表非堆位置 (如寄存器, 栈，全局变量) 指向堆内的指针
  - 若存在一条从根结点到 p 的路径，则称 p 是可达的，否则成为垃圾



Mark and Sweep Collecting

- 在空间耗尽之前不启动垃圾回收，正常调用 malloc
- 当空间耗尽，调用垃圾回收器
  - 从根结点开始染色，在每个块中标记是否是可达结点
  - 遍历堆空间，释放不可达结点

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer -> do nothing
   if (markBitSet(p)) return;       // if already marked -> do nothing
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // for each word in p's block
     mark(p[i]);                    //  make recursive call
   return;
}
```

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {                // for entire heap
      if markBitSet(p)              // did we reach this block?
         clearMarkBit();            //  yes -> so just clear mark bit
      else if (allocateBitSet(p)) // never reached: is it allocated?
         free(p);                   //  yes -> its garbage, free it
      p += length(p);               // goto next block
}
```

Mark and Sweep Collecting

- C 语言只能采用保守的回收策略
- 保守的指可能出现将垃圾标记为可达的情况
- 根本原因：数据可以伪装成指针，is_ptr 函数不能保证正确性！

与内存有关的错误

- 间接引用坏指针
  - scanf("%d", &val)
- 读未初始化的内存
  - 堆中内容并不初始化为 0，应使用 calloc 初始化
- 栈缓冲区溢出
- 指针的大小
  - int **A = (int **)Malloc(n * sizeof(int));
- 指针的运算顺序
  - *size–;

The End