

# 第9章 虚拟内存： 动态内存分配 —— 高级概念

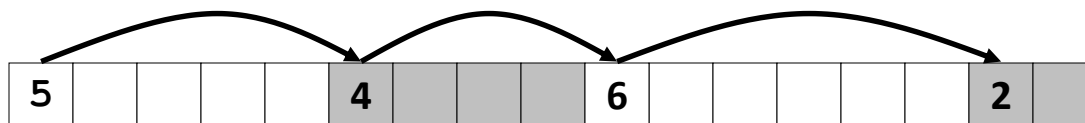
计算机科学与技术学院  
哈尔滨工业大学

# 主要内容

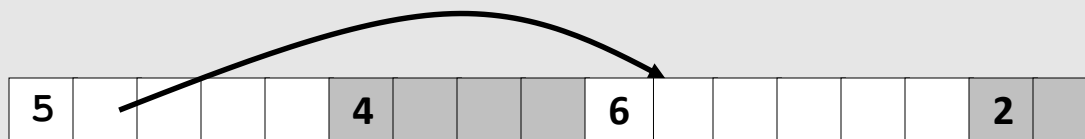
- 显式空闲链表
- 分离的空闲链表
- 垃圾收集
- 内存相关的风险和陷阱

# 跟踪空闲块

- 方法 1: **隐式空闲链表** 通过头部中的大小字段隐含地连接空闲块



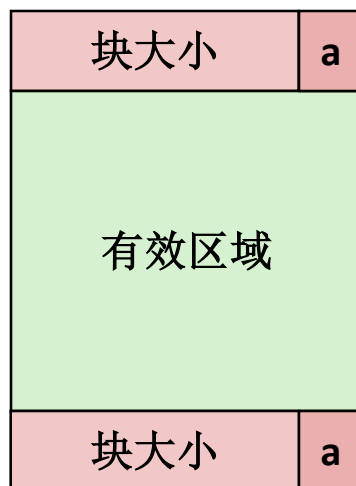
- 方法 2: **显式空闲链表** 在空闲块中使用指针连接空闲块



- 方法 3: **分离的空闲链表**
  - 每个大小类的空闲链表包含大小相等的块
- 方法 4: **按照大小排序的块**
  - 可以使用平衡树（例如红黑树），在每个空闲块中有指针，大小作为键。

# 显式空闲链表

已分配块



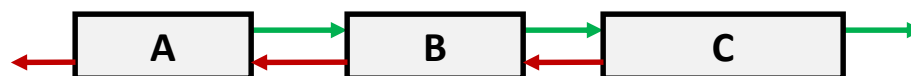
空闲块



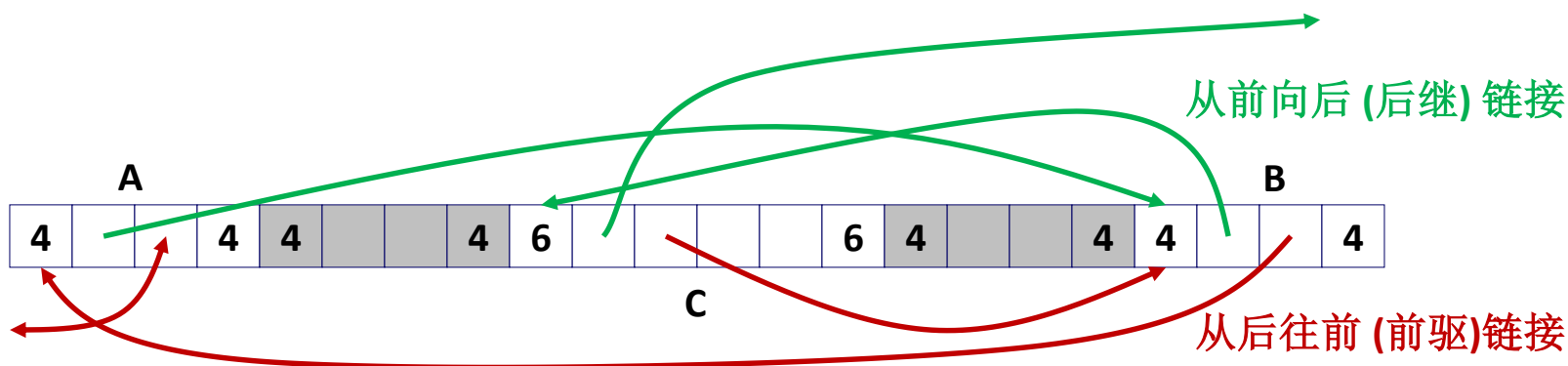
- 保留 **空闲块** 链表, 而不是 **所有块**
  - “下一个” 空闲块可以在任何地方
    - 因此我们需要存储前/后指针，而不仅仅是大小
  - 还需要合并边界标记
  - 幸运的是，我们只跟踪空闲块，所以我们可以使用有效区域。

# 显式空闲链表

## ■ 逻辑地:



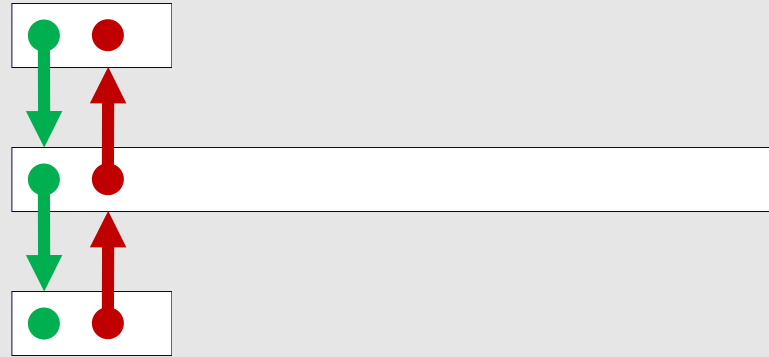
## ■ 物理地: 块的顺序是任何的



# 显式空闲链表的分配

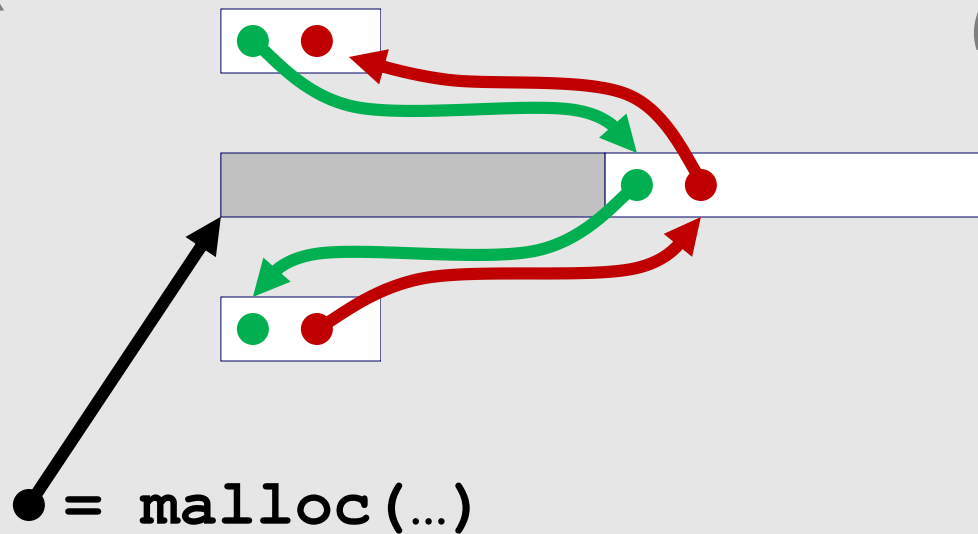
分配示意图

分配前



分配后

(with splitting)



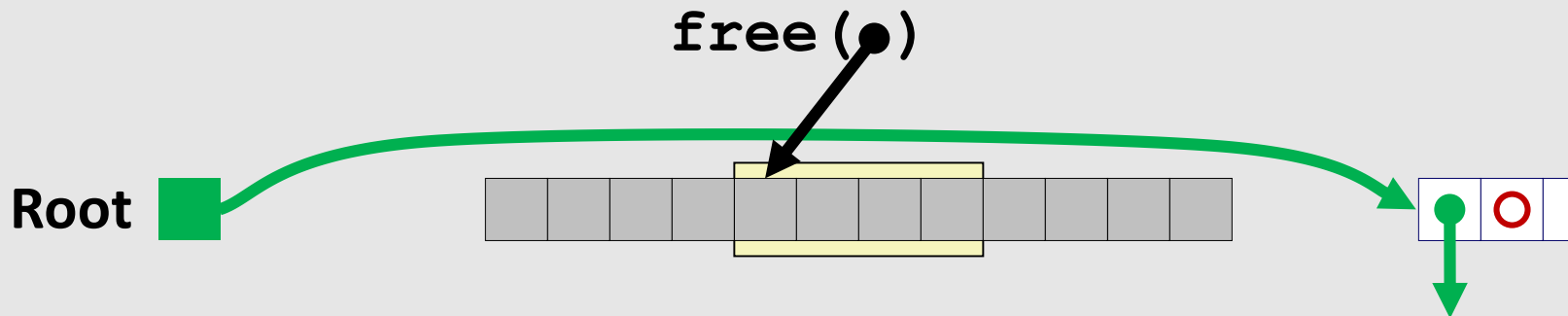
# 显式空闲链表释放

- **插入原则:** 一个新释放的块放在空闲链表的什么位置?
- **LIFO (last-in-first-out) policy** 后进先出法
  - 将新释放的块放置在链表的开始处
  - **Pro好处:** 简单, 常数时间
  - **Con不足:** 研究表明碎片比地址排序更糟糕
- **Address-ordered policy** 地址顺序法
  - 按照地址顺序维护链表:  
$$addr(\text{祖先}) < addr(\text{当前回收块}) < addr(\text{后继})$$
  - **Con不足:** 需要搜索
  - **Pro好处:** 研究表明碎片要少于LIFO (后进先出法)

# LIFO (后进先出) 的回收策略 (案例)

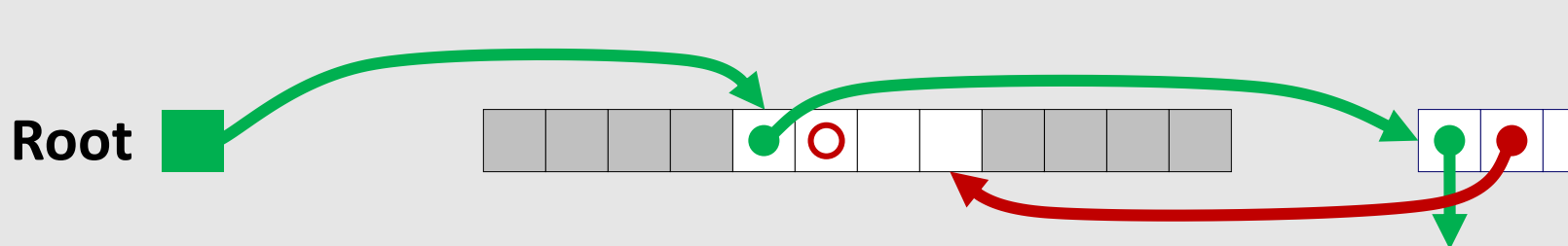
conceptual graphic

回收前



- 将新释放的块放置在链表的开始处

回收后





# 显式链表小结

## ■ 与隐式链表相比较:

- 分配时间从块总数的线性时间减少到空闲块数量的线性时间
  - 当大量内存被占用时 **快得多**
- 因为需要在列表中拼接块，释放和分配稍显复杂一些

## ■ 额外的空间? 每个块需要两个额外的字

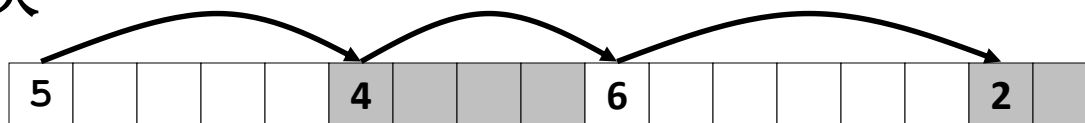
- 这样会不会增加内部碎片?

## ■ 最常用的链表连接是将分离的空闲链表结合在一起

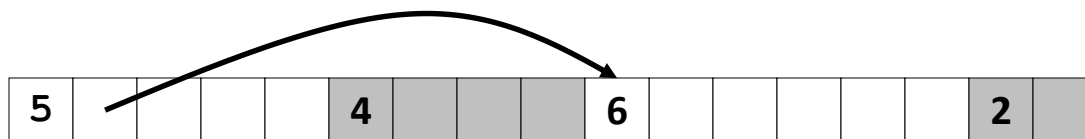
- 维护多个不同大小类的或者有可能的话维护多个不同对象类型的链表

# 跟踪空闲块

- 方法 1: **隐式空闲链表** 通过头部中的大小字段隐含地连接空闲块



- 方法 2: **显式空闲链表** 在空闲块中使用指针连接空闲块



- 方法 3: **分离的空闲链表**

- 每个大小类的空闲链表包含大小相等的块

- 方法 4: **按照大小排序的块**

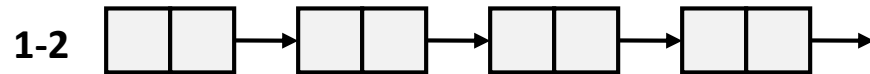
- 可以使用平衡树（例如红黑树），在每个空闲块中有指针，大小作为键。

# 主要内容

- 显示空闲链表
- 分离的空闲链表
- 垃圾收集
- 内存相关的风险和陷阱

# 分离链表分配器

- 每个**大小类**中的块构成一个空闲链表



- 通常每个小块都有单独的大小类
- 对于大块: 按照2的幂分类

# 分离适配

- 分配器维护空闲链表数组，每个大小类一个空闲链表
- 当分配器需要一个大小为 $n$ 的块时：
  - 搜索相应的空闲链表，其大小要满足 $m > n$
  - 如果找到了合适的块：
    - 拆分块，并将剩余部分插入到适当的可选列表中
  - 如果找不到合适的块, 就搜索下一个更大的大小类的空闲链表
  - 直到找到为止。
- 如果空闲链表中没有合适的块：
  - 向操作系统请求额外的堆内存 (使用 `sbrk()`)
  - 从这个新的堆内存中分配出  $n$  字节
  - 将剩余部分放置在适当的大小类中.

# 分离适配

## ■ 释放块:

- 合并，并将结果放置到相应的空闲链表中

## ■ 分离适配的优势

- 更高的吞吐量
- 更高的内存使用率
  - 对分离空闲链表的简单的首次适配搜索，其内存利用率近似于对整个堆的最佳适配搜索的内存利用率.
  - 极端示例：如果每个块都属于它本身尺寸的大小类，那么就相当于最佳适应算法。

# 关于分配的更多信息

- **D. Knuth, “*The Art of Computer Programming*”, 2<sup>nd</sup> edition, Addison Wesley, 1973**
  - 动态存储分配的经典参考
  
- **Wilson et al, “*Dynamic Storage Allocation: A Survey and Critical Review*”, Proc. 1995 Int’l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.**
  - 全面的综述
  - 从 CS:APP student site ([csapp.cs.cmu.edu](http://csapp.cs.cmu.edu)) 可以获取

# 主要内容

- 显示空闲链表
- 分离的空闲链表
- 垃圾收集
- 内存相关的风险和陷阱



# 隐式内存管理----垃圾收集

- **垃圾收集**: 自动回收堆存储的过程—应用从不显式释放

```
void foo() {  
    int *p = malloc(128);  
    return; /* p block is now garbage */  
}
```

- 常见于多种动态语言中:
  - Python, Ruby, Java, Perl, ML, Lisp, Mathematica
- “保守的”垃圾收集器为 **C** 和 **C++** 程序提供垃圾收集
  - 然而，它并不能收集所有的垃圾

# 垃圾收集

## ■ 内存管理器如何知道何时可以释放内存？

- 一般我们不知道下一步会用到什么，因为这取决于具体条件
- 但是我们知道如果没有指针，某些块就不能被使用

## ■ 必须做些关于指针的假设

- 内存管理器可以区分指针和非指针
- 所有指针都指向一个块的起始地址
- 无法隐藏指针

# 经典的垃圾收集算法

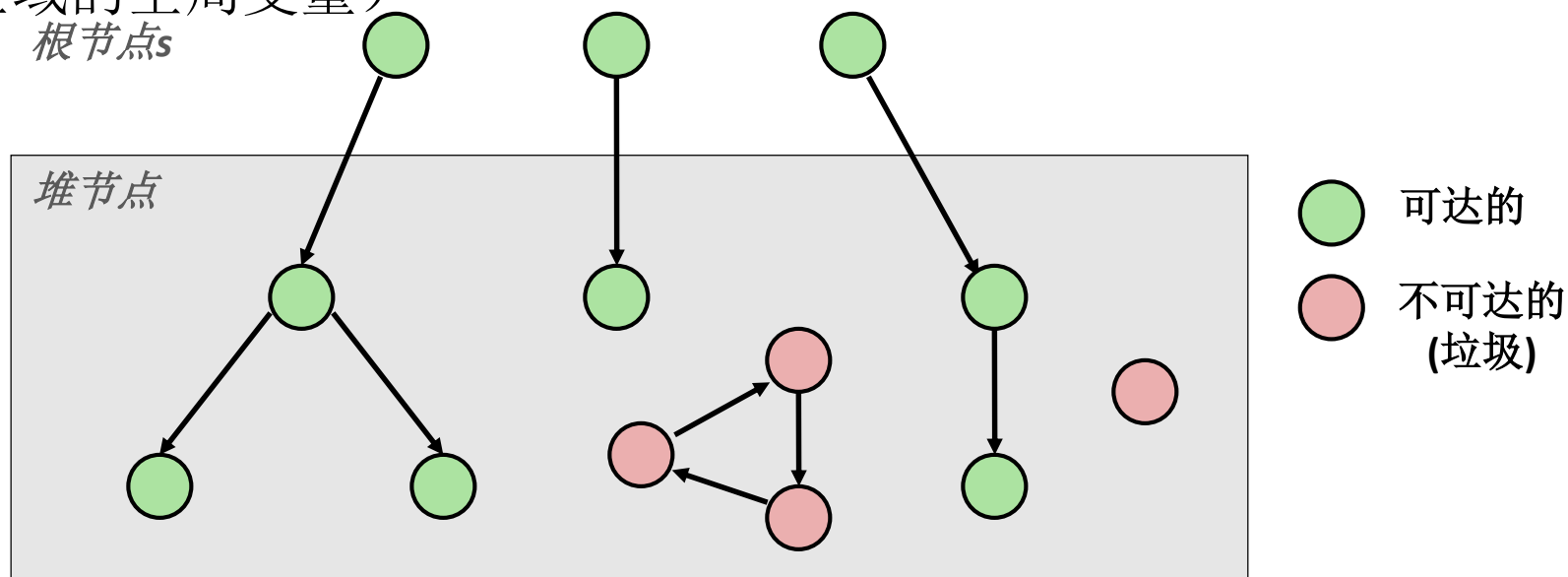
- **Mark-and-sweep collection (McCarthy, 1960) 标记-清除 算法**
  - 不移动块 (除非要 “紧凑”)
- **Reference counting (Collins, 1960) 引用计数**
  - 不移动块 (未讨论)
- **Copying collection (Minsky, 1963) 复制收集**
  - 移动块 (未讨论)
- **Generational Collectors (Lieberman and Hewitt, 1983)**
  - 基于生命期的收集
    - 大部分分配很快就会变成垃圾
    - 因此回收工作的重点应该是刚刚分配的内存区域
- **获得更多信息:**  
**Jones and Lin, “*Garbage Collection: Algorithms for Automatic Dynamic Memory*”, John Wiley & Sons, 1996.**

# 将内存当作图

## ■ 将内存看作一张有向图

- 每个块是图中的一个节点
- 每个指针是图中的一个边，表示某块中的位置指向另一块中的位置
- 根节点的位置不在堆中，包含指向堆的指针

(这些位置可以是寄存器、栈里的变量，或者是虚拟内存中读写数据区域的全局变量)

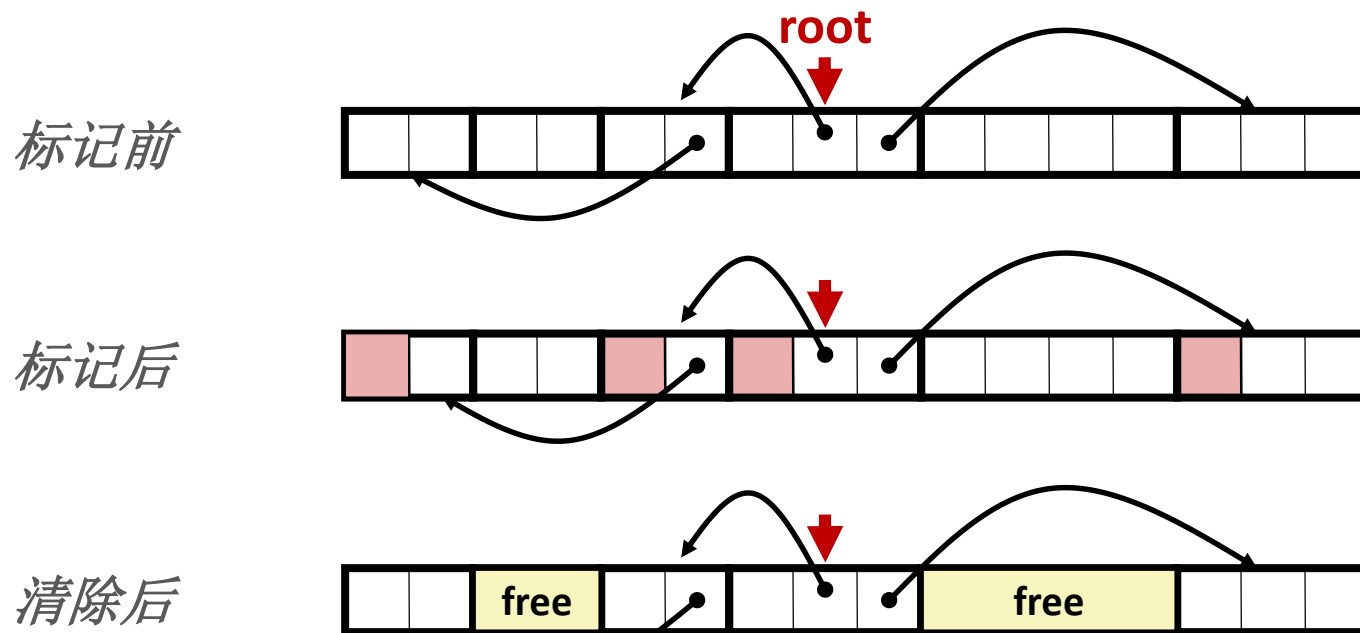


**可达**节点：存在一条从任意根节点出发并到达该节点的有向路径


不可达节点是**垃圾**(不能被应用再次使用)

# 标记&清除（Mark-and-sweep）垃圾收集器

- 可以建立在已存在的malloc包的基础上
  - 使用malloc分配直到你“用完了空间”
- 当“用完了空间”：
  - 使用块头部中的 **mark bit 标记位**
  - **标记**: 从根节点开始标记所有的可达块
  - **清除**: 扫描所有块并释放没有被标记的块



注意: 箭头表示内存引用, 而不是空闲链表指针

 标记位设置

# 简单实现的假设

## ■ 应用

- **new(n)**: 返回指向所有位置已被清除的新块的指针
- **read(b, i)**: 读取 **b** 块位置 **i** 的内容到寄存器
- **write(b, i, v)**: 将内容 **v** 写入到 **b** 块位置 **i**

## ■ 每个块都会有一个包含一个字的头部

- 对于块**b**, 标记为**b[-1]**
- 用在不同的收集器中, 可以起到不同的作用

## ■ 垃圾收集器使用函数的说明

- **is\_ptr(p)**: 判断**p**是不是指向一个已分配块的指针
- **length(b)**: 返回块**b**的以字为单位的长度 (不包括头部)
- **get\_roots()**: 返回所有根节点

# Mark and Sweep (cont.)

## Mark（标记）函数

```
ptr mark(ptr p) {
    if (!is_ptr(p)) return;
    if (markBitSet(p)) return;
    setMarkBit(p);
    for (i=0; i < length(p); i++)
        mark(p[i]);
    return;
}
```

// 不指向一个已分配块则什么都不做  
 // 检查是否已标记  
 // 设置标记位  
 // 调用mark标记块中的每个字  
 // 标记节点p的所有未标记且可达的后继节点

## Sweep（清除）函数

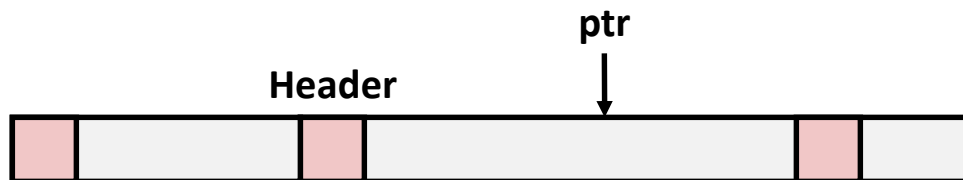
```
ptr sweep(ptr p, ptr end) {
    while (p < end) {
        if markBitSet(p)
            clearMarkBit();
        else if (allocateBitSet(p))
            free(p);
        p += length(p);
    }
}
```

释放所有未标记的  
已分配块（垃圾）

# C程序的保守的Mark & Sweep

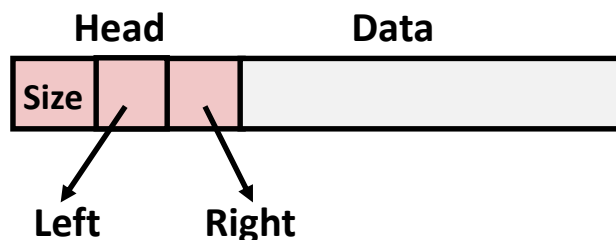
## ■ C程序的“保守的”垃圾收集器

- 通过检查某个字是否指向已分配的内存块来确定该字是否为指针
- 但是，在C语言中指针可以指向一个块的中间位置



## ■ 如何找到块的起始位置?

- 可以使用平衡二叉树跟踪所有分配的块
- 平衡树指针可以存储在每个已分配块的头部（使用两个额外的字left和right）



**Left:** 较小的地址  
**Right:** 较大的地址



# 主要内容

- 显示空闲链表
- 分离的空闲链表
- 垃圾收集
- 内存相关的风险和陷阱

# 内存相关的风险和陷阱

- 间接引用坏指针
- 读未初始化的内存
- 覆盖内存
- 引用不存在的变量
- 多次释放内存
- 引用空闲堆块中的数据
- 释放内存失败

# C operators

## 运算符

() [] -> .  
 ! ~ ++ -- + - \* & (type) sizeof  
 \* / %  
 + -  
 << >>  
 < <= > >=  
 == !=  
 &  
 ^  
 |  
 &&  
 ||  
 ?:  
 = += -= \*= /= %= &= ^= != <<= >>=  
 ,

## Associativity

从左到右

从右到左

从左到右

从左到右

从左到右

从左到右

从左到右

从左到右

从左到右

从左到右

从左到右

从左到右

从右到左

从右到左

从左到右

## C语言中的运算符

优先级	运算符	功能	结合方式
1	() [] -> .	括号, 数组, 两种结构成员访问	由左向右
2	! ~ ++ -- + - * & (类型) sizeof	否定, 按位否定, 增量, 减量, 正负号, 间接, 取地址, 类型转换, 求大小	由右向左
3	* / %	乘, 除, 取模	由左向右
4	+ -	加, 减	由左向右
5	<< >>	左移, 右移	由左向右
6	< <= >= >	小于, 小于等于, 大于等于, 大于	由左向右
7	== !=	等于, 不等于	由左向右
8	&	按位与	由左向右
9	^	按位异或	由左向右
10		按位或	由左向右
11	&&	逻辑与	由左向右
12		逻辑或	由左向右
13	?=	条件	由右向左
14	= += -= *= /= &= ^=  = <<= >>=	各种赋值	由右向左
15	,	逗号 (顺序)	由左向右

# C 指针的使用: 来做自个自测吧!

```
int *p
```

p is a pointer to int (P是指针)

```
int *p[13]
```

p is an array[13] of pointer to int (p是指针数组)

```
int *(p[13])
```

p is an array[13] of pointer to int (p是指针数组)

```
int **p
```

p is a pointer to a pointer to an int (p是指针的指针)

```
int (*p)[13]
```

p is a pointer to an array[13] of int (p是数组指针)

```
int *f()
```

f is a function returning a pointer to int (指针函数, 返回一个指针)

```
int (*f)()
```

f is a pointer to a function returning int (函数指针)

# 间接引用坏指针

## ■ 经典的scanf错误

```
int val;  
  
...  
  
scanf("%d", val) ; //应当&val
```

从stdin读一个整数到一个变量val

正确方法：传递变量的地址给scanf

# 读未初始化的内存

- 常见的错误是假设堆内存被初始化为零

```
/* return y = Ax */
int *matvec(int **A, int *x) {
    int *y = malloc(N*sizeof(int));
    int i, j;

    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            y[i] += A[i][j]*x[j];
    return y;
}
```

# 覆盖内存

## ■ 分配（可能）错误大小的对象

```
int **p;  
  
p = malloc(N*sizeof(int)); //int*  
  
for (i=0; i<N; i++) {  
    p[i] = malloc(M*sizeof(int));  
}
```

假设指向对象的指针和它们所指向的对象是相同大小的  
（这里p是个指针数组）



# 覆盖内存

## ■ Off-by-one error 错位错误

```
int **p;  
  
p = malloc(N*sizeof(int *));  
  
for (i=0; i<=N; i++) {  
    p[i] = malloc(M*sizeof(int));  
}
```

这里指针数组大小为N

# 覆盖内存

## ■ 不检查输入串的大小

```
char s[8];  
int i;  
  
gets(s);    /* reads "123456789" from stdin */  
            //复制任意长度的串到缓冲区
```

## ■ 经典缓冲区溢出攻击的基础

# 覆盖内存

## ■ 误解指针运算

```
int *search(int *p, int val) {  
    //扫描整型数组p  
    while (*p && *p != val)  
        p += sizeof(int);  
  
    return p;  
}
```

指针的算术操作是以它们指向的对象大小为单位

P++: p指向下一个int元素 (p+4)

P+4: p指向下4个int元素\_(p+4\*4)

# 覆盖内存

- 引用指针而不是它所指向的对象

```
int *BinheapDelete(int **binheap, int *size) {  
    int *packet;  
    packet = binheap[0];  
    binheap[0] = binheap[*size - 1];  
    *size--; //应当是(*size)--  
    Heapify(binheap, *size, 0);  
    return(packet);  
}
```

# 引用不存在的变量

- 忘记当函数返回时局部变量将消失

```
int *foo () {  
    int val;  
  
    return &val;  
}
```

栈帧释放后，**val**变量已不存在，对函数所返回指针的赋值（**\*p**）将存在风险

# 多次释放

## ■ Nasty!很讨厌

```
x = malloc(N*sizeof(int));  
    <manipulate x>  
free(x);  
  
y = malloc(M*sizeof(int));  
    <manipulate y>  
free(x);
```

运行会报错！空指针可以多次释放  
若释放后

**x = NULL**  
**free(x)成立**

# 引用空闲堆块中的数据

## ■ Evil!很邪恶

```
x = malloc(N*sizeof(int));  
    <manipulate x>  
free(x);  
    ...  
y = malloc(M*sizeof(int));  
for (i=0; i<M; i++)  
    y[i] = x[i]++; //x[i] 已被释放, 可能已是其他已分配的堆块的部分
```

# 释放失败（内存泄漏）

## ■ Slow, long-term killer!

```
foo() {  
    int *x = malloc(N*sizeof(int));  
    ...  
    return;  
}
```



# 释放失败（内存泄漏）

## ■ 释放部分数据

```
struct list {  
    int val;  
    struct list *next;  
};  
  
foo() {  
    struct list *head = malloc(sizeof(struct list));  
    head->val = 0;  
    head->next = NULL;  
    <create and manipulate the rest of the list>  
    ...  
    free(head);  
    return;  
}
```