动态存储管理机制

情况说明：这个题目实在没有思路，参考了如下网址的资料，对源代码进行分析写了一个总结
link

介绍

通过重载 new 和 delete 运算符，基于**定位创生表达式**（placement new expression）等实现高效的内存管理机制

在程序开始时分配一片足够大的存储空间，用 pool 指针指向；其后所有的内存分配和收回都使用重载后的 new 和 delete 操作符在 pool 指向的存储池（storage pool）上进行.

创生表达式原型

```
void* operator new( std::size_t size, void* p );
```

声明

```
char* base_addr = new char[4096]; // 分配 4096 个字节的存储空间
int* p = new(base_addr) int(16); // 在该存储空间中分配整数匿名数据对象
```

重载格式

```
void* operator new( std::size_t size, STORAGEPOOL* pool )
{
 return Acquire( pool, size );
}
void operator delete( void* ptr )
{
 Reclaim( ptr );
}
```

调用接口

```
int* p = new(pool) int; // 假设 pool 指向的存储池数据对象已分配
delete p; // 使用完毕后通过 delete 操作符调用重载的 operator delete() 回收存储空间
```

Buddy system

项目结构

```
buddy_sysytem/
|
├─storagepool.h
├─storagepool.cpp
├─main.cpp   └──
```

主要思路

初始化流程

1. **构造函数 (`STORAGEPOOL(int size_0)`)**
   - 确定存储池的基础大小。
   - 根据需要初始化伙伴节点系统。
2. **伙伴节点系统初始化 (`initBuddyNodes`)**
   - 为伙伴节点分配内存和初始化数据结构。

内存分配流程

1. **内存请求 (`Acquire(int size_0)`)**
   - 根据请求调整大小并计算所需对数大小。
   - 从合适的大小类别获取节点或通过分割获取。

```
void* STORAGEPOOL::Acquire( int size_0 )
{
    size_0 += 8;
    int Count = log2size( size_0 ) - 4;
    if( size < Count + 4 ) return NULL;

    BuddyNode* node;
    if( SizeArray[Count] )
    {
        node = SizeArray[Count];
        pop( node );
    } else
    {
        int splitCount = Count + 1;
        while( splitCount <= size - 4 && ! SizeArray[splitCount] )
            splitCount++;

        if( splitCount > size - 4 ) return NULL;

        while( Count < splitCount ) {
            split( SizeArray[splitCount] );
            splitCount--;
        }
        node = SizeArray[Count];
    }

    pop( node );
    return startAddr( node );
}
```

2. **节点分割（`split(BuddyNode* node)`）**
   - 将大节点分割成两个小节点以满足内存请求。
3. **调整大小**
   - 如果需要，调整请求的大小以满足内存对齐。

内存回收流程

1. **内存释放（`Reclaim(BuddyNode* node)`）**
   - 标记节点为就绪状态。
   - 尝试与伙伴节点合并。
2. **节点合并（`merge(BuddyNode* node)`）**
   - 合并相邻的空闲节点以形成更大的空闲块。

3. **加入空闲列表 (`join(BuddyNode* node)`)**
    - 将回收的节点加入到相应大小的空闲列表。

伙伴节点管理

1. **伙伴节点创建 (`newNodes`)**
    - 创建新的伙伴节点以供使用。
2. **节点初始化 (`initNodes(BuddyNode* node)`)**
    - 初始化新创建的伙伴节点。
3. **释放节点 (`freeNode(BuddyNode* node)`)**
    - 将不再使用的节点标记为空闲并加入到空闲列表。
4. **清理所有节点 (`freeAllNodes`)**
    - 释放存储池中所有节点的内存。

内存池清理

1. **析构函数 (`~STORAGEPOOL()`)**
    - 清理存储池占用的内存资源。
2. **伙伴节点系统清理 (`deleteBuddyNodes`)**
    - 清理伙伴节点系统使用的内存资源。

工具函数

- **计算对数大小 (`log2size(int size)`)**
    - 确定所需内存块的大小类别。
- **数据池初始化 (`initDataPool(void* DataPool, BuddyNode* node)`)**
    - 在内存块中存储节点信息。
- **获取起始地址 (`startAddr(void* DataAddr)`)**
    - 计算用户可访问的内存起始地址。

全局状态管理

- 管理存储池计数、伙伴节点数组、空闲节点列表等全局状态。

特殊操作符重载

- 重载 `new` 和 `delete` 操作符以使用 `STORAGEPOOL` 类进行内存管理。

伙伴内存管理机制的优势

- 减少内存碎片：

伙伴系统通过将内存分割成固定大小的块，有助于减少外部和内部碎片。

- 快速分配和回收：

由于内存块大小固定，分配和回收操作可以非常快速地执行，因为不需要搜索合适大小的块。

- 做内存对齐

按照指数的方式进行对齐，可能可以满足部分特殊的的系统要求