内存管理

> 主要考察析构函数

由于 libc 原生的内存管理 API 的种种问题（例如，内存泄露和 double-free），某些数据库管理系统基于这些 API 实现了新的动态内存管理机制。具体地，这些实现中引入了一个称为“memory context”的概念（后文简称为“MC”），动态内存管理不再直接使用 `malloc` / `free` 等函数，而是调用某个 MC 对象提供的 API 进行。这些 MC 对象被组织为树状结构，如图 1 所示。这样，使用者不再需要为之前分配的每个内存块调用 `free` 等函数，而是等到相应的 MC 对象被销毁时统一地归还其下的动态内存。同时，由于树状结构的存在，在销毁较为高层的 MC 对象时，处于低层次的 MC 对象也同样会被销毁。这种机制极大降低了管理动态内存的心智负担，更好地避免了内存泄露和 double-free 等问题。

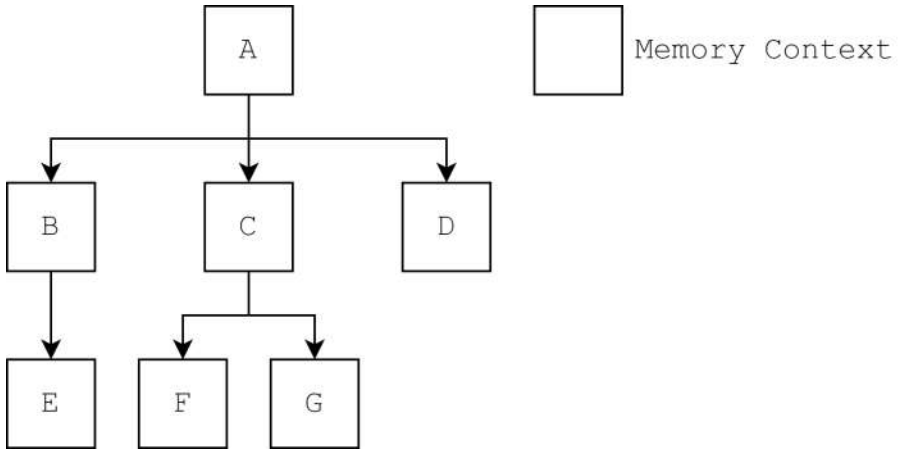

图 1：MC 的树状结构

我们提供了骨架代码，请填充其实现。需要实现如下的功能：

- 维护 MC 对象之间的父子关系
- 给定一个 ID，分配一块内存并将二者关联起来（即，`MemoryContext::alloc` 函数）
- 在析构函数中进行上述的销毁工作

输入描述

本题不需要处理输入。

输出描述

销毁某个 ID 为 `X` 的内存块时，输出

```
Chunk X freed.
```

对于一个 MC 下的内存块，按照**分配顺序的逆序**（即，后进先出的顺序）进行销毁。

如果一个 MC 有多个子 MC，那么按照创建这些子 MC 的顺序进行销毁。

示例

示例 1

MC 结构和操作序列

图 2：示例 1 的 MC 结构

操作序列为：

1. A.alloc("1")
2. A.alloc("2")
3. A.alloc("3")

输出

```
Chunk 3 freed.
Chunk 2 freed.
Chunk 1 freed.
```

示例 2

MC 结构和操作序列

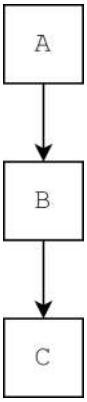

图 3：示例 2 的 MC 结构

操作序列为:

1. A.alloc("1")
2. A.alloc("2")
3. A.alloc("3")
4. B.alloc("1/1")
5. B.alloc("1/2")
6. B.alloc("1/3")
7. C.alloc("1/1/1")
8. C.alloc("1/1/2")
9. C.alloc("1/1/3")

输出

```
Chunk 1/1/3 freed.
Chunk 1/1/2 freed.
Chunk 1/1/1 freed.
Chunk 1/3 freed.
Chunk 1/2 freed.
Chunk 1/1 freed.
Chunk 3 freed.
Chunk 2 freed.
Chunk 1 freed.
```

示例 3

MC 结构和操作序列

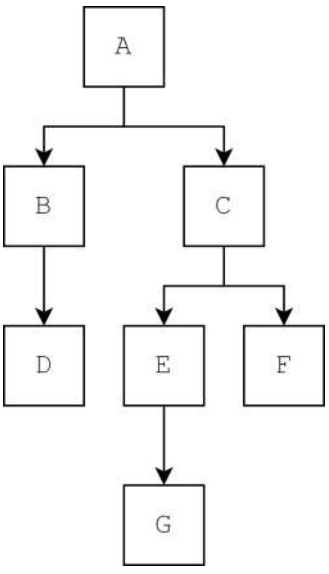

图 4：示例 3 的 MC 结构

操作序列：

1. A.alloc("1")
2. A.alloc("2")
3. A.alloc("3")
4. B.alloc("1/1")
5. C.alloc("1/2")
6. D.alloc("1/1/1")
7. D.alloc("1/1/2")
8. G.alloc("1/2/1/1")

输出

```
Chunk 1/1/2 freed.
Chunk 1/1/1 freed.
Chunk 1/1 freed.
Chunk 1/2/1/1 freed.
Chunk 1/2 freed.
Chunk 3 freed.
Chunk 2 freed.
Chunk 1 freed.
```

骨架代码

```
#include <cassert>
#include <functional>
#include <iostream>
#include <memory>
#include <string>
#include <unordered_map>

// 除了 TODO 标出的部分，不要修改原有的声明和定义，否则后果自负！

class MemoryContext {
 public:
  /**
   * @param parent 父节点，可能为 nullptr
   */
  MemoryContext(MemoryContext *parent);

  ~MemoryContext();
```

```
  // 禁止拷贝和移动
  MemoryContext(const MemoryContext &) = delete;
  MemoryContext &operator=(const MemoryContext &) = delete;
  MemoryContext(MemoryContext &&) = delete;
  MemoryContext &operator=(MemoryContext &&) = delete;

  using chunk_id_t = std::string;

  void alloc(const chunk_id_t &chunk_id);

 private:
  // TODO: your code
};

MemoryContext::MemoryContext(MemoryContext *parent) {
  // TODO: your code
}

MemoryContext::~MemoryContext() {
  // TODO: your code
}

void MemoryContext::alloc(const chunk_id_t &chunk_id) {
  // TODO: your code
}

void test_1() {
  std::unique_ptr<MemoryContext> A = std::make_unique<MemoryContext>
(nullptr);
  A->alloc("1");
  A->alloc("2");
  A->alloc("3");
}

void test_2() {
  std::unique_ptr<MemoryContext> A = std::make_unique<MemoryContext>
(nullptr);
  MemoryContext *B = new MemoryContext(A.get());
  MemoryContext *C = new MemoryContext(B);

  A->alloc("1");
  A->alloc("2");
  A->alloc("3");
  B->alloc("1/1");
  B->alloc("1/2");
  B->alloc("1/3");
  C->alloc("1/1/1");
  C->alloc("1/1/2");
```

```
  C->alloc("1/1/3");
}

void test_3() {
  std::unique_ptr<MemoryContext> A = std::make_unique<MemoryContext>
(nullptr);
  MemoryContext *B = new MemoryContext(A.get());
  MemoryContext *C = new MemoryContext(A.get());
  MemoryContext *D = new MemoryContext(B);
  MemoryContext *E = new MemoryContext(C);
  MemoryContext *F = new MemoryContext(C);
  MemoryContext *G = new MemoryContext(E);

  A->alloc("1");
  A->alloc("2");
  A->alloc("3");
  B->alloc("1/1");
  C->alloc("1/2");
  D->alloc("1/1/1");
  D->alloc("1/1/2");
  G->alloc("1/2/1/1");
}

void test_4() {
  std::unique_ptr<MemoryContext> A = std::make_unique<MemoryContext>
(nullptr);
  MemoryContext *B = new MemoryContext(A.get());
  MemoryContext *C = new MemoryContext(A.get());
  MemoryContext *D = new MemoryContext(B);
  MemoryContext *E = new MemoryContext(B);
  MemoryContext *F = new MemoryContext(C);
  MemoryContext *G = new MemoryContext(C);

  A->alloc("1");
  A->alloc("2");
  A->alloc("3");
  B->alloc("1/1");
  C->alloc("1/2");
  D->alloc("1/1/1");
  D->alloc("1/1/3");
  E->alloc("1/1/2");
  F->alloc("1/2/1");
  G->alloc("1/2/3");
  G->alloc("1/2/5");
  G->alloc("1/2/2");
  G->alloc("1/2/4");
}

void test_5() {
```

```
  std::unique_ptr<MemoryContext> A = std::make_unique<MemoryContext>
(nullptr);
  MemoryContext *B = new MemoryContext(A.get());
  MemoryContext *C = new MemoryContext(A.get());
  MemoryContext *D = new MemoryContext(B);
  MemoryContext *G = new MemoryContext(C);

  A->alloc("2");
  A->alloc("1");
  A->alloc("3");
  A->alloc("4");
  B->alloc("2/1");
  B->alloc("3/5");
  C->alloc("1024/2");
  C->alloc("1024/1");
  G->alloc("8192/1/4095");
}

#define REGISTER_TEST_CASE(name) \
  { #name, name }

int main() {
  std::unordered_map<std::string, std::function<void()>>
      test_functions_by_name = {
          REGISTER_TEST_CASE(test_1), REGISTER_TEST_CASE(test_2),
          REGISTER_TEST_CASE(test_3), REGISTER_TEST_CASE(test_4),
          REGISTER_TEST_CASE(test_5),
      };

  std::string test_case_name;
  std::cin >> test_case_name;
  auto it = test_functions_by_name.find(test_case_name);
  assert(it != test_functions_by_name.end());
  auto fn = it->second;
  fn();
}
```