Тема 7. Контейнеры и аллокаторы

Рассмотренные ранее списочные структуры использовали однородные узловые элементы (ListNode, TreeNode и т.п.). Эти элементы размещались в памяти при помощи стандартных системных операций типа new или malloc. Эти стандартные операции достаточно ресурсоемки, т.е. выполняются относительно медленно и используют дополнительеную память на каждый выделенный элемент. Можно самому управлять таким выделением, предоставивь простейший контейнер-аллокатор.

Allocator — Класс, который размещает (однородные) элементы в памяти по запросу и также по запросу освобождает эти элементы (удаляет их из памяти).

```
template <class T>
class Allocator
{
   public:
      T * Allocate();
      void Delete(const T*);
};
```

Идея реализации — список блоков-массивов для размещения элементов.
Элементом блока является объединение

```
template <class T>
union AllocatorNode
{
    T value;                      // T() = default;
    AllocatorNode *next;
};
```

Иллюстрация.

```
template <class T>
class AllocatorBlock
{
    AllocatorNode<T> *elems;
    int size;
    AllocatorBlock *next;
 public:
    AllocatorBlock(int n) {
       size = n;
       *elems = new AllocatorNode<T>[size];
       next = nullptr;
    }
};
```

Блоки завязаны в список. Элементы блоков “провязаны” ссылками траектории свободных мест.

```
template <class T>
class Allocator
{
    int blocksize;
    int size, maxsize;  // сколько занятых и всего
    AllocatorBlock<T> * startOfBlocks;  // начало списка блоков
    AllocatorNode<T> * startOfNodes;    // начало списка свободных узлов

    public:
      Allocator(int bsize = 64);
      T * Allocate();
      void Delete(const T*);
};

template <class T>
Allocator<T>::Allocator(int bsize)
{
    blocksize = bize;
    startOfBlocks = new AllocatorBlock<T>(blocksize);
    startOfNodes = startOfBlocks.elems;
    for (int i=0; i<blocksize-1; i++)  {
        startOfBlocks.elems[i].next = startOfBlocks.elems + (i+1);
    }
    startOfBlocks.elems[blocksize-1].next = nullptr;
    size = 0;
    maxsize = blocksize;
}

template <class T>
T * Allocator<T>::Allocate()
{
    if (size == maxsize) { // нет места
       выделить новый блок  new AllocatorBlock<T>(blocksize)
       привязать его к цепочке блоков
       разметить его элементы как свободные и включить в траекторию свободных
       обновить maxsize
       в случае отказа вернуть nullptr
    }
    // место есть
    T * ptr = &(startOfNodes->value);
    startOfNodes = startOfNodes.next;
    return ptr;
}

template <class T>
void Allocator<T>::Delete(const T* p)
{
```

```
        AllocatorNode<T> * q = reinterpret_cast<AllocatorNode<T> *>(p);
        q->next = startOfNodes;
        startOfNodes = q;
        ++size;
    }
```

Еще реализоать деструкторы блоков и самого аллокатора.

Реализация абсолютно не защищена!!!

При удалении надо проверять входящий адрес на правильность.

Можно также ввести битовое множество для разметки свободных занятых участков.

Можно возвращать системе освобожденные блоки (delete).

Элементы разного размера.

Функции типа malloc | free.

Идея 1.

Поддерживаем в памяти фрагментированность.

Свободные участки определяются адресом и размером.

Занятые участки также определяются адресом и размером.

Для регистрации имеем два дерева:

Дерево свободных — упорядочено по размеру

Дерево занятых упорядочено по адресу.

Запрос на размещение — ищем подходящий участок в дереве свободных,

удаляем его из дерева,

разделяем на занятую и свободную части,

заносим эти части в соответствующие деревья.

Запрос на освобождение — ищем адрес в дереве занятых,

удаляем из дерева занятых,

размещаем в дереве свободных.

Плохо — сохраняется фрагментация.

Можно все фрагменты прописать в двусвязный список, упорядоченный по адресам,

и через него выполнять объединение соседних свободных фрагментов.

Но подобная структура уже получается слишком громоздкой.

Идея 2.

Выделяется некоторый значительный блок памяти.

В каждом блоке выделение идет по стековому принципу.

В окрестности каждого каждого фрагмернта записываются

— длина фрагмента

— адрес предыдущего фрагмента

— признак занятости

Блок хранит адрес начала свободного места и количество занятых фрагментов. При освобождении выполняется слияние с соседними свободными фрагментами. Если освбождается "вершина стека то сдвигается адрес свободного места.

Элементы “переменного” размера.

Реальные и виртуальные файловые системы. Структурное разбиение диска, BR, MBR, partition table.

В реальных файловых системах множество технических соглашений и вариантов, в том числе по поводу прав доступа, режимов использования файлов и т.п. Поиск файла — понятие каталога (имена файлов) и структурной организации хранения.

Здесь это мы не рассматриваем.

объект (файл) — набор блоков (кластеров)

FAT — однонаправленный список кластеров. Ссылки между кластерами хранятся в отдельной таблице File Allocation Table, там же отмечены свобоные кластеры. Области FAT, DIR, DATA. Расширение VFAT для длинных имен. Система Fat32, ExFat.

Ext — все блоки файла перечислены в таблице наподобие B-дерева. Отдельное битовое множество описывает распределение свободных/занятых блоков. BlockGroup: superblock, descriptor table, inode bitmap, block bitmap, inodes, blocks.

NTFS — все блоки файла перечислены отдельной записью (с использованием участков — run). Отдельное битовое множество описывает распределение свободных/занятых блоков. Master File Table, Data blocks. File Record, метафайлы, атрибуты файлов.

ISO — система на CD дисках нам не интересна так как там файлы после записи уже не меняются. Оглавление и данные (блоки) файла по типу односвязного списка.