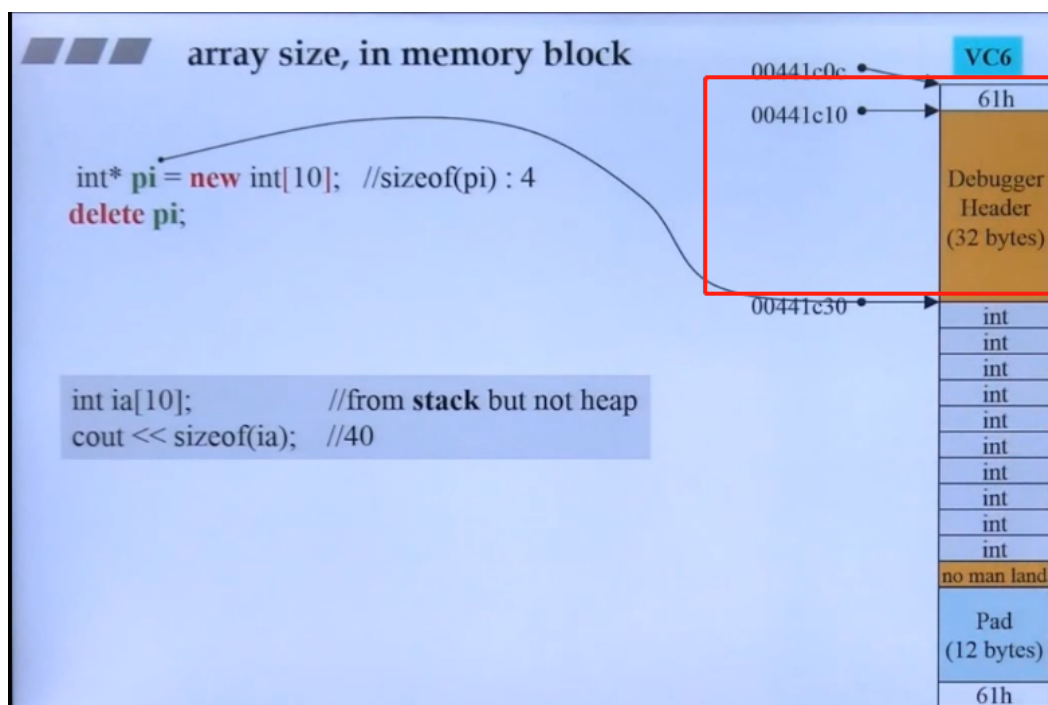


C++内存管理

per-class allocator

引入内存池，一次性分配大内存。因为malloc前面有个cookie，占用一部分空间。内存池的设计就是处于两个目的：1) 速度，减少调用malloc的次数（虽然malloc也挺快的）。2) 空间，减少每个malloc前面的cookie，防止浪费。



per-class allocator 2

上一节中添加进了一个next指针指向下一个，这样会多出4个字节。在该版本中，使用union，让指针借用前4个字节，避免多分配空间？嵌入式的指针。

static allocator

前边的都是在每个类里单独重载operator new和operator delete，重复性的工作太多。单独设计一个类allocator，里面有allocate和deallocate两个函数，可以被其他类所调用执行。

static allocator

當你受困於必須為不同的 classes 重寫一遍幾乎相同的 member operator new 和 member operator delete 時，應該有方法將一個總是分配特定尺寸之區塊的 memory allocator 概念包裝起來，使它容易被重複使用。以下展示一種作法，每個 allocator object 都是個分配器，它體內維護一個 free-lists；不同的 allocator objects 維護不同的 free-lists。

```
class allocator
{
private:
    struct obj {
        struct obj* next; //embedded pointer
    };
public:
    void* allocate(size_t);
    void deallocate(void*, size_t);
private:
    obj* freeStore = nullptr;
    const int CHUNK = 5; //小一些以便觀察
};
```

```
void
allocator::deallocate(void* p, size_t)
{
    //將 *p 收回插入 free list 前端
    ((obj*)p)->next = freeStore;
    freeStore = (obj*)p;
}
```

```
void* allocator::allocate(size_t size)
{
    obj* p;
    if (!freeStore) {
        //linked list 為空，於是申請一大塊
        size_t chunk = CHUNK * size;
        freeStore = p = (obj*)malloc(chunk);
        //將分配得來的一大塊當做 linked list 般，
        //小塊小塊串接起來
        for (int i=0; i < (CHUNK-1); ++i) {
            p->next = (obj*)((char*)p + size);
            p = p->next;
        }
        p->next = nullptr; //last
        p = freeStore;
        freeStore = freeStore->next;
        return p;
    }
}
```

BooLan

每次只要了5小块，标准的allocator是20？那么这5个之间应该相邻，但是5个和5个之间不一定相邻。

static allocator

```
class Foo {
public:
    long L;
    string str;
    static allocator myAlloc;
public:
    Foo(long l) : L(l) { }
    static void* operator new(size_t size)
    { return myAlloc.allocate(size); }
    static void operator delete(void* pdead, size_t size)
    { return myAlloc.deallocate(pdead, size); }
};
allocator Foo::myAlloc;
```

```
class Goo {
public:
    complex<double> c;
    string str;
    static allocator myAlloc;
public:
    Goo(const complex<double>& x) : c(x) { }
    static void* operator new(size_t size)
    { return myAlloc.allocate(size); }
    static void operator delete(void* pdead, size_t size)
    { return myAlloc.deallocate(pdead, size); }
};
allocator Goo::myAlloc;
```

這比先前的設計乾淨多了，application classes 不再與內存分配細節糾纏不清，所有相關細節都讓 allocator 去操心，我們的工作是讓 application classes 正確運作。

macro for static allocator

设计成了宏

macro for static allocator

```
class Foo {
public:
    long L;
    string str;
    static allocator myAlloc;
public:
    Foo(long l) : L(l) { }
    static void* operator new(size_t size)
    { return myAlloc.allocate(size); }
    static void operator delete(void* pdead, size_t size)
    { return myAlloc.deallocate(pdead, size); }
};
allocator Foo::myAlloc;
```

```
// DECLARE_POOL_ALLOC -- used in class definition
#define DECLARE_POOL_ALLOC() \
public: \
    void* operator new(size_t size) { return myAlloc.allocate(size); } \
    void operator delete(void* p) { myAlloc.deallocate(p, 0); } \
protected: \
    static allocator myAlloc;

// IMPLEMENT_POOL_ALLOC -- used in class implementation file
#define IMPLEMENT_POOL_ALLOC(class_name) \
allocator class_name::myAlloc;
```

写的时候更加精简了。

```
class Foo {
    DECLARE_POOL_ALLOC()
public:
    long L;
    string str;
public:
    Foo(long l) : L(l) { }
};
IMPLEMENT_POOL_ALLOC(Foo)
```

```
class Goo {
    DECLARE_POOL_ALLOC()
public:
    complex<double> c;
    string str;
public:
    Goo(const complex<double>& x) : c(x) { }
};
IMPLEMENT_POOL_ALLOC
```

Boolean

new handler

调用new失败前不止一次调用new handler，这个可以由程序员设定，相当于在抛异常前由你来决定该怎么办。

new handler

當 operator new 沒能力為你分配出你所申請的 memory，會拋一個 `std::bad_alloc` exception。某些老舊編譯器則是返回 0 — 你仍然可以令編譯器那麼做：

```
new (nothrow) Foo;
```

此稱為 nothrow 形式。

拋出 exception 之前會先（不只一次）調用一個可由 client 指定的 handler，以下是 new handler 的形式和設定方法：

```
typedef void (*new_handler)();
new_handler set_new_handler(new_handler p) throw();
```

設計良好的 new handler 只有兩個選擇：

- 讓更多 memory 可用
- 調用 `abort()` 或 `exit()`