C/C++内存管理复习

1. C/C++内存管理

```
int globalVar = 1;
static int staticGlobalVar = 1;
void Test()
{
    static int staticVar = 1;
    int localVar = 1;

    int num1[10] = { 1, 2, 3, 4 };
    char char2[] = "abcd";
    const char* pChar3 = "abcd";
    int* ptr1 = (int*)malloc(sizeof(int) * 4);
    int* ptr2 = (int*)calloc(4, sizeof(int));
    int* ptr3 = (int*)realloc(ptr2, sizeof(int) * 4);
    free(ptr1);
    free(ptr3);
}

1. 选择题：
   选项：A.栈  B.堆  C.数据段(静态区)  D.代码段(常量区)
   globalVar在哪里？_____
   staticGlobalVar在哪里？_____
   staticVar在哪里？_____
   localVar在哪里？_____
   num1 在哪里？_____

   char2在哪里？_____
   *char2在哪里？___
   pChar3在哪里？_____
   *pChar3在哪里？_____
   ptr1在哪里？_____
   *ptr1在哪里？_____
```

1. `globalVar` 是一个全局变量，全局变量的生命周期是整个程序运行期间，存储在数据段（静态存储区）。C
2. `staticGlobalVar` 是静态全局变量，静态全局变量和普通全局变量一样存储在数据段，但它的作用域仅限于当前文件（链接属性不同）。C

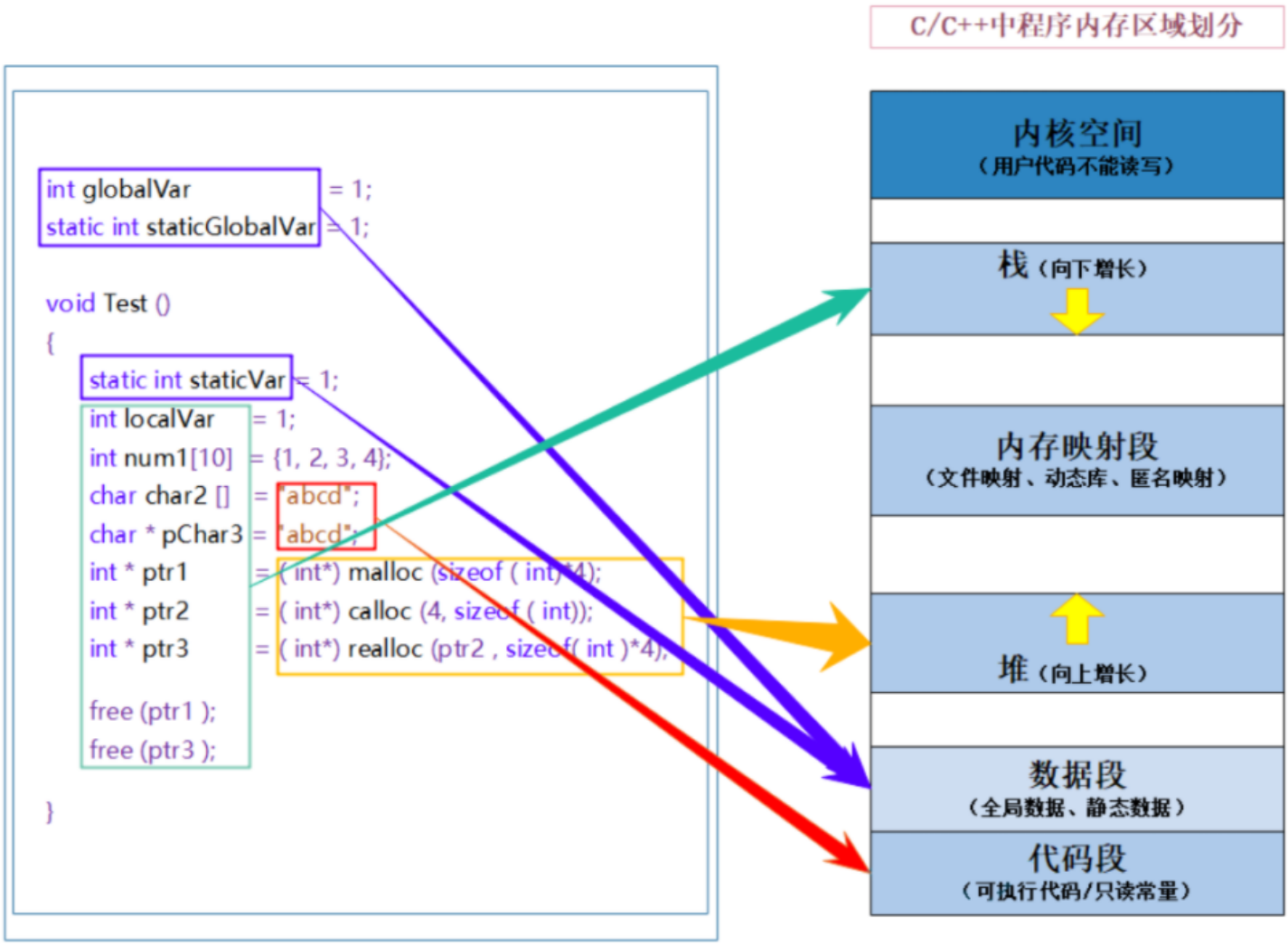
3. `staticVar` 是函数内的静态局部变量，静态局部变量的生命周期是整个程序运行期间（但作用域仅限于函数内），存储在数据段。 C
4. `localVar` 是普通局部变量，局部变量的生命周期是函数调用期间，存储在栈上，函数返回时自动释放。 A
5. `num1` 是一个局部数组，虽然它存储了多个元素，但它本身是一个栈上的变量，数组的内存空间在栈上分配。 A
6. `char2` 是一个局部字符数组，定义在函数内部，存储在栈上。 **`char2` 的内容 `"abcd"` 会被拷贝到栈上的数组空间，而不是直接引用常量区。** A
7. `*char2` 是 `char2` 数组的第一个字符（即 `'a'`），而 `char2` 的整个内容（`"abcd"`）存储在栈上，所以 `*char2` 也在栈上。
8. `pChar3` 是一个局部指针变量，指针变量本身存储在栈上。
9. `pChar3` 指向的是一个字符串常量 `"abcd"`，字符串常量存储在代码段（常量区），因此 `*pChar3`（即 `'a'`）在常量区。**字符串常量是只读的，编译器会将其放在常量区，不同于栈或堆。**
10. `ptr1` 是一个局部指针变量，存储在栈上。
11. `ptr1` 指向的内存是通过 `malloc` 动态分配的，动态内存分配在堆上，因此 `*ptr1`（即 `ptr1` 指向的内容）在堆区。

总结对比

变量/表达式	存储位置	原因
globalVar	数据段（静态区）	全局变量，静态存储
staticGlobalVar	数据段（静态区）	静态全局变量，作用域限于文件，但存储位置与全局变量相同
staticVar	数据段（静态区）	静态局部变量，生命周期全局，作用域局部
localVar	栈	普通局部变量，函数调用时分配，返回时释放
num1	栈	局部数组，栈上分配连续空间
char2	栈	局部字符数组，内容 **"abcd"** 被拷贝到栈
***char2**	栈	**char2** 的内容在栈上，***char2** 是第一个字符
pChar3	栈	局部指针变量
***pChar3**	代码段（常量区）	**pChar3** 指向字符串常量，常量存储在只读区域
ptr1	栈	局部指针变量
***ptr1**	堆	**ptr1** 指向 **malloc** 分配的堆内存

关键区分点

1. 局部变量（如 `localVar`、`num1`、`char2`、`pChar3`、`ptr1`）默认在栈上。
2. 静态变量（如 `staticVar`、`staticGlobalVar`、`globalVar`）在数据段。
3. 动态分配的内存（如 `malloc` / `calloc`）在堆上。
4. 字符串常量（如 `"abcd"`）在代码段（常量区）。



【说明】

1. **栈**又叫堆栈--非静态局部变量/函数参数/返回值等等，栈是向下增长的。

2. **内存映射段**是高效的I/O映射方式，用于装载一个共享的动态内存库。用户可使用系统接口创建共享共享内存，做进程间通信。

3. **堆**用于程序运行时动态内存分配，堆是可以上增长的。

4. **数据段**--存储全局数据和静态数据。

5. **代码段**--可执行的代码/只读常量

2. C语言中的动态内存管理方式：malloc/calloc/realloc/free

1. malloc/calloc/realloc的区别?

表格

	函数	功能	初始化内存	
1	malloc	分配指定大小的内存块	不初始化	ma
2	calloc	分配指定数量和大小的内存块，并初始化为 0	初始化为 0	call size
3	realloc	调整已分配内存块的大小（可扩大或缩小），可能移动内存到新地址	保留原数据	rea size

3 条记录

详细说明：

- **malloc**
  - 只分配内存，不初始化内容（内容可能是随机的）。
  - 示例：`int *arr = (int*)malloc(10 * sizeof(int));`
  - 适用于不需要初始化的场景（如后续手动赋值）。
- **calloc**
  - 分配内存并初始化为 0（适合数组或结构体）。
  - 示例：`int *arr = (int*)calloc(10, sizeof(int));`
  - 适用于需要清零内存的场景（如避免未初始化导致的错误）。
- **realloc**
  - 调整已分配内存的大小（可扩大或缩小）：
    - 如果新大小 ≤ 原大小，可能原地缩减。
    - 如果新大小 > 原大小，可能移动内存到新地址（原数据会被保留）。
  - 示例：`arr = (int*)realloc(arr, 20 * sizeof(int));`
  - 适用于动态扩容（如可变长数组）。

注意事项：

1. 内存泄漏风险：`realloc` 失败时返回 `NULL`，但原指针仍有效，需正确处理：

代码块

```
int *new_ptr = (int*)realloc(ptr, new_size);
if (new_ptr == NULL)
{
    // 处理失败，ptr 仍需手动释放
    free(ptr);
    return ERROR;
}
ptr = new_ptr; // 更新指针
```

2. 性能差异：`calloc` 因初始化稍慢，但安全性更高；`malloc` 更快但不安全。

2. malloc的实现原理?

> `malloc` 是 C 标准库提供的动态内存分配函数，其底层实现依赖操作系统的内存管理机制。以下是核心实现原理：

1）内存管理的基本方式

- 系统调用层：
  - 在 Linux 中，`malloc` 最终调用 `brk` 或 `mmap` 向操作系统申请内存：
    - `brk`：调整堆顶指针，适合小内存分配。
    - `mmap`：直接映射匿名内存页，适合大内存分配（如超过 `MMAP_THRESHOLD`，默认 128KB）。
- 用户层管理：
  - `malloc` 不会每次分配都调用系统调用（昂贵），而是维护一个内存池，通过算法管理已分配和空闲的内存块。

2）内存池与空闲链表

- 空闲链表（Free List）：
  - `malloc` 维护一个链表，记录所有空闲内存块的信息（地址、大小）。
  - 分配时，遍历链表找到足够大的块（如 首次适应、最佳适应 或 最差适应 算法）。
- 内存块结构:

代码块

```
struct mem_block
{
```

```
    size_t size;          // 块大小（含头部信息）
    int free;             // 是否空闲
    struct mem_block *next; // 指向下一个块
    // ... 可能还有其他元数据
};
```

- 实际返回给用户的地址是 `mem_block + 1`（跳过头部信息）。

3）分配流程

1. 检查空闲链表中是否有足够大的块：
   - 若有，分割或直接标记为已占用。
   - 若无，调用 `sbrk` / `mmap` 申请新内存并加入链表。
2. 返回可用内存的起始地址（跳过头部信息）。

4）释放流程（free）

1. 根据用户提供的指针，找到对应的 `mem_block` 头部。
2. 标记为空闲，尝试合并相邻的空闲块（减少碎片）。

5）内存碎片问题

- 外部碎片：空闲内存分散，无法满足大块请求。
  - 解决：合并相邻空闲块、使用 `mmap` 分配大块。
- 内部碎片：分配的内存比请求的大（因对齐或块最小大小限制）。
  - 例如，请求 5 字节，但分配 8 字节（对齐要求）。

6）优化策略

- Slab 分配器：针对小对象的高效分配（如 Linux 内核）。
- Thread-Local Cache：每个线程维护独立的内存池，减少锁竞争（如 glibc 的 `ptmalloc`）。

代码示例：

代码块

```
#include <unistd.h>

struct block_meta
{
    size_t size;
    struct block_meta *next;
    int free;
```

```
};

void *malloc(size_t size)
{
    struct block_meta *block;
    // 遍历空闲链表，寻找合适的块...
    if (找不到合适块)
    {
        // 调用 sbrk 申请新内存
        block = sbrk(sizeof(struct block_meta) + size);
        block->size = size;
        block->free = 0;
    }
    return (void*)(block + 1); // 跳过头部
}
```

总结

- `malloc` 的核心是内存池 + 空闲链表管理，通过算法高效分配/释放内存。
- 实际实现（如 glibc 的 `ptmalloc`）会更复杂，考虑多线程、碎片优化等问题。

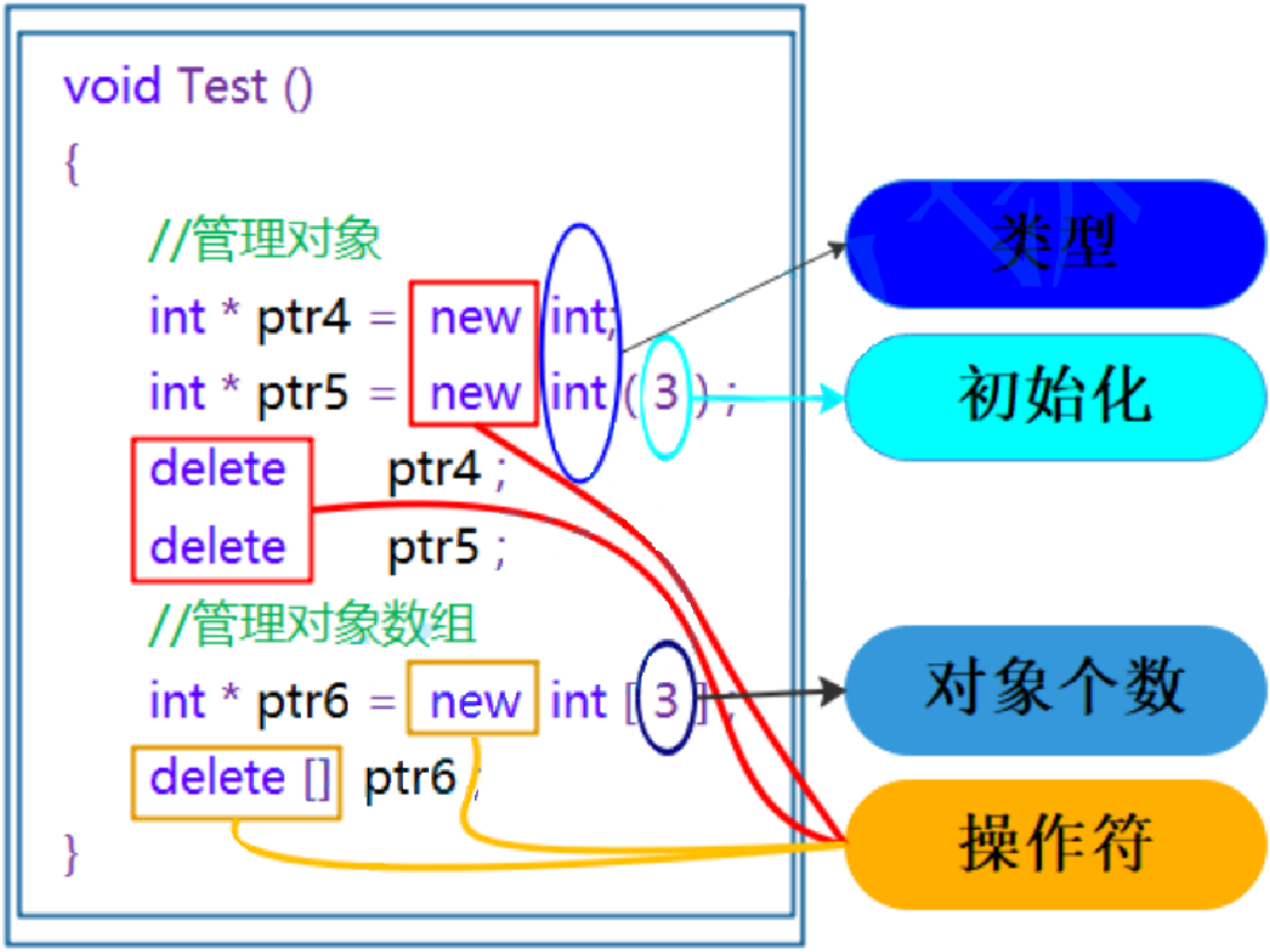
3. C++内存管理方式

> C语言内存管理方式在C++中可以继续使用，但有些地方就无能为力，而且使用起来比较麻烦，因此C++又提出了自己的内存管理方式：**通过new和delete操作符进行动态内存管理**。

3.1 new/delete操作内置类型

代码块

```
void Test()
{
    // 动态申请一个int类型的空间
    int* ptr4 = new int;
    // 动态申请一个int类型的空间并初始化为10
    int* ptr5 = new int(10);
    // 动态申请10个int类型的空间
    int* ptr6 = new int[3];
    delete ptr4;
    delete ptr5;
    delete[] ptr6;
}
```



注意：申请和释放单个元素的空间，使用new和delete操作符，申请和释放连续的空间，使用 new[] 和 delete[]，注意：匹配起来使用。

3.2 new和delete操作自定义类型

代码块

```
class A
{
public:
    A(int a = 0)
        : _a(a)
    {
        cout << "A():" << this << endl;
    }

    ~A()
    {
        cout << "~A():" << this << endl;
    }

private:
```

```
16      int _a;
17  };
18
19  int main()
20  {
21      // new/delete 和 malloc/free最大区别是 new/delete对于【自定义类型】除了开空间
22      // 还会调用构造函数和析构函数
23      A* p1 = (A*)malloc(sizeof(A));
24      A* p2 = new A(1);
25      free(p1);
26      delete p2;
27
28      // 内置类型是几乎是一样的
29      int* p3 = (int*)malloc(sizeof(int)); // C
30      int* p4 = new int;
31      free(p3);
32      delete p4;
33
34      A* p5 = (A*)malloc(sizeof(A)*10);
35      A* p6 = new A[10];
36      free(p5);
37      delete[] p6;
38      return 0;
39  }
```

注意：在申请自定义类型的空间时，new会调用构造函数，delete会调用析构函数，而malloc与free不会。

4. ※Operator new与operator delete函数

4.1 Operator new与operator delete函数(重点)

> **new和delete是用户进行动态内存申请和释放的操作符，** `operator new` **和** `operator delete` 是 系统提供的**全局函数，new在底层调用** `operator new` 全局函数来申请空间，**delete在底层通过** `operator delete` 全局函数来释放空间。

代码块

```
1  /*
2  operator new：该函数实际通过malloc来申请空间，当malloc申请空间成功时直接返回；申请空
   间
3  失败，尝试执行空 间不足应对措施，如果改应对措施用户设置了，则继续申请，否
4  则抛异常。
5  */
```

```
6   void *__CRTDECL operator new(size_t size) _THROW1(_STD bad_alloc)
7   {
8   // try to allocate size bytes
9   void *p;
10  while ((p = malloc(size)) == 0)
11      if (_callnewh(size) == 0)
12      {
13          // report no memory
14          // 如果申请内存失败了，这里会抛出bad_alloc 类型异常
15          static const std::bad_alloc nomem;
16          _RAISE(nomem);
17      }
18      return (p);
19  }
20  /* operator delete: 该函数最终是通过free来释放空间的 */
21
22  void operator delete(void *pUserData)
23  {
24      _CrtMemBlockHeader * pHead;
25      RTCCALLBACK(_RTC_Free_hook, (pUserData, 0));
26      if (pUserData == NULL) return;
27
28      _mlock(_HEAP_LOCK); /* block other threads */
29      __TRY
30
31          /* get a pointer to memory block header */
32          pHead = pHdr(pUserData);
33
34          /* verify block type */
35          _ASSERTE(_BLOCK_TYPE_IS_VALID(pHead->nBlockUse));
36
37          _free_dbg( pUserData, pHead->nBlockUse );
38
39      __FINALLY
40          _munlock(_HEAP_LOCK); /* release other threads */
41      __END_TRY_FINALLY
42      return;
43  }
44
45  /*
46  free的实现
47  */
48  #define free(p) _free_dbg(p, _NORMAL_BLOCK)
```

通过上述两个全局函数的实现知道，**operator new 实际也是通过malloc来申请空间**，如果malloc申请空间成功就直接返回，否则执行用户提供的空间不足应对措施，如果用户提供该措施就

继续申请，否则就抛异常。**operator delete 最终是通过free来释放空间的**。

5. new和delete的实现原理

5.1 内置类型

> 如果申请的是内置类型的空间，new和malloc，delete和free基本类似，不同的地方是：new/delete申请和释放的是单个元素的空间，new[]和delete[]申请的是连续空间，而且new在申请空间失败时会抛异常，malloc会返回NULL。

5.2 自定义类型

- **new的原理**

  调用`operator new`函数申请空间。

  在申请的空间上执行构造函数,完成对对象的构造。

- **delete的原理**

  在空间上执行析构函数，完成对象中资源的清理工作。

  调用`operator delete`函数释放对象的空间。

- **new T[N]的原理**

  调用`operator new[]`函数,在函数中实际调用`operator new`函数完成对N个对象空间的申请。

  在申请空间中执行N次构造函数。

- **delete[] 的原理**

  在释放的对象空间上执行N次析构函数，完成N个对象中资源的清理。

  调用`operator delete[]`释放空间，实际在`operator delete[]`中调用`operator delete`来释放空间。

6. 定位new表达式(placement-new)

> 定位new表达式是在**已分配的原始是内存空间中调用构造函数初始化一个 对象。它不分配内存，仅调用构造函数初始化对象。**
>
> **核心特点：**
>
> - 不分配内存，仅构造对象。
> - 显示指定对象的内存地址。
> - 需手动调用析构函数销毁对象。

> **使用格式：**
>
> new (place_address) type或者new (place_address) type(initializer-list)
>
> place_address必须是一个指针，initializer-list是类型的初始化列表

1. 基本语法

代码块

```
#include <new>  // 必须包含 <new> 头文件

void* memory = malloc(sizeof(MyClass));  // 预先分配内存
MyClass* obj = new (memory) MyClass();   // Placement new
```

- `memory`：已分配的内存地址（可以是堆、栈或静态存储区）。
- `MyClass()`：构造函数参数。

2. 为什么需要Placement new?

典型场景：

- 自定义内存管理：在内存池、对象池中复用内存块，避免频繁new/delete的开销。
- 非默认内存位置： 在共享内存、硬件寄存器或特定地址(如嵌入式系统)上构造对象。
- 避免额外拷贝： 直接在目标内存构造对象，省去临时对象拷贝(如 STL 容器 `emplace_back` 底层实现)。

3. 使用步骤与示例

1）分配内存

代码块

```
// 方式1：动态分配（堆）
void* buffer = malloc(sizeof(MyClass));

// 方式2：静态分配（栈）
alignas(MyClass) char buffer[sizeof(MyClass)];  // 需对齐
```

2） 构造对象(Placement new)

代码块

```
class MyClass
```

```
{
public:
    MyClass(int x) : data(x) {}
    ~MyClass() { std::cout << "Destructor\n"; }
    int data;
};

MyClass* obj = new (buffer) MyClass(42);  // 在 buffer 上构造对象
```

3） 手动调用析构函数

代码块

```
obj->~MyClass();  // 必须显式调用析构函数!
```

4） 释放内存

代码块

```
free(buffer);  // 若内存是动态分配的
```

4. 注意事项

1） 内存对齐：

- 确保预先分配的内存满足对象的对齐要求（如使用 `alignas`）。
- 错误示例：

代码块

```
char buffer[sizeof(MyClass)];  // 可能未对齐
new (buffer) MyClass();        // 未对齐内存导致未定义行为!
```

2） 生命周期管理：

- Placement new 构造的对象不会自动调用析构函数，必须手动调用。
- 错误示例：

代码块

```
delete obj;  // 错误! 内存不是通过普通 new 分配的!
```

3） 与普通new的区别：

表格

	特性	普通new	Placement
1	内存分配	是	否(需要先分配)
2	调用构造函数	是	是
3	内存释放	delete自动处理	需要手动管理内存和析构
4	使用场景	通用动态分配	自定义内存管理

4 条记录

5. 实际应用案例 (基于DeepSeek)

1) 内存池实现

代码块

```
class MemoryPool
{
public:
    void* allocate(size_t size)
    {
        return pool.get_free_block(size);  // 从池中获取空闲块
    }
    void deallocate(void* ptr)
    {
        pool.release_block(ptr);           // 释放回池中
    }
};

// 使用
MemoryPool pool;
void* mem = pool.allocate(sizeof(MyClass));
MyClass* obj = new (mem) MyClass();
obj->~MyClass();
pool.deallocate(mem);
```

2) STL 容器的 `allocator`

STL 容器的内存分配器（如 `std::allocator`）底层使用 Placement new 在预分配的内存上构造元素。

3) 嵌入式系统

代码块

```
volatile uint32_t* hardware_reg = reinterpret_cast<uint32_t*>(0x40000000);
new (hardware_reg) HardwareDriver();  // 在硬件寄存器地址构造驱动对象
```

6. 常见的问题

- **能否对同一内存块多次调用Placement new?**

  可以，但需要确保每次构造前手动调用析构函数，否则会导致资源泄露(如未释放前一个对象的资源)。

- **Placement new 会抛异常吗?**

  仅当构造函数抛出异常时才会抛出，内存分配本身不会失败(因为内存已预先分配)。

- **如何避免手动调用析构函数?**

  使用 RAII 封装(如智能指针自定义删除器)：

  代码块

  ```
  std::unique_ptr<MyClass, void(*)(MyClass*)> obj
  (
      new (buffer) MyClass(),
      [](MyClass* p) { p->~MyClass(); }
  );
  ```

7. 总结

- Placement new 用于在指定内存地址构造对象，不涉及内存分配。
- 适用场景：内存池、高性能自定义分配器、特殊硬件地址操作。
- 必须手动管理：析构函数调用和内存释放。
- 注意对齐：确保内存地址满足对象对齐要求

7. malloc/free和new/delete的区别

malloc/free和new/delete的共同点是：都是从堆上申请空间，并且需要用户手动释放。不同的地方是：

a. malloc和free是函数，new和delete是操作符。

b. malloc申请的空间不会初始化，new可以初始化。

c. malloc申请空间时，需要手动计算空间大小并传递，new只需在其后跟上空间的类型即可，如果是多个对象，[]中指定对象个数即可。

d. malloc的返回值为void*, 在使用时必须强转，new不需要，因为new后跟的是空间的类型 。

e. malloc申请空间失败时，返回的是NULL，因此使用时必须判空，new不需要，但是new需要捕获异常 。

f. 申请自定义类型对象时，malloc/free只会开辟空间，不会调用构造函数与析构函数，而new在申请空间后会调用构造函数完成对象的初始化，delete在释放空间前会调用析构函数完成空间中资源的清理释放。