Lesson05--C/C++内存管理

【本节目标】

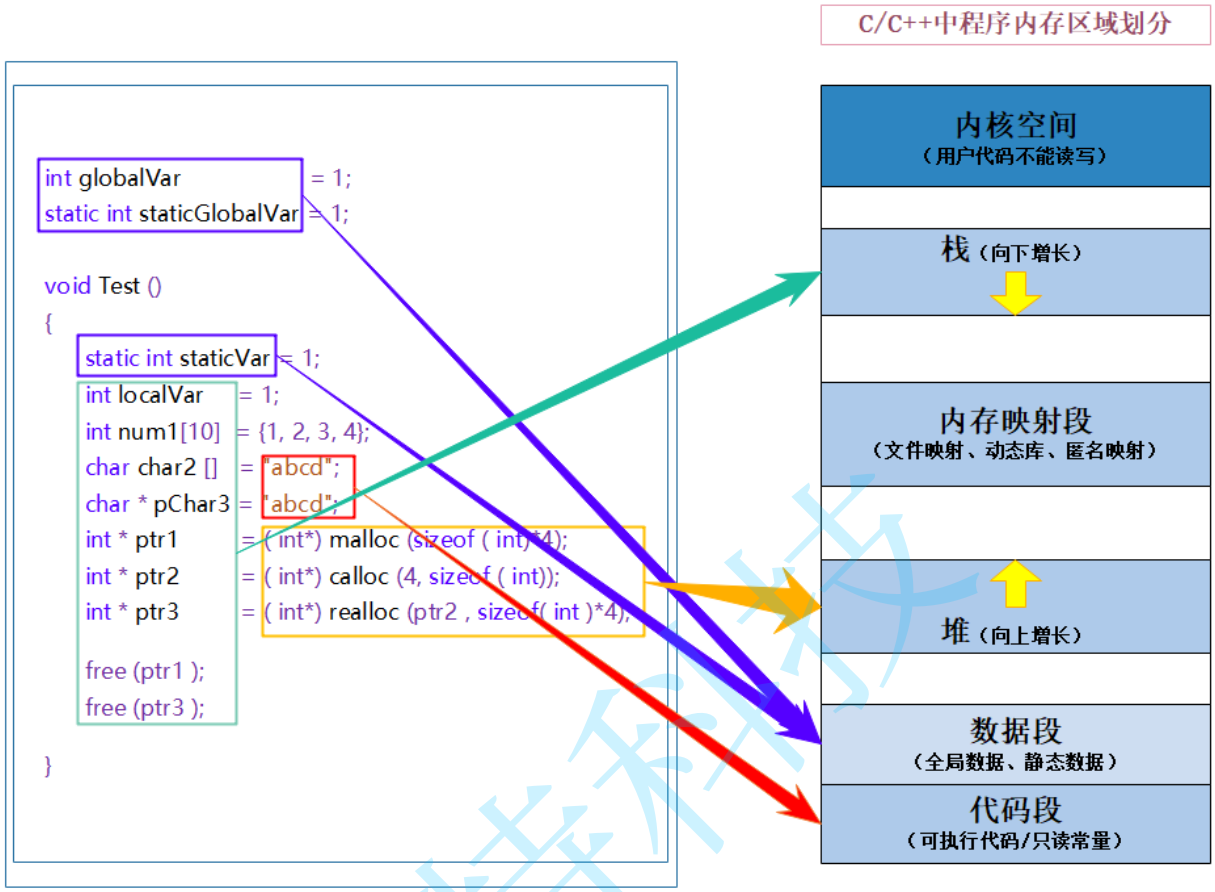
- 1. C/C++内存分布
- 2. C语言中动态内存管理方式
- 3. C++中动态内存管理
- 4. operator new与operator delete函数
- 5. new和delete的实现原理
- 6. 定位new表达式(placement-new)
- 7. 常见面试题

1. C/C++内存分布

> 我们先来看下面的一段代码和相关问题

```
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";
    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  D.代码段
   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? ____

2. 填空题:
   sizeof(num1) = ____;
```

```
sizeof(char2) = ____;       strlen(char2) = ____;
sizeof(pChar3) = ____;      strlen(pChar3) = ____;
sizeof(ptr1) = ____;
```



【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。（Linux课程如果没学到这块，现在只需要了解一下）
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语言中动态内存管理方式

2.1 malloc/calloc/realloc和free

```c
void Test ()
{
    int* p1 = (int*) malloc(sizeof(int));
    free(p1);

    // 1.malloc/calloc/realloc的区别是什么?
    int* p2 = (int*)calloc(4, sizeof (int));
    int* p3 = (int*)realloc(p2, sizeof(int)*10);

    // 这里需要free(p2)吗?
    free(p3 );
}
```

【面试题】

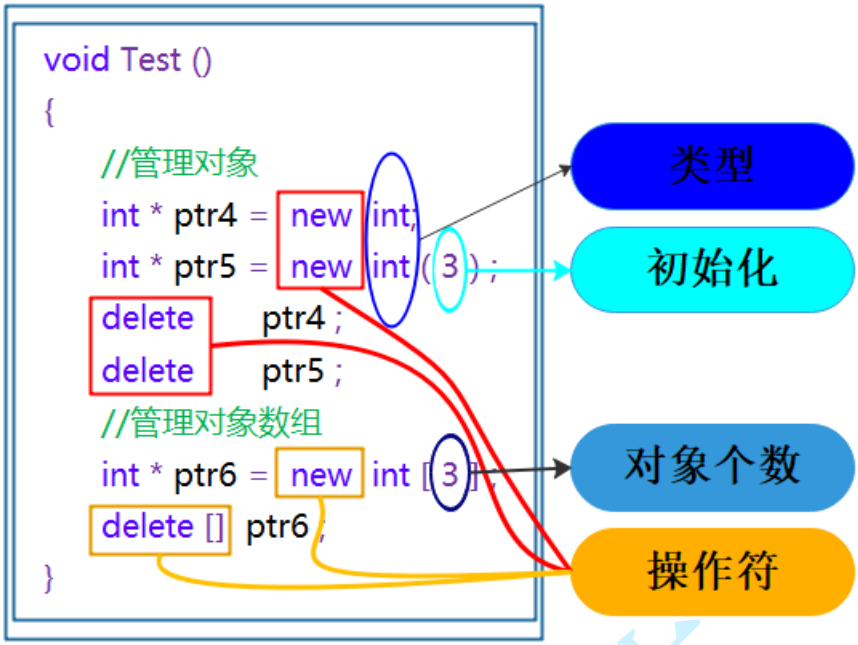
malloc/calloc/realloc的区别?

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

```c
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[]

3.2 new和delete操作自定义类型

```
class Test
{
public:
    Test()
        : _data(0)
    {
        cout<<"Test():"<<this<<endl;
    }

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

private:
    int _data;
};

void Test2()
{
    // 申请单个Test类型的空间
    Test* p1 = (Test*)malloc(sizeof(Test));
    free(p1);

    // 申请10个Test类型的空间
    Test* p2 = (Test*)malloc(sizoef(Test) * 10);
    free(p2);
}
```

```
void Test2()
{
    // 申请单个Test类型的对象
    Test* p1 = new Test;
    delete p1;

    // 申请10个Test类型的对象
    Test* p2 = new Test[10];
    delete[] p2;
}
```

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（重要点进行讲解）

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。

```
/*
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，否则抛异常。
*/
void *__CRTDECL operator new(size_t size) _THROW1(_STD bad_alloc)
{
    // try to allocate size bytes
    void *p;
    while ((p = malloc(size)) == 0)
        if (_callnewh(size) == 0)
        {
            // report no memory
            // 如果申请内存失败了，这里会抛出bad_alloc 类型异常
            static const std::bad_alloc nomem;
            _RAISE(nomem);
        }

    return (p);
}

/*
operator delete: 该函数最终是通过free来释放空间的
*/
void operator delete(void *pUserData)
{
        _CrtMemBlockHeader * pHead;

        RTCCALLBACK(_RTC_Free_hook, (pUserData, 0));


        if (pUserData == NULL)
```

```
            return;

        _mlock(_HEAP_LOCK);  /* block other threads */
        __TRY

            /* get a pointer to memory block header */
            pHead = pHdr(pUserData);

             /* verify block type */
            _ASSERTE(_BLOCK_TYPE_IS_VALID(pHead->nBlockUse));

            _free_dbg( pUserData, pHead->nBlockUse );

        __FINALLY
            _munlock(_HEAP_LOCK);  /* release other threads */
        __END_TRY_FINALLY

        return;
}

/*
free的实现
*/
#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。**

4.2 operator new与operator delete的类专属重载（了解）

下面代码演示了，针对链表的节点ListNode通过**重载类专属 operator new/ operator delete**，实现链表节点使用内存池申请和释放内存，提高效率。

```
struct ListNode
{
    ListNode* _next;
    ListNode* _prev;
    int _data;

    void* operator new(size_t n)
    {
        void* p = nullptr;
        p = allocator<ListNode>().allocate(1);
        cout << "memory pool allocate" << endl;
        return p;
    }

    void operator delete(void* p)
    {
        allocator<ListNode>().deallocate((ListNode*)p, 1);

        cout << "memory pool deallocate" << endl;
```

```
    }
};

class List
{
public:
    List()
    {
        _head = new ListNode;
        _head->_next = _head;
        _head->_prev = _head;
    }

    ~List()
    {
        ListNode* cur = _head->_next;
        while (cur != _head)
        {
            ListNode* next = cur->_next;
            delete cur;
            cur = next;
        }

        delete _head;
        _head = nullptr;
    }

private:
    ListNode* _head;
};

int main()
{
    List l;

    return 0;
}
```


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**
>
> 1. 调用operator new函数申请空间

2. 在申请的空间上执行构造函数，完成对象的构造

- **delete的原理**
  1. 在空间上执行析构函数，完成对象中资源的清理工作
  2. 调用operator delete函数释放对象的空间
- **new T[N]的原理**
  1. 调用operator new[]函数，在operator new[]中实际调用operator new函数完成N个对象空间的申请
  2. 在申请的空间上执行N次构造函数
- **delete[]的原理**
  1. 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
  2. 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) （了解）

定位new表达式是在**已分配的原始内存空间中调用构造函数初始化一个对象。**

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

使用场景：

定位new表达式在实际中一般是配合内存池使用。因为内存池分配出的内存没有初始化，所以如果是自定义类型的对象，需要使用new的定义表达式进行显示调构造函数进行初始化。

```
class Test
{
public:
   Test()
       : _data(0)
   {
       cout<<"Test():"<<this<<endl;
   }

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

private:
   int _data;
};

void Test()
{
   // pt现在指向的只不过是与Test对象相同大小的一段空间，还不能算是一个对象，因为构造函数没有执行

   Test* pt = (Test*)malloc(sizeof(Test));
```

```
    new(pt) Test;  // 注意：如果Test类的构造函数有参数时，此处需要传参
}
```

7. 常见面试题

7.1 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：
>
> 1. malloc和free是函数，new和delete是操作符
> 2. malloc申请的空间不会初始化，new可以初始化
> 3. 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
> 4. 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
> 5. 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
> 6. 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

7.2 请设计一个类，该类只能在堆上创建对象

> - 方法一：构造函数私有化
>
>   1. 将类的构造函数私有，拷贝构造声明成私有。防止别人调用拷贝在栈上生成对象。
>   2. 提供一个静态的成员函数，在该静态成员函数中完成堆对象的创建
>
>   ```
>   class HeapOnly
>   {
>   public:
>       static HeapOnly* CreateObject()
>       {
>           return new HeapOnly;
>       }
>   private:
>       HeapOnly() {}
>
>       // C++98
>       // 1.只声明,不实现。因为实现可能会很麻烦，而你本身不需要
>       // 2.声明成私有
>       HeapOnly(const HeapOnly&);
>
>       // or
>
>       // C++11
>       HeapOnly(const HeapOnly&) = delete;
>   };
>   ```

7.3 请设计一个类，该类只能在栈上创建对象

类似上面

```
class StackOnly
{
public:
    static StackOnly CreateObject()
    {
        return StackOnly();
    }
private:
    StackOnly()  {}
};
```

只能在栈上创建对象，即不能在堆上创建，因此只要将new的功能屏蔽掉即可，即屏蔽掉operator new和定位new表达式，注意：屏蔽了operator new，实际也将定位new屏蔽掉。

```
class StackOnly
{
public:
    StackOnly()  {}
private:
    void* operator new(size_t size);
    void operator delete(void* p);
};
```

7.4 单例模式（这个在面试中经常出现）

设计模式：

设计模式（Design Pattern）是一套**被反复使用、多数人知晓的、经过分类的、代码设计经验的总结**。为什么会产生设计模式这样的东西呢？就像人类历史发展会产生兵法。最开始部落之间打仗时都是人拼人的对砍。后来春秋战国时期，七国之间经常打仗，就发现打仗也是有**套路**的，后来孙子就总结出了《孙子兵法》。孙子兵法也是类似。

使用设计模式的目的：为了代码可重用性、让代码更容易被他人理解、保证代码可靠性。 设计模式使代码编写真正工程化；设计模式是软件工程的基石脉络，如同大厦的结构一样。

单例模式：

一个类只能创建一个对象，即单例模式，该模式可以保证系统中该类只有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。比如在某个服务器程序中，该服务器的配置信息存放在一个文件中，这些配置数据由一个单例对象统一读取，然后服务进程中的其他对象再通过这个单例对象获取这些配置信息，这种方式简化了在复杂环境下的配置管理。

单例模式有两种实现模式：

- **饿汉模式**

  就是说不管你将来用不用，程序启动时就创建一个唯一的实例对象。

```
// 饿汉模式
```

```
// 优点：简单
// 缺点：可能会导致进程启动慢，且如果有多个单例类对象实例启动顺序不确定。
 class Singleton
 {
 public:
     static Singleton* GetInstance()
     {
         return &m_instance;
     }

 private:
    // 构造函数私有
   Singleton(){};

   // C++98 防拷贝
   Singleton(Singleton const&);
   Singleton& operator=(Singleton const&);

   // or

   // C++11
   Singleton(Singleton const&) = delete;
   Singleton& operator=(Singleton const&) = delete;

   static Singleton m_instance;
 };

 Singleton Singleton::m_instance;  // 在程序入口之前就完成单例对象的初始化
```

如果这个单例对象在多线程高并发环境下频繁使用，性能要求较高，那么显然使用饿汉模式来避免资源竞争，提高响应速度更好。

- **懒汉模式**

  如果单例对象构造十分耗时或者占用很多资源，比如加载插件啊， 初始化网络连接啊，读取文件啊等等，而有可能该对象程序运行时不会用到，那么也要在程序一开始就进行初始化，就会导致程序启动时非常的缓慢。 所以这种情况使用懒汉模式（**延迟加载**）更好。

  ```
  // 懒汉
  // 优点：第一次使用实例对象时，创建对象。进程启动无负载。多个单例实例启动顺序自由控制。
  // 缺点：复杂

  #include <iostream>
  #include <mutex>
  #include <thread>
  using namespace std;

  class Singleton
  {
  public:
      static Singleton* GetInstance() {
          // 注意这里一定要使用Double-Check的方式加锁，才能保证效率和线程安全

          if (nullptr == m_pInstance) {
  ```

```
            m_mtx.lock();
            if (nullptr == m_pInstance) {
                m_pInstance = new Singleton();
            }
            m_mtx.unlock();
        }
        return m_pInstance;
    }

    // 实现一个内嵌垃圾回收类
    class CGarbo {
    public:
        ~CGarbo(){
            if (Singleton::m_pInstance)
                delete Singleton::m_pInstance;
        }
    };

    // 定义一个静态成员变量，程序结束时，系统会自动调用它的析构函数从而释放单例对象
    static CGarbo Garbo;

private:
    // 构造函数私有
    Singleton(){};

    // 防拷贝
    Singleton(Singleton const&);
    Singleton& operator=(Singleton const&);

    static Singleton* m_pInstance; // 单例对象指针
    static mutex m_mtx;            //互斥锁
};

Singleton* Singleton::m_pInstance = nullptr;
Singleton::CGarbo Garbo;
mutex Singleton::m_mtx;

void func(int n)
{
    cout<< Singleton::GetInstance() << endl;
}

// 多线程环境下演示上面GetInstance()加锁和不加锁的区别。
int main()
{
    thread t1(func, 10);
    thread t2(func, 10);

    t1.join();
    t2.join();

    cout << Singleton::GetInstance() << endl;

    cout << Singleton::GetInstance() << endl;
```

```
}
```

7.5 内存泄漏

7.5.1 什么是内存泄漏，内存泄漏的危害

> 什么是内存泄漏：内存泄漏指因为疏忽或错误造成程序未能释放已经不再使用的内存的情况。内存泄漏并不是指内存在物理上的消失，而是应用程序分配某段内存后，因为设计错误，失去了对该段内存的控制，因而造成了内存的浪费。
>
> 内存泄漏的危害：长期运行的程序出现内存泄漏，影响很大，如操作系统、后台服务等等，出现内存泄漏会导致响应越来越慢，最终卡死。

```
void MemoryLeaks()
{
     // 1.内存申请了忘记释放
    int* p1 = (int*)malloc(sizeof(int));
    int* p2 = new int;

    // 2.异常安全问题
    int* p3 = new int[10];

    Func(); // 这里Func函数抛异常导致 delete[] p3未执行，p3没被释放.

    delete[] p3;
}
```

7.5.2 内存泄漏分类（了解）

> C/C++程序中一般我们关心两种方面的内存泄漏：
>
> - **堆内存泄漏(Heap leak)**
>
>   堆内存指的是程序执行中依据须要分配通过malloc / calloc / realloc / new等从堆中分配的一块内存，用完后必须通过调用相应的 free或者delete 删掉。假设程序的设计错误导致这部分内存没有被释放，那么以后这部分空间将无法再被使用，就会产生Heap Leak。
> - **系统资源泄漏**
>
>   指程序使用系统分配的资源，比方套接字、文件描述符、管道等没有使用对应的函数释放掉，导致系统资源的浪费，严重可导致系统效能减少，系统执行不稳定。

7.5.2 如何检测内存泄漏（了解）

> - 在linux下内存泄漏检测：linux下几款内存泄漏检测工具
> - 在windows下使用第三方工具：VLD工具说明
> - 其他工具：内存泄漏工具比较

7.5.3如何避免内存泄漏

1. 工程前期良好的设计规范，养成良好的编码规范，申请的内存空间记着匹配的去释放。ps：这个理想状态。但是如果碰上异常时，就算注意释放了，还是可能会出问题。需要下一条智能指针来管理才有保证。
2. 采用RAII思想或者智能指针来管理资源。
3. 有些公司内部规范使用内部实现的私有内存管理库。这套库自带内存泄漏检测的功能选项。
4. 出问题了使用内存泄漏工具检测。ps：不过很多工具都不够靠谱，或者收费昂贵。

总结一下:

内存泄漏非常常见，解决方案分为两种：1、事前预防型。如智能指针等。2、事后查错型。如泄漏检测工具。

7.6 如何一次在堆上申请4G的内存?

```
// 将程序编译成x64的进程，运行下面的程序试试?
#include <iostream>
using namespace std;

int main()
{
    void* p = new char[0xffffffffu];
    cout << "new:" << p << endl;

    return 0;
}
```

【总结】

知识块	知识点	分类	掌握程度
C/C++内存分段	栈、内存映射段、堆、数据段、代码段、	考点型	掌握
复习C中动态内存管理	malloc、calloc、realloc/free	考点型	掌握
C++动态内存管理	new/delete , new[]/delete[]	概念型	掌握
	new操作符	应用型	掌握
	delete[] p. 如何知道调多少次析构函数	原理型	了解
	操作符new	概念型	熟悉
	定位new表达式	概念型	了解
	malloc/free和new/delete的区别和联系	考点型	掌握
特殊要求的类的设计	只能在堆上创建对象的类	应用型	掌握
	只能在栈上创建对象的类	应用型	掌握
	只能创建一个对象的类	应用型	掌握

【作业】

1. 写一篇博客，归纳整理本节的内容。
2. 请认真思考并完成课件中第7点得常见面试题。