5.C/C++内存管理

1. C/C++内存分布

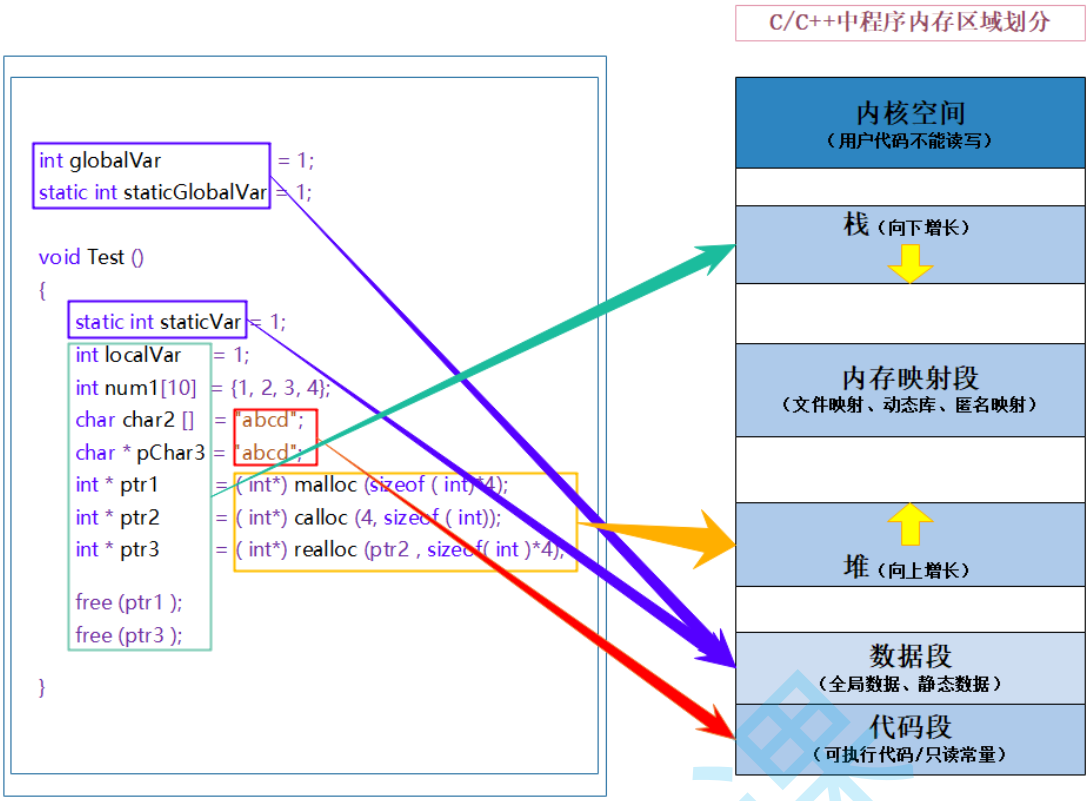
> 我们先来看下面的一段代码和相关问题

```
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";
    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？_____
```



【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：malloc/calloc/realloc/free

```
void Test ()
{
 // 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】

1. malloc/calloc/realloc的区别？
2. malloc的实现原理？ glibc中malloc实现原理

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。**

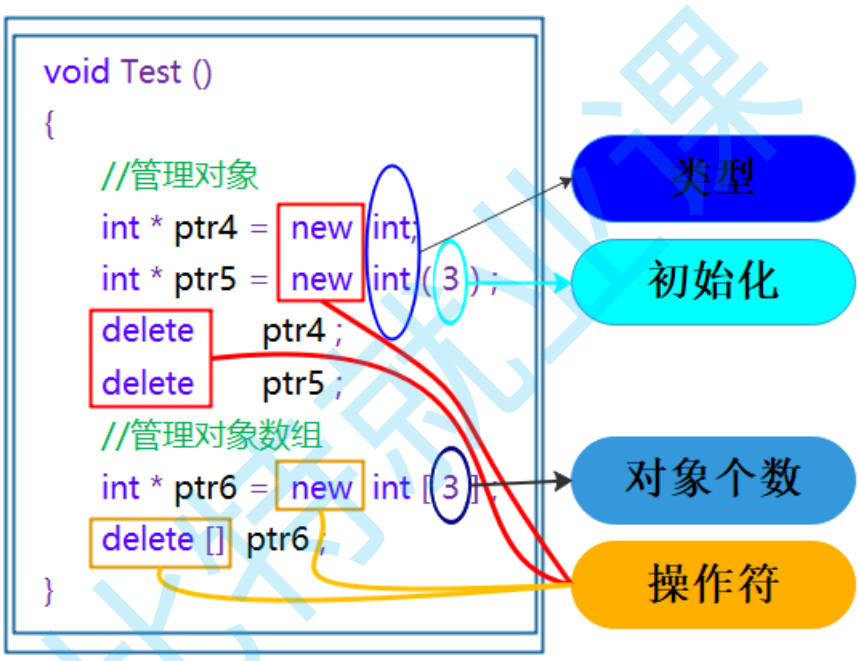
3.1 new/delete操作内置类型

```
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

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

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

private:
```

```
    int _a;
};

int main()
{
    // new/delete 和 malloc/free最大区别是 new/delete对于【自定义类型】除了开空间还会调用构造函数和析构函数
    A* p1 = (A*)malloc(sizeof(A));
    A* p2 = new A(1);
    free(p1);
    delete p2;

    // 内置类型是几乎是一样的
    int* p3 = (int*)malloc(sizeof(int)); // C
    int* p4 = new int;
 free(p3);
 delete p4;

    A* p5 = (A*)malloc(sizeof(A)*10);
    A* p6 = new A[10];
    free(p5);
    delete[] p6;

    return 0;
}
```

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。**

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

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

private:
    int _a;
};

// 定位new/replacement new
int main()
{
    // p1现在指向的只不过是与A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
    A* p1 = (A*)malloc(sizeof(A));
    new(p1)A;  // 注意：如果A类的构造函数有参数时，此处需要传参
    p1->~A();
    free(p1);
```

```
    A* p2 = (A*)operator new(sizeof(A));
    new(p2)A(10);
    p2->~A();
    operator delete(p2);
  return 0;
}
```

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：
>
> 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，如果是多个对象，[]中指定对象个数即可
> 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释放

######

######