

- 1、在main执行之前和之后执行的代码可能是什么
- 2、结构体内存对齐问题？
 - 2.1什么是内存对齐
 - 2.1为什么要进行内存对齐
 - 2.3内存对齐规则
- 3、指针和引用的区别
- 4、堆和栈的区别
- 5、区别以下指针类型？
- 6、基类的虚函数表存放在内存的什么区，虚表指针vptr的初始化时间
 - 6.1虚函数基础知识
 - 6.2代码分析
- 7、new / delete 与 malloc / free的异同
- 8、宏定义和函数有何区别？
- 9、malloc和new的区别？
 - 9.1 申请的内存所在位置
 - 9.2 返回类型安全性
 - 9.3 内存分配失败时的返回值
 - 9.4 是否需要指定内存大小
 - 9.5 是否调用构造函数/析构函数
 - 9.6 对数组的处理
 - 9.7 new与malloc是否可以相互调用
 - 9.8 是否可以被重载
 - 9.9 能够直观地重新分配内存
 - 9.10 客户处理内存分配不足
- 10、delete和delete[]区别？

1、在main执行之前和之后执行的代码可能是什么

main函数执行之前，主要就是初始化系统相关资源：

- 设置栈指针
- 初始化静态static变量和global全局变量，即.data段的内容
- 将未初始化部分的全局变量赋初值：数值型short, int, long等为0, bool为FALSE, 指针为NULL等等，即.bss段的内容
- 全局对象初始化，在main之前调用构造函数，这是可能会执行前的一些代码
- 将main函数的参数argc, argv等传递给main函数，然后才真正运行main函数

main函数执行之后：

- 全局对象的析构函数会在main函数之后执行；
- 可以用 atexit 注册一个函数，它会在main 之后执行；

2、结构体内存对齐问题？

2.1什么是内存对齐

- 还是用一个例子带出这个问题，看下面的小程序，理论上，32位系统下，int占4byte，char占一个byte，那么将它们放到一个结构体中应该占4+1=5byte；但是实际上，通过运行程序得到的结果是8 byte，这就是内存对齐所导致的。

```
//32位系统
#include<stdio.h>
struct{
    int x;
    char y;
}s;

int main()
{
    printf("%d\n",sizeof(s)); // 输出8
    return 0;
}
```

2.1为什么要进行内存对齐

- 尽管内存是以字节为单位，但是大部分处理器并不是按字节块来存取内存的.它一般会以双字节,四字节,8字节,16字节甚至32字节为单位来存取内存，我们将上述这些存取单位称为**内存存取粒度**.
- 现在考虑4字节存取粒度的处理器取int类型变量（32位系统），该处理器只能从地址为4的倍数的内存开始读取数据。
- 假如没有内存对齐机制，数据可以任意存放，现在一个int变量存放在从地址1开始的连续四个字节地址中，该处理器去取数据时，要先从0地址开始读取第一个4字节块,剔除不想要的字节（0地址）,然后从地址4开始读取下一个4字节块,同样剔除不要的数据（5，6，7地址）,最后留下的两块数据合并放入寄存器.这需要很多工作.
- 有了内存对齐的，int类型数据只能存放在按照对齐规则的内存中，比如说0地址开始的内存。那么现在该处理器在取数据时一次性就能将数据读出来了，而且不需要做额外的操作，提高了效率。

2.3内存对齐规则

- 每个特定平台上的编译器都有自己的默认“对齐系数”（也叫对齐模数）。gcc中默认#pragma pack(4)，可以通过预编译命令#pragma pack(n)，n = 1,2,4,8,16来改变这一系数。
- 有效对其值：是给定值#pragma pack(n)和结构体中最长数据类型长度中较小的那个。有效对齐值也叫对齐单位。

了解了上面的概念后，我们现在可以来看看内存对齐需要遵循的规则：

(1) 结构体第一个成员的偏移量（offset）为0，以后每个成员相对于结构体首地址的 offset 都是该成员大小与有效对齐值中较小那个的整数倍，如有需要编译器会在成员之间加上填充字节。

(2) 结构体的总大小为 有效对齐值 的整数倍，如有需要编译器会在最末一个成员之后加上填充字节。

下面给出几个例子以便于理解：

```
//32位系统
#include<stdio.h>
```

```

struct
{
    int i;
    char c1;
    char c2;
}x1;

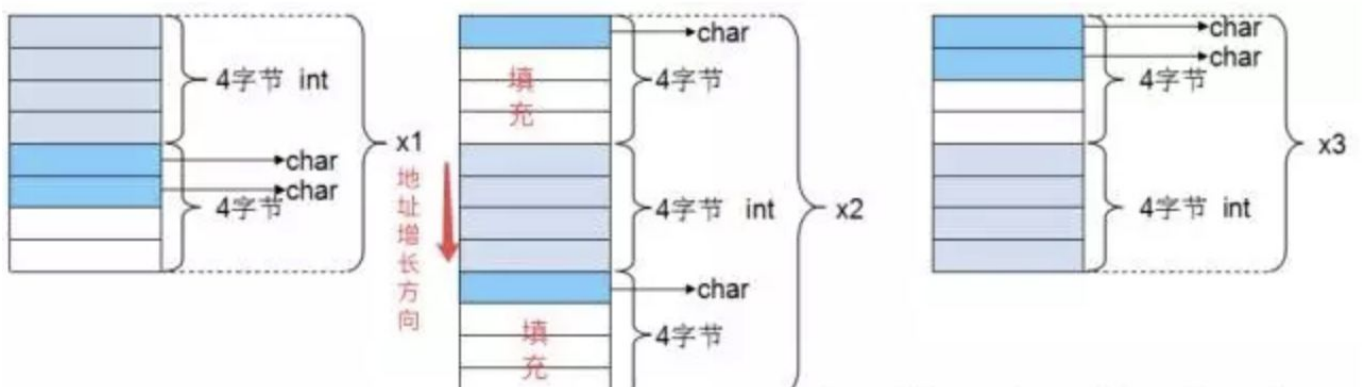
struct{
    char c1;
    int i;
    char c2;
}x2;

struct{
    char c1;
    char c2;
    int i;
}x3;

int main()
{
    printf("%d\n",sizeof(x1)); // 输出8
    printf("%d\n",sizeof(x2)); // 输出12
    printf("%d\n",sizeof(x3)); // 输出8
    return 0;
}

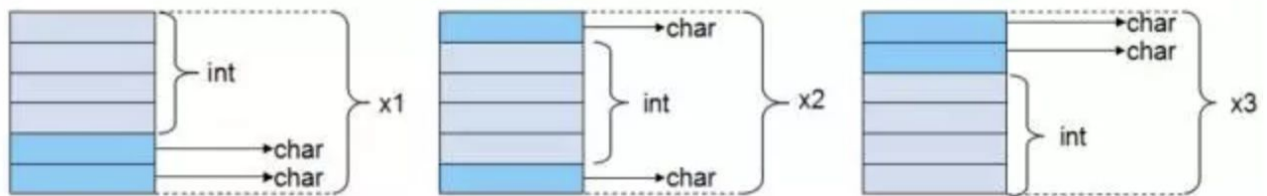
```

以上测试都是在Linux环境下进行的，linux下默认#pragma pack(3)，且结构体中最长的数据类型为4个字节，所以有效对齐单位为4字节，不难得出上面例子三个结构体的内存布局如下：

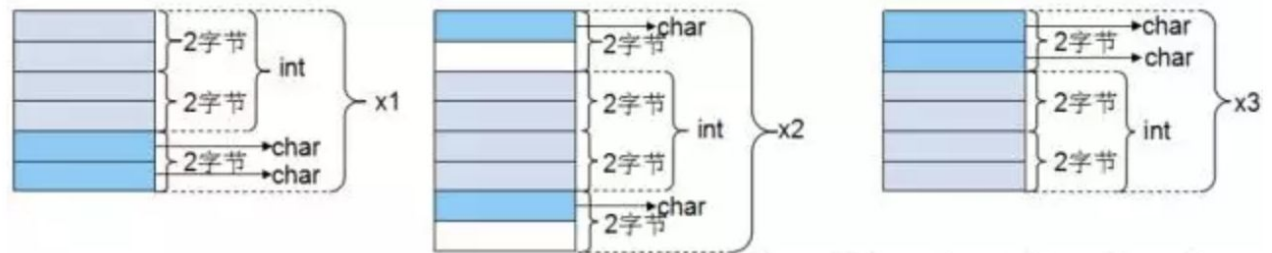


pragma pack(n)

- 不同平台上编译器的 pragma pack 默认值不同。而我们可以通过预编译命令#pragma pack(n), n=1,2,4,8,16来改变对齐系数。
- 例如，对于上个例子的三个结构体，如果前面加上#pragma pack(1)，那么此时有效对齐值为1字节，此时根据对齐规则，不难看出成员是连续存放的，三个结构体的大小都是6字节。



- 如果前面加上#pragma pack(2), 有效对齐值为2字节, 此时根据对齐规则, 三个结构体的大小应为6,8,6。内存分布图如下:
- 如果前面加上#pragma pack(2), 有效对齐值为2字节, 此时根据对齐规则, 三个结构体的大小应为6,8,6。内存分布图如下:



参考资料: [C/C++内存对齐详解](#)

3、指针和引用的区别

- 指针是一个变量, 存储的是一个地址, 引用跟原来的变量实质上是同一个东西, 是原变量的别名
- 指针可以有多级, 引用只有一级
- 指针可以为空, 但是引用不能为NULL且在定义时必须初始化
- 指针在初始化后可以改变指向, 而引用在初始化之后不可再改变
- sizeof指针得到的是本指针的大小, sizeof引用得到的是引用所指向变量的大小
- 当把指针作为参数进行传递时, 也是将实参的一个拷贝传递给形参, 两者指向的地址相同, 但不是同一个变量, 在函数中改变这个变量的指向不影响实参, 而引用却可以。
- 指针是具体变量, 需要占用存储空间。引用只是别名, 不占用具体存储空间, 只有声明没有定义;
- 指针声明和定义可以分开, 可以先只声明指针变量而不初始化, 等用到时再指向具体变量。引用在声明时必须初始化为另一变量, 一旦出现必须为typename refname &varname形式;
- 指针变量可以重新指向别的变量。引用一旦初始化之后就不可以再改变 (变量可以被引用为多次, 但引用只能作为一个变量引用);
- 但是存在指向空值的指针。不存在指向空值的引用, 必须有具体实体; 参考代码:

```
void test(int *p)
{
    int a=1;
    p=&a;
    cout<<p<<" "<<*p<<endl;
}
int main(void)
{
    int *p=NULL;
    test(p);
}
```

```
    if(p==NULL)
        cout<<"指针p为NULL"<<endl;
    return 0;
}
//运行结果为:
//0x22ff44 1
//指针p为NULL
void testPTR(int* p) {
    int a = 12;
    p = &a;
}
void testREFF(int& p) {
    int a = 12;
    p = a;
}
void main()
{
    int a = 10;
    int* b = &a;
    testPTR(b); //改变指针指向, 但是没改变指针的所指的内容
    cout << a << endl; // 10
    cout << *b << endl; // 10
    a = 10;
    testREFF(a);
    cout << a << endl; //12
}
```

4、堆和栈的区别

- **申请方式不同**: 栈由系统自动分配; 堆是自己申请和释放的。
- **申请大小限制不同**: 栈顶和栈底是之前预设好的, 栈是向栈底扩展, 大小固定, 可以通过- **申请效率不同**: 栈由系统分配, 速度快, 不会有碎片; 堆由程序员分配, 速度慢, 且会有碎片。 **形象的比喻**
- 栈就像我们去饭馆里吃饭, 只管点菜 (发出申请)、付钱、和吃 (使用), 吃饱了就走, 不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作, 他的好处是**快捷**, 但是**自由度小**。
- 堆就象是自己动手做喜欢吃的菜肴, 比较麻烦, 但是比较符合自己的口味, 而且自由度大。《C++中堆(heap)和栈(stack)的区别》: https://blog.csdn.net/qq_34175893/article/details/83502412

5、区别以下指针类型?

```
int *p[10] //指针数组
int (*p)[10] //数组指针
int *p(int) //函数声明
int (*p)(int) //函数指针
```

- `int *p[10]`表示**指针数组**，**强调数组概念**，是一个数组变量，数组大小为10，**数组内每个元素都是指向int类型的指针变量**。
- `int (*p)[10]`表示**数组指针**，**强调是指针**，只有一个变量，是指针类型，不过**指向的是一个int类型的数组**，这个数组大小是10。
- `int *p(int)`是**函数声明**，函数名是`p`，参数是`int`类型的，返回值是`int`类型的。
- `int (*p)(int)`是**函数指针**，**强调是指针**，该指针指向的函数具有`int`类型参数，并且返回值是`int`类型的。

6、基类的虚函数表存放在内存的什么区，虚表指针vptr的初始化时间

6.1虚函数基础知识

- C++中，一个类存在虚函数，那么编译器就会为这个类生成一个虚函数表，在虚函数表里存放的是这个类所有虚函数的地址。
- 当生成类对象的时候，编译器会自动的将**类对象的前四个字节设置为虚表的地址**，而这四个字节就可以看作是一个指向虚函数表的指针。**虚函数表可以看做一个函数指针数组**。

6.2代码分析

```
class Base
{
public:
    virtual void Hello()
    {
        cout << "Base Hello" << endl;
    }
};

class Derived:public Base
{
public:
    virtual void Hello()
    {
        cout << "Derived Hello" << endl;
    }
};

int main()
{
    //获取进程基址
    HANDLE hBase = GetModuleHandle(NULL);

    //基类
    Base* base = new Base();
    //获取虚函数表地址偏移
    DWORD baseVirtualTable = 0;
    memcpy(&baseVirtualTable, base, sizeof(DWORD));
    baseVirtualTable -= (DWORD)hBase;
    printf("base VirtualTable offset is 0x%08X\n", baseVirtualTable);
}
```

```

//派生类
Derived* derived= new Derived();
//获取虚函数表地址偏移
DWORD derivedVirtualTable = 0;
memcpy(&derivedVirtualTable, derived, sizeof(DWORD));
derivedVirtualTable -= (DWORD)hBase;
printf("derived VirtualTable is 0x%08X\n", derivedVirtualTable);

//基类指针指向子类对象
Base* pBaseToDerived = new Derived();

//获取虚函数表地址偏移
DWORD pBaseToDerivedVirtualTable = 0;
memcpy(&pBaseToDerivedVirtualTable, pBaseToDerived, sizeof(DWORD));
pBaseToDerivedVirtualTable -= (DWORD)hBase;
printf("pBaseToDerived VirtualTable is 0x%08X\n", pBaseToDerivedVirtualTable);
}

```

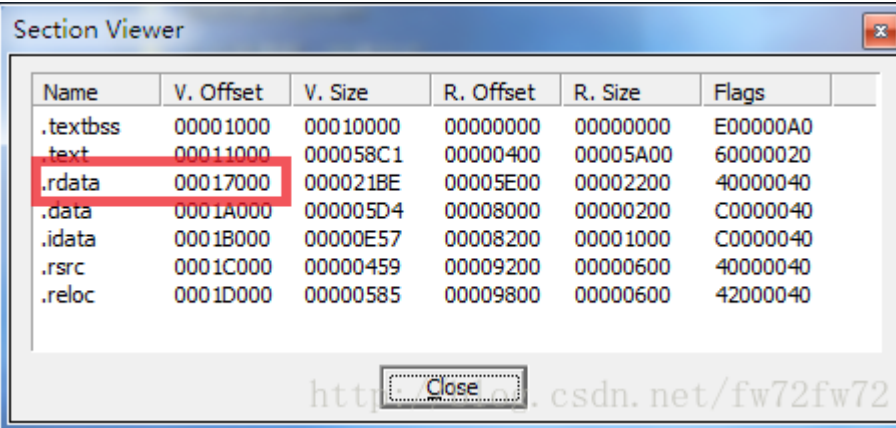
代码分别打印出，基类对象，子类对象，以及指向子类的基类指针的虚函数表相对于进程基址的偏移，结果如下图所示

```

base VirtualTable offset is 0x000183E8
derived VirtualTable is 0x00017834
pBaseToDerived VirtualTable is 0x00017834
请按任意键继续. /blog.csdn.net/fw72fw72

```

- 这里可以看出，**虚函数表是属于类，类的所有对象共享这个类的虚函数表。**并且，子类对象与指向子类的基类指针指向的对象，使用同一个虚函数表，符合C++的多态要求。
- 随后，使用PE工具，打开代码生成的exe文件，各个Section的偏移地址如下图



Name	V. Offset	V. Size	R. Offset	R. Size	Flags
.textbss	00001000	00010000	00000000	00000000	E00000A0
.text	00011000	000058C1	00000400	00005A00	60000020
.rdata	00017000	000021BE	00005E00	00002200	40000040
.data	0001A000	000005D4	00008000	00000200	C0000040
.idata	0001B000	00000E57	00008200	00001000	C0000040
.rsrc	0001C000	00000459	00009200	00000600	40000040
.reloc	0001D000	00000585	00009800	00000600	42000040

- 刚才虚函数表的相对偏移地址为0x000183E8和0x00017834，属于.rdata段。**由此可见，虚函数表存储在进程的只读数据段（.rdata），也就是C++内存模型中的常量区；而虚函数则位于代码段（.text），也就是C++内存模型中的代码区。**

结论:

1. 虚函数表属于类，类的所有对象共享这个类的虚函数表。

2. 虚函数表由编译器在编译时生成，保存在.rdata只读数据段,也就是C++内存模型中的常量区；而虚函数则位于代码段（.text），也就是C++内存模型中的代码区。

参考资料：

1. C++ 虚函数表存在哪

7、new / delete 与 malloc / free的异同

相同点

- 都可用于内存的动态申请和释放 不同点
- 前者是C++运算符，后者是C/C++语言标准库函数
- new自动计算要分配的空间大小，malloc需要手工计算
- new是类型安全的，malloc不是。例如：

```
int *p = new float[2]; //编译错误
int *p = (int*)malloc(2 * sizeof(double)); //编译无错误
```

- new调用名为operator new的标准库函数分配足够空间并调用相关对象的构造函数，delete对指针所指对象运行适当的析构函数；然后通过调用名为operator delete的标准库函数释放该对象所用内存。后者均没有相关调用
- 后者需要库文件支持，前者不用
- new是封装了malloc，直接free不会报错，但是这只是**释放内存，而不会析构对象**

8、宏定义和函数有何区别？

- 宏定义直接执行和函数跳转执行；宏在编译时完成替换，之后被替换的文本参与编译，相当于直接插入了代码，运行时不存在函数调用，执行起来更快；函数调用在运行时需要跳转到具体调用函数。
- 宏定义无返回值和函数有返回值；宏定义属于在结构中插入代码，没有返回值；函数调用具有返回值。
- 宏定义参数没有类型，不进行类型检查；函数参数具有类型，需要检查类型。
- 宏定义不要在最后加分号。

9、malloc和new的区别？

特征	new/delete	malloc/free
分配内存的位置	自由存储区	堆
内存分配失败返回值	完整类型指针	void*
内存分配失败返回值	默认抛出异常	返回NULL
分配内存的大小	由编译器根据类型计算得出	必须显式指定字节数

特征	new/delete	malloc/free
处理数组	有处理数组的new版本new[]	需要用户计算数组的大小后进行内存分配
已分配内存的扩充	无法直观地处理	使用realloc简单完成
是否相互调用	可以，看具体的operator new/delete实现	不可调用new
分配内存时内存不足	客户能够指定处理函数或重新制定分配器	无法通过用户代码进行处理
函数重载	允许	不允许
构造函数与析构函数	调用	不调用

9.1 申请的内存所在位置

- **new操作符从自由存储区（free store）上为对象动态分配内存空间，而malloc函数从堆上动态分配内存。**
- 自由存储区是C++基于new操作符的一个抽象概念，**凡是通过new操作符进行内存申请，该内存即为自由存储区。**
- 而堆是操作系统中的术语，是操作系统所维护的一块特殊内存，用于程序的内存动态分配，C语言使用malloc从堆上分配内存，使用free释放已分配的对应内存。
- 那么自由存储区是否能够是堆（问题等价于new是否能在堆上动态分配内存），这取决于operator new的实现细节。自由存储区不仅可以是堆，还可以是静态存储区，这都看operator new在哪里为对象分配内存。
- 特别的，new甚至可以不为对象分配内存！定位new的功能可以办到这一点：

```
new (place_address) type
```

place_address为一个指针，代表一块内存的地址。当使用上面这种仅以一个地址调用new操作符时，new操作符调用特殊的operator new，也就是下面这个版本：

```
void * operator new (size_t,void *) //不允许重定义这个版本的operator new
```

这个operator new不分配任何的内存，它只是简单地返回指针实参，然后由new表达式负责在place_address指定的地址进行对象的初始化工作。

malloc和free是标准库函数，支持覆盖；new和delete是运算符，并且支持重载。malloc仅仅分配内存空间，free仅仅回收空间，不具备调用构造函数和析构函数功能，用malloc分配空间存储类的对象存在风险；new和delete除了分配回收功能外，还会调用构造函数和析构函数。malloc和free返回的是void类型指针（必须进行类型转换），new和delete返回的是具体类型指针。

9.2 返回类型安全性

- new操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无须进行类型转换，故new是符合类型安全性的操作符。
- 而malloc内存分配成功则是返回void *，需要通过强制类型转换将void*指针转换成我们需要的类型。
- 类型安全很大程度上可以等价于内存安全，类型安全的代码不会试图修改自己没被授权的内存区域。

9.3 内存分配失败时的返回值

- new内存分配失败时，会抛出bad_alloc异常，它不会返回NULL；malloc分配内存失败时返回NULL。在使用C语言时，我们习惯在malloc分配内存后判断分配是否成功：

```
int *a = (int *)malloc ( sizeof (int ));
if(NULL == a)
{
    ...
}
else
{
    ...
}
```

从C语言走入C++阵营的新手可能会把这个习惯带入C++：

```
int * a = new int();
if(NULL == a)
{
    ...
}
else
{
    ...
}
```

- 实际上这样做一点意义也没有，因为new根本不会返回NULL，而且程序能够执行到if语句已经说明内存分配成功了，如果失败早就抛异常了。正确的做法应该是使用异常机制：

```
try
{
    int *a = new int();
}
catch (bad_alloc)
{
    ...
}
```

9.4 是否需要指定内存大小

- 使用new操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算，而malloc则需要显式地指出所需内存的尺寸。

```
class A{...}
A * ptr = new A;
A * ptr = (A *)malloc(sizeof(A)); //需要显式指定所需内存大小sizeof(A);
```

9.5 是否调用构造函数/析构函数

使用new操作符来分配对象内存时会经历三个步骤：

- 第一步：调用operator new 函数（对于数组是operator new[]）分配一块足够大的，原始的，未命名的内存空间以便存储特定类型的对象。
- 第二步：编译器运行相应的构造函数以构造对象，并为其传入初值。
- 第三步：对象构造完成后，返回一个指向该对象的指针。使用delete操作符来释放对象内存时会经历两个步骤：
 - 第一步：调用对象的析构函数。
 - 第二步：编译器调用operator delete(或operator delete[])函数释放内存空间。 **总而言之，new/delete会调用对象的构造函数/析构函数以完成对象的构造/析构。而malloc则不会。**

```
class A
{
public:
    A() :a(1), b(1.11){}
private:
    int a;
    double b;
};
int main()
{
    A * ptr = (A*)malloc(sizeof(A));
    return 0;
}
```

在return处设置断点，观看ptr所指内存的内容：

可以看出A的默认构造函数并没有被调用，因为数据成员a,b的值并没有得到初始化，这也是上面我为什么说使用malloc/free来处理C++的自定义类型不合适，其实不止自定义类型，标准库中凡是需要构造/析构的类型通通不合适。

而使用new来分配对象时：

```
int main()
{
    A * ptr = new A;
}
```

查看程序生成的汇编代码可以发现，A的默认构造函数被调用了：

9.6 对数组的处理

C++提供了new[]与delete[]来专门处理数组类型：

```
A * ptr = new A[10]; //分配10个A对象
```

使用new[]分配的内存必须使用delete[]进行释放：

```
delete [] ptr;
```

new对数组的支持体现在它会分别调用构造函数函数初始化每一个数组元素，释放对象时为每个对象调用析构函数。**注意delete[]要与new[]配套使用，不然会找出数组对象部分释放的现象，造成内存泄漏。**

至于malloc，它并不知道你在这块内存上要放的数组还是啥别的东西，反正它就给你一块原始的内存，在给你个内存的地址就完事。所以如果要动态分配一个数组的内存，还需要我们手动自定数组的大小：

```
int * ptr = (int *) malloc( sizeof(int) ); //分配一个10个int元素的数组
```

9.7 new与malloc是否可以相互调用

operator new /operator delete的实现可以基于malloc，而malloc的实现不可以去调用new。下面是编写operator new /operator delete 的一种简单方式，其他版本也与之类似：

```
void * operator new (size_t size)
{
    if(void * mem = malloc(size)
        return mem;
    else
        throw bad_alloc();
}
void operator delete(void *mem) noexcept
{
    free(mem);
}
```

9.8 是否可以被重载

operator new /operator delete可以被重载。标准库是定义了operator new函数和operator delete函数的8个重载版本：

```
//这些版本可能抛出异常
void * operator new(size_t);
void * operator new[](size_t);
void * operator delete (void *) noexcept;
void * operator delete[](void *) noexcept;
//这些版本承诺不抛出异常
void * operator new(size_t ,nothrow_t&) noexcept;
void * operator new[](size_t, nothrow_t& );
void * operator delete (void *,nothrow_t& )noexcept;
void * operator delete[](void *,nothrow_t& ) noexcept;
```

我们可以自定义上面函数版本中的任意一个，前提是自定义版本必须位于全局作用域或者类作用域中。太细节的东西不在这里讲述，总之，我们知道我们有足够的自由去重载operator new /operator delete ,以决定我们的new与delete如何为对象分配内存，如何回收对象。

而malloc/free并不允许重载。

9.9 能够直观地重新分配内存

- 使用malloc分配的内存后，如果在使用过程中发现内存不足，可以使用realloc函数进行内存重新分配实现内存的扩充。
- realloc先判断当前的指针所指内存是否有足够的连续空间，如果有，原地扩大可分配的内存地址，并且返回原来的地址指针；
- 如果空间不够，先按照新指定的大小分配空间，将原有数据从头到尾拷贝到新分配的内存区域，而后释放原来的内存区域。
- new没有这样直观的配套设施来扩充内存。

9.10 客户处理内存分配不足

在operator new抛出异常以反映一个未获得满足的需求之前，它会先调用一个用户指定的错误处理函数，这就是new-handler。new_handler是一个指针类型：

```
namespace std
{
    typedef void (*new_handler)();
}
```

指向了一个没有参数没有返回值的函数,即为错误处理函数。为了指定错误处理函数，客户需要调用set_new_handler，这是一个声明于的一个标准库函数：

```
namespace std
{
    new_handler set_new_handler(new_handler p ) throw();
}
```

set_new_handler的参数为new_handler指针，指向了operator new 无法分配足够内存时该调用的函数。其返回值也是个指针，指向set_new_handler被调用前正在执行（但马上就要发生替换）的那个new_handler函数。

意味着，对于new在内存分配不够时，能够返回指针指向出现异常的函数，客户能够指定处理函数或者重新制定分配器；而对于malloc，客户并不能够去编程决定内存不足以分配时要干什么事，只能看着malloc返回NULL。

参考资料: [c++ new 与malloc有什么区别](#)

10、delete和delete[]区别？

- delete只会调用一次析构函数。
- delete[]会调用数组中每个元素的析构函数。