C++内存管理

一些概念

内存对齐

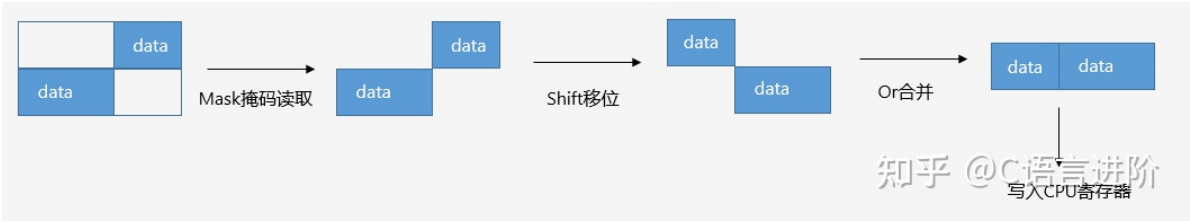
对于基础类型，如float, double, int, char等，它们的大小和内存占用是一致的。而对于结构体而言，如果我们取得其sizeof的结果，会发现这个值有可能会大于结构体内所有成员大小的总和，这是由于结构体内部成员进行了内存对齐。

为什么要进行内存对齐

内存对齐使数据读取更高效

在硬件设计上，数据读取的处理器只能从地址为k的倍数的内存处开始读取数据。这种读取方式相当于将内存分为了多个"块"，假设内存可以从任意位置开始存放的话，数据很可能会被分散到多个"块"中，处理分散在多个块中的数据需要移除首尾不需要的字节，再进行合并，非常耗时。

为了提高数据读取的效率，程序分配的内存并不是连续存储的，而是按首地址为k的倍数的方式存储；这样就可以一次性读取数据，而不需要额外的操作。



读取非对齐内存的过程示例 #### 内存对齐的规则

定义有效对齐值（alignment）为结构体中 最宽成员 和 编译器/用户指定对齐值 中较小的那个。

(1) 结构体起始地址为有效对齐值的整数倍

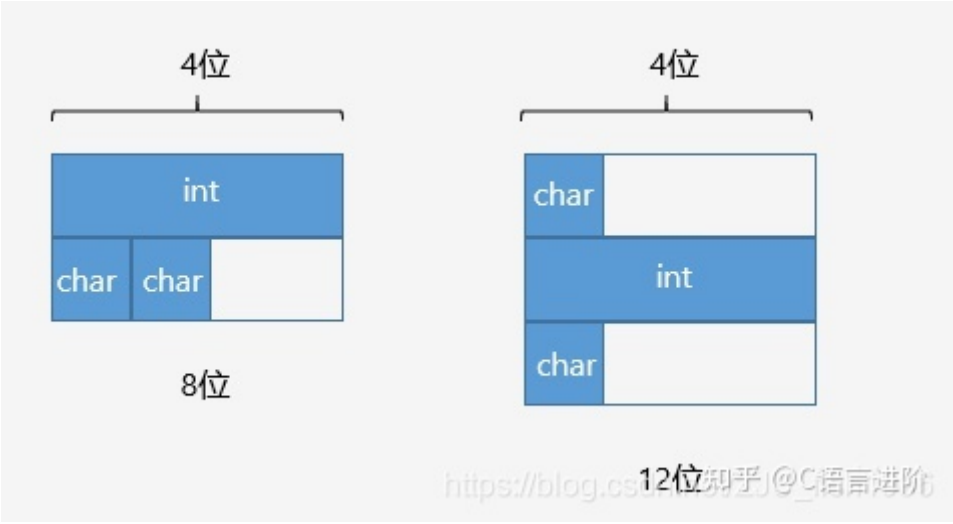
(2) 结构体总大小为有效对齐值的整数倍

(3) 结构体第一个成员偏移值为0，之后成员的偏移值为 min(有效对齐值, 自身大小) 的整数倍

相当于每个成员要进行对齐，并且整个结构体也需要进行对齐。

```
struct A
{
    int i;
    char c1;
    char c2;
};

int main()
{
    cout << sizeof(A) << endl; // 有效对齐值为4, output : 8
    return 0;
}
```

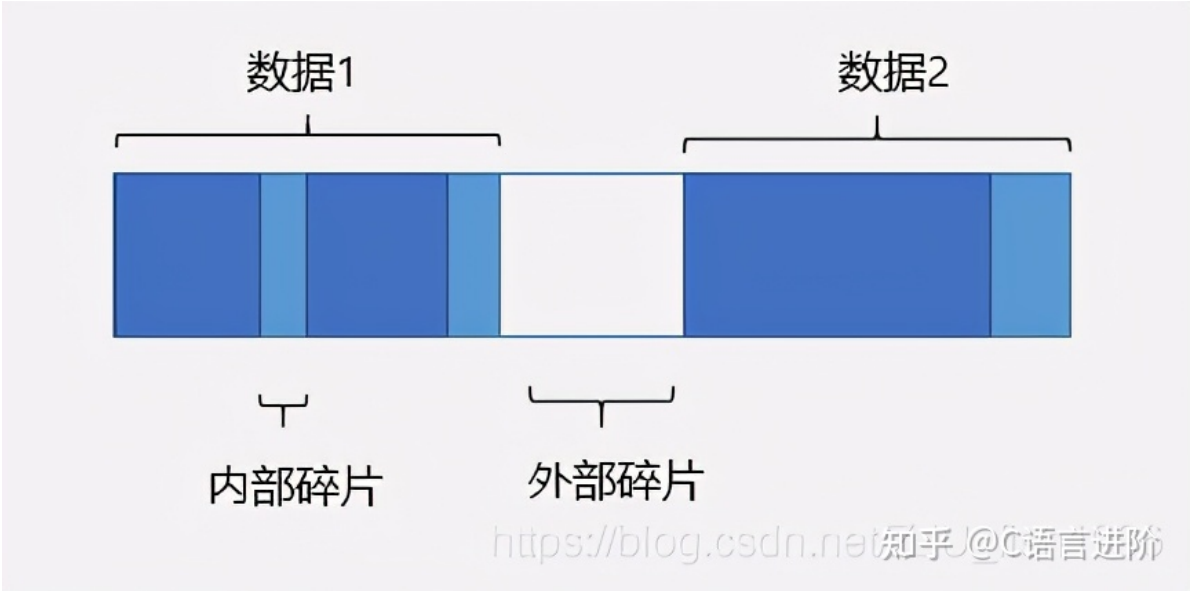


内存碎片

程序的内存往往不是紧凑连续排布的，而是存在着许多碎片。我们根据碎片产生的原因把碎片分为内部碎片和外部碎片两种类型：

(1) 内部碎片：系统分配的内存大于实际所需的内存（由于对齐机制）；

(2) 外部碎片：不断分配回收不同大小的内存，由于内存分布散乱，较大内存无法分配；



C 空间分配

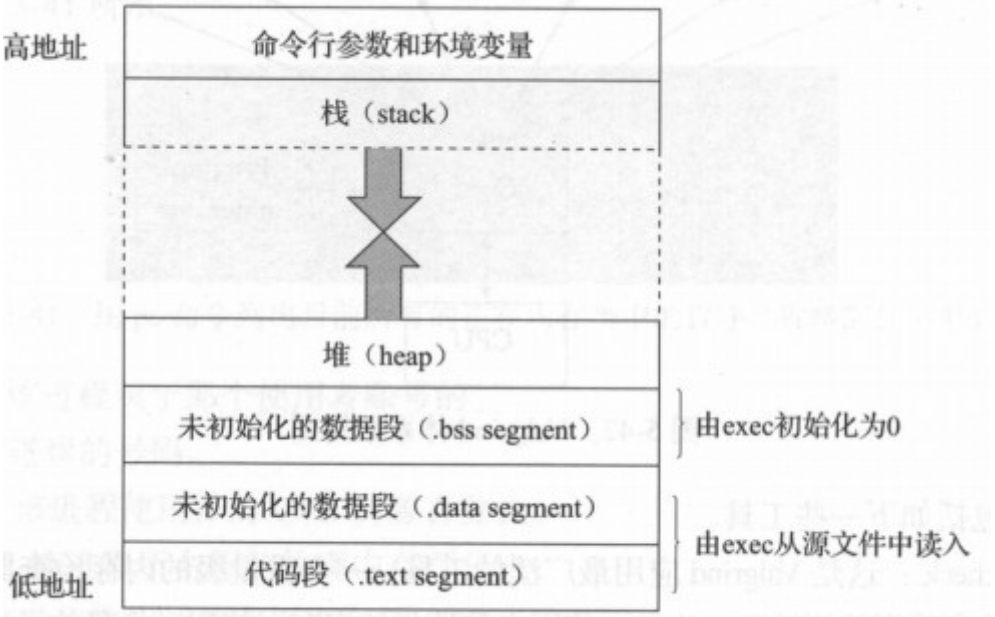


图 5-43　典型内存空间布局

代码段（.text segment）：代码段通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读，某些架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。程序段是程序代码在内存中的映射，一个程序可以在内存中有多个副本。
初始化数据段（.data segment）：通常是指用来存放程序中已初始化的全局变量的一块内存区域，例如，位于所有函数之外的全局变量：`int val=100`。需要强调的是，以上内容都是位于程序的可执行文件中，内核在调用 exec 函数启动该程序时从源程序文件中读人。数据段属于静态内存分配。
未初始化数据段（.bss segment）：通常是指用来存放程序中未初始化的全局变量的一块内存区域。BSS 是 Block Started by Symbol 的简称。
堆（heap）：堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态地扩张或缩减。当进程调用`malloc/free`等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）或释放的内存从堆中被剔除（堆被缩减）。
栈（stack）：栈存放程序的局部变量（但不包括 static 声明的变量，static意味着在数据段中存放变量）。除此以外，在函数被调用时，找用来传递参数和返回值。由于栈的先进后出特点，所以栈特别方便用来保存/恢复调用现场。而动态内存分配，需要程序员手工分配，手工释放。

C++ 空间分配

栈：在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。
堆：就是那些由`new`分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个`new`就要对应一个`delete`。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。
自由存储区：就是那些由`malloc`等分配的内存块，他和堆是十分相似的，不过它是用`free`来结束自己的生命的。
全局/静态存储区：全局变量和静态变量被分配到同一块内存中，在以前的C语言中，全局变量又分为初始化的和未初始化的，在C++里面没有这个区分了，他们共同占用同一块内存区。
常量存储区：这是一块比较特殊的存储区，他们里面存放的是常量，不允许修改。

堆和栈的差异

- 申请方式不同。

    - 栈： 由系统自动分配。 例如，声明在函数中一个局部变量 `int b`，系统自动在栈中为 b 开辟空间。
    - 堆：需要程序员自己申请，并指明大小，在 c 中用 malloc 函数，如 `p1= (char *)malloc(10)`;在 C++中用 new 运算符，如 `p2 = (char *)malloc(10)`。pl 、 p2 本身是在栈中的 。

      > 在栈内存中存放了一个指向一块堆内存的指针`p`。在程序会先确定在堆中分配内存的大小，然后调用`operator new`分配内存，然后返回这块内存的首地址，放入栈中

- 申请后系统的响应不同 。
    - 栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常，栈溢出。
    - 堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间中大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序 。 其次，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样代码中的 delete 语句才能正确的释放本内存空间。最后，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动地将多余的那部分重新放入空闲链表中 。
- 申请大小的限制不同 。
    - 栈：栈是向低地址扩展的数据结构，是一块连续的内存的区域。 这句话的意思是栈顶的地址和拢的最大容量是系统预先规定好的，在 Linux 下，栈的大小是一个常数（虽然可以设置，但它是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示Stack Overflow。 因此，能从栈获得的空间较小 。 用`ulimit -a`命令可以看到栈大小的限制，如图所示，其中 stack size 的值就是栈的大小了（本测试机是 8M ） 。 可以通过 ulimit -s 修改栈的大小。

      > **ulimit 功能说明**：控制shell程序的资源，查看用户所有限制值。

    - 堆：堆是向高地址扩展的数据结构，是不连续的内存区域 。 这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。 堆的大小受限于计算机系统中有效的虚拟内存 。 由此可见，堆获得的空间比较灵活，也比较大。
- 申请效率不同 。
    - 栈由系统向动分配，速度较快；但程序员是无法控制的 。
    - 堆是由 new 分配的内存， 一般速度比较慢，而且容易产生内存碎片；不过用起来最方便 。
- 堆和棋中的存储内容不同 。
    - 栈： 在函数调用时，第一个进梭的是主函数中后的下一条指令（ 函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的 C 编译器中，参数是由右往左入械的，然后是函数中的局部变量。 注意静态变量是不人拢的 。 当本次函数调用结束后，局部变量先出栈，然后是参数，最后找顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行 。
    - 堆： 一般是在堆的头部用一个字节存放堆的大小。 堆中的具体内容由程序员安排 。
- 碎片问题
    - 栈：频繁的`new/delete`势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。
    - 堆：不会存在这个问题，因为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出，在他弹出之前，在他上面的后进的栈内容已经被弹出
- 增长方向不同
    - 栈由高地址向低地址增长。
    - 堆由低地址向高地址增长。

delete 与 delete[] 的区别

c++ 中对 new 申请的内存的释放方式有 delete 和 delete[] 两种方式，到底这两者有什么区别呢?

- delete 释放 new 分配的单个对象指针指向的内存
- delete[] 释放 new 分配的对象数组指针指向的内存

1. 针对简单类型使用 new 分配后的不管是数组还是非数组形式内存空间用两种方式均可 如:

```
int *a = new int[10];
delete a;
delete[] a;
```

此种情况中的释放效果相同，原因在于：分配简单类型内存时，内存大小已经确定，系统可以记忆并且进行管理，在析构时，系统并不会调用析构函数。

它直接通过指针可以获取实际分配的内存空间，哪怕是一个数组内存空间(在分配过程中 系统会记录分配内存的大小等信息)。

1. 针对类 Class，两种方式体现出具体差异

当你通过下列方式分配一个类对象数组:

```
class A   {
private:
    char    *m_cBuffer;
    int     m_nLen;
public:
    A()  {  m_cBuffer = new char[m_nLen]; }
    ~A() {  delete [] m_cBuffer; }
};

A *a = new A[10];

// 仅释放了a指针指向的全部内存空间 但是只调用了a[0]对象的析构函数
// 剩下的从a[1]到a[9]这9个用户自行分配的m_cBuffer对应内存空间将不能释放,造成内存泄漏
delete a;

// 调用使用类对象的析构函数释放用户自己分配内存空间并且释放了a指针指向的全部内存空间
delete[] a;
```

所以总结下就是，如果 ptr 代表一个用 new 申请的内存返回的内存空间地址，即所谓的指针，那么:

- delete ptr 代表用来释放内存，且只用来释放 ptr 指向的内存。
- delete[] rg 用来释放 rg 指向的内存，！！还逐一调用数组中每个对象的 destructor！！

对于像 int/char/long/int*/struct 等等简单数据类型，由于对象没有 destructor ，所以用 delete 和 delete []是一样的！但是如果是 C++ 对象数组就不同了！

- 为基本数据类型分配和回收空间；delete[] 和 delete 是等同。
- 为自定义类型分配和回收空间；delete[] 和 delete 是不同。

杜绝“野指针”

“野指针”不是 `NULL` 指针，是指向“垃圾”内存的指针。人们一般不会错用 `NULL` 指针，因为用 `if` 语句很容易判断。但是“野指针”是很危险的，`if` 语句对它不起作用。 “野指针”的成因主要有三种:

- 指针变量没有被初始化。任何指针变量刚被创建时不会自动成为NULL指针，它的缺省值是随机的，它会乱指一气。所以，指针变量在创建的同时应当被初始化，要么将指针设置为NULL，要么让它指向合法的内存。例如：

```
char *p = NULL;
char *str = (char *) malloc(100);
```

- 指针p被free或者delete之后，没有置为NULL，让人误以为p是个合法的指针。

为什么需要new/delete

`malloc`与`free`是C++/C语言的标准库函数，`new/delete`是C++的运算符。它们都可用于申请动态内存和释放内存。

对于非内部数据类型的对象而言，光用`maloc/free`无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于`malloc/free`是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于`malloc/free`。
因此C++语言需要一个能完成动态内存分配和初始化工作的运算符`new`，以及一个能完成清理与释放内存工作的运算符`delete`。注意`new/delete`不是库函数。

我们先看一看`malloc/free`和`new/delete`如何实现对象的动态内存管理，见示例：

```
class Obj{
    public :
   Obj(void){ cout << “Initialization” << endl; }
   ~Obj(void){ cout << “Destroy” << endl; }
   void Initialize(void){ cout << “Initialization” << endl; }
   void Destroy(void){ cout << “Destroy” << endl; }
};

void UseMallocFree(void){
    Obj *a = (obj *)malloc(sizeof(obj)); // 申请动态内存
    a->Initialize(); // 初始化
    //…

    a->Destroy(); // 清除工作
    free(a); // 释放内存
}
void UseNewDelete(void){
    Obj *a = new Obj; // 申请动态内存并且初始化
    //…

    delete a; // 清除并且释放内存
}
```

类`Obj`的函数`Initialize`模拟了构造函数的功能，函数`Destroy`模拟了析构函数的功能。函数`UseMallocFree`中，由于`malloc/free`不能执行构造函数与析构函数，必须调用成员函数`Initialize`和`Destroy`来完成初始化与清除工作。函数`UseNewDelete`则简单得多。

所以我们不要企图用`malloc/free`来完成动态对象的内存管理，应该用`new/delete`。由于内部数据类型的“对象”没有构造与析构的过程，对它们而言`malloc/free`和`new/delete`是等价的。

既然`new/delete`的功能完全覆盖了`malloc/free`，为什么C++不把`malloc/free`淘汰出局呢？这是因为C++程序经常要调用C函数，而C程序只能用`malloc/free`管理动态内存。

如果用`free`释放“new创建的动态对象”，那么该对象因无法执行析构函数而可能导致程序出错。如果用`delete`释放“malloc申请的动态内存”，结果也会导致程序出错，但是该程序的可读性很差。所以`new/delete`必须配对使用，`malloc/free`也一样。