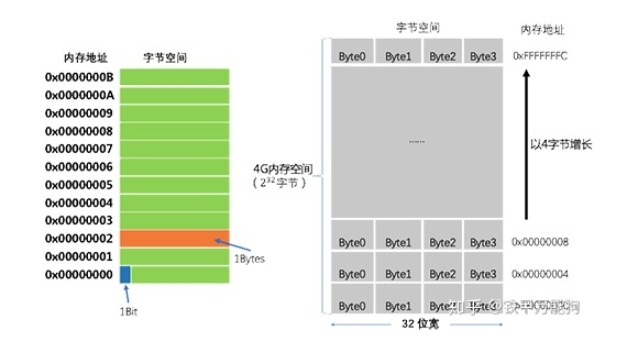
C++内存管理

在网上看到一篇不错的讲内存管理的帖子，整理了一下，供大家一起学习。

## ****前言****

RAM是计算机的主要或主要内存。 它是执行程序时存储文本，数据，指令和中间结果的地方。 总存储器被组织成字节数，每个字节再次被分成8位。 位是存储器中的最小单元，这些位是将数据存储为1和0的位置，称为二进制数据。 在存储器中，每个字节用一个称为地址的数字标识。 它总是一个正数。



## ****C/C++ 结构体及其数组的内存对齐****

内存对齐也叫字节对齐(data aligment):就是数据对象的内存大小可以被2的N次方的整数整除,也就是说字节对齐可以用某个2的N次方的整数去对齐.

目前计算机的32位的CPU可以在每个时刻周期从内存读取4个字节并填充数据总线,而64位的CPU每个时刻周期可以读取8个字节,而C语言的的设计者为了遵循CPU的这种特性。就给C编译器，当然后来的C++编译器也继承这一特性，在对C/C++源码编译的时候会对源码中的数据对象自动执行内存对齐操作(对数据对象之前填充一些没用的字节块)。

为什么要内存对齐?

因为我们要访问物理内存并能够在一次访问中获取整个数据，所以想象以下我们要从内存中读取一个4字节的int，而前两个字节在内存中的一个字中，而后两个字节在另外一个字中。我们不想分两次读取两个字然后将字节读取的字节再次装拼成一个整数，这是一种低效的内存访问。

### ****内存对齐规律****

**一般来说，对于需要X字节的原始数据类型，地址必须是X的倍数.**

**struct的起始地址决于其成员变量中的数据类型的sizeof()最大值作为对齐条件。**

**编译器在编译阶段对struct中未使用的内存空间执行填充操作,以确保字段对齐。**

**char类型不需要对齐,可自由分配任意可用的1字节尺寸的内存空间之中.**

**基本数据类型的对齐要求**

在不同的硬件架构和OS上执行的内存对齐是不一样,我们下面有个表,



需要特别指出的是

* 对于char类型没有任何对齐要求。
* 对于double类型即便是IA32的硬件架构,事实上可以通过gcc编译的时候指定命令行选项-malign-double,double类型也会以8字节对齐而不是4字节对齐。  
  从这个表可知,不同类型的CPU,对齐操作是不一样的.

**x86\_64环境下的对齐示例**

以下示例的左侧的数字表示内存地址,为了一目了然,使用十进制表示,并且在IA32 Windows 或 x86\_64环境下，这个示例主要用来解析上面的对齐规则。

**struct** foo{

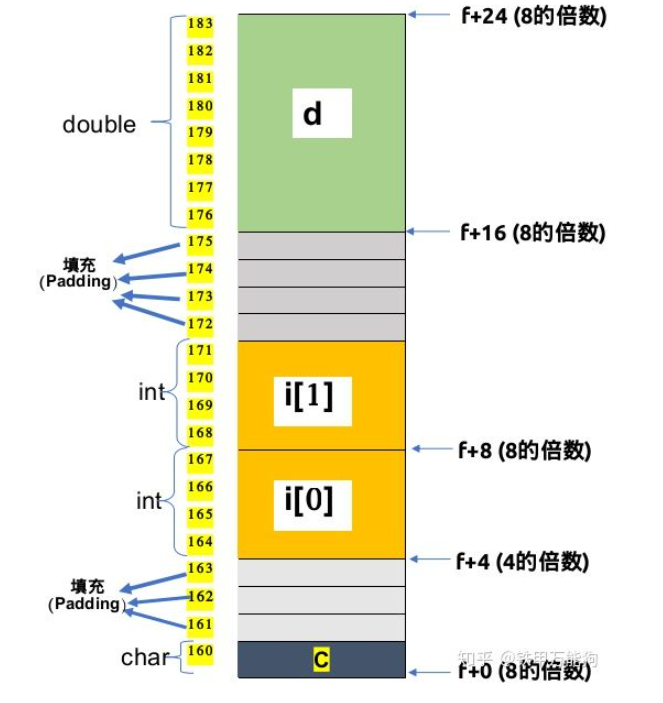
**char** c;

**int** [2];

**double** d;

}

* 首先,在结构体内部,必须满足每个成员的对齐条件
* 纵观整个struct的成员变量
  + 每个结构体都由一个对齐条件整数K，K即是结构体中所有成员变量中的类型尺寸的最大值。
  + 起始地址和结构体的长度必须是整数K的倍数。



那么该结构体的对齐条件整数K是多少？

* 根据规则1和2:由于这个结构体的对齐条件是8字节,因为它是struct成员变量中对齐条件最大是8个字节,所以这个结构体的起始地址必须是8的倍数,从低位r地址算起当然是地址160.
* 根据规则4:由于char类型是没有任何对齐条件的限制,并且在结构体中第一个声明的变量,所以它就落在结构体的起始地址,也就是地址160的位置.
* 根据规则1,由于int数组类型的每个元素占用4个字节,那么int类型的起始地址以4的倍数开始的那么就自然落到了164这个地址,紧挨着的i[2]元素自然落在168这个地址.
* 成员变量c和int[0]之间有还有未使用的内存空间,根据规则3,会填充未使用的字节.
* 根据规则1,Double类型的成员以8的倍数对齐,自然落在自struct起始地址起第16个字节的位置,即地址176的地方算起占用8个字节表示成员变量d的数据.
* 根据规则3,另外编译器会填充成员变量d和数组元素int[1]之间未使用的内存位置.

### ****节省内存空间****

从上面的例子我们得知,虽然内存对齐操作有助于优化CPU对内存的访问,但会带来一下副作用，就是会浪费一些内存空间.但这个问题不能全赖在编译器身上，作为程序员不良的写码习惯也是很大关系滴!!

我们在看看下面的示例从下图可以得知**结构体内部成员变量声明的先后顺序和编译器对内存对齐后,结构体所占的内存空间有很大的关系**.

* 下图左手边的内存布局是x86\_64架构下对应的C代码

**struct**{

**char** c;

**int** i[2];

**double** d;

} **\***f;

* 下图右手边的内存布局是x86\_64架构下对应的C代码

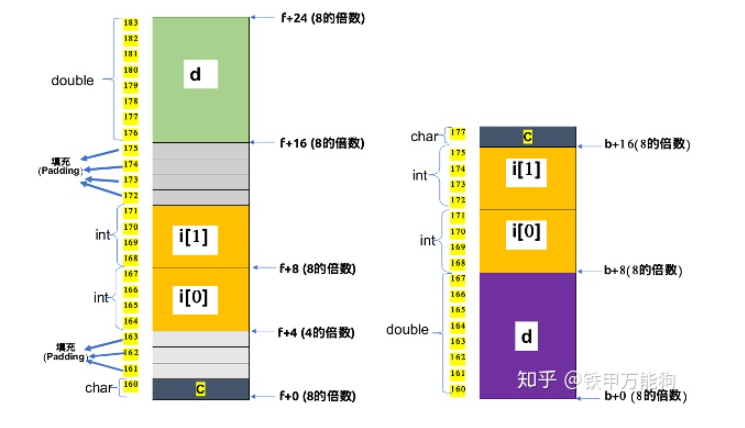
**struct**{

**double** d;

**int** i[2];

**char** c;

} **\***b;



由上面的的内存布局对比,我们得到得到一个基本结论:  
**在struct内部,将成员变量按照其类型的sizeof()值由大到小,依存声明的话能够不仅可以最大限度地减少编译器填充未使用内存块的操作,而且填充的内存块出现的次数越少,那么CPU每次从对应内存块中加载数据到寄存器中执行shift运算的次数也会相应地减少。同时能够兼顾CPU对对齐内存的访问.**

有读者对我上面的结果提出疑问，无论在32位还是64位计算机上，sizeof(double)始终为8个字节。 不同之处在于在32位Linux系统上，double对齐倍数是4字节数据类型一样。 要将double对齐的倍数为8个字节，请使用-malign-double（编译时选项)。

### ****结构体数组的内存分布****

灰常不幸的是!!，结构体的数组是无法满足上面的所讲的节省内存的特性的。

**typedef** **struct**{

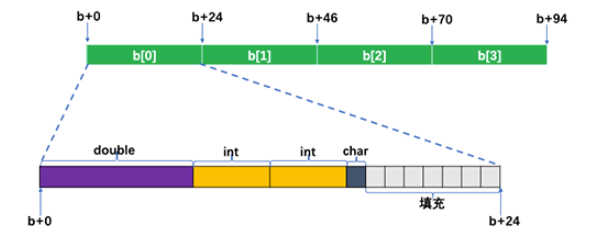
**double** d;

**int** i[2];

**char** c;

} b;

b[4];

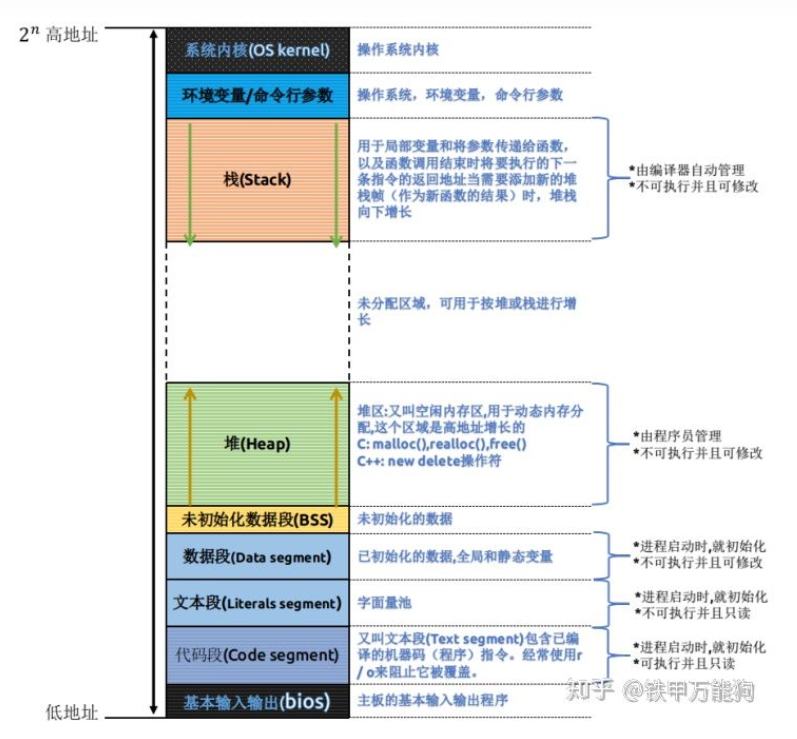


我们已经知道结构体b的对齐条件是8的倍数,由于结构体b的占据17个字节的内存空间,因此和它相关的数组中的每个元素占用内存空间必须要达到24个字节,才能达成每个元素的对齐条件是8的倍数, 因此每个结构体元素,编译器还需要为每个元素填充7个字节

## ****C/C++ 内存布局与程序栈****

### ****C/C++程序（进程）的内存寻址模型****

另外,像其他任何程序一样，BIOS和OS都需要内存(废话)，此处为了表示计算机内存模型的完整性。我们都把所有内存相关的内存区域都一一列出了,对于程序员感兴趣的主要内存区域是代码段，数据段和字面量池和bss，堆栈和堆。



从程序的组织的方式来查看程序的内存布局

* 代码段:程序的所有指令会存放在这个区域，这是已经编译后的机器码。
* 字面量池是程序初始化时的一些字符串字面量，在程序中用于显示文字
* 全局数据段：程序初始化时的常量和全局/静态的变量。C/C++ 用global/static声明的变量都存放在这个区域，对所有函数公开可见。
* 堆:这里保存的数据只是为了临时存储一些值而创建的，而我们可能在程序运行过程中可能会回收此内存。因为我们在程序执行期间不需要很长时间，所以使用C中的new或malloc这类内存分配程序来为我们所需的特定数据类型提供新的空间，并且随着我们要求越来越多的动态数据空间而该区域不断扩大，并且在内存中逐渐增长到更高的地址。
* 栈:当我们执行这些过程调用时,堆的基本特性是LIFO,存储着该程序“上下文”，它将从内存的高层地址开始，然后向另一个方向向下扩展。上下文其实就是程序中各个函数之间调用的先后顺序。

### ****EIP、EBP、ESP的作用****

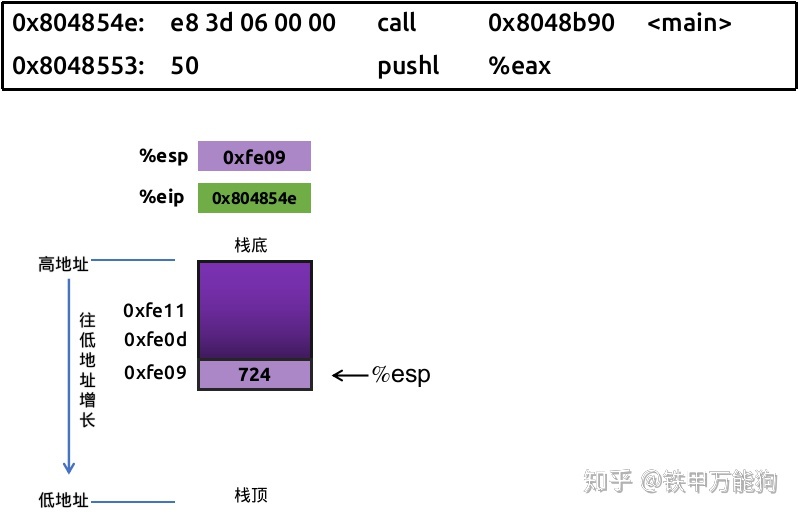
EIP存储着下一条指令的地址，每执行一条指令，该寄存器变化一次。

EBP存储着当前函数栈底的地址，栈低通常作为基址，我们可以通过栈底地址和偏移相加减来获取变量地址（很重要）。

ESP就是前面说的，始终指向栈顶，只要ESP指向变了，那么当前栈顶就变了。

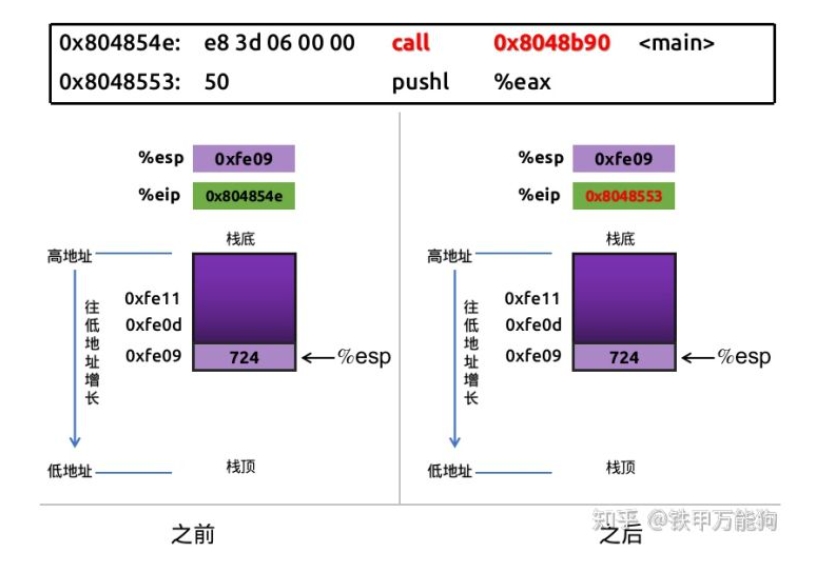
### ****程序栈的执行流****

从前文的一个粗略的例子，我们已经知道**调用者**调用**被调用者**会用到call指令,被调用者在执行结束时以ret指令返回，我们这里将进一步说明**栈**如何支援call和return指令。以下是一段反编译后的代码段



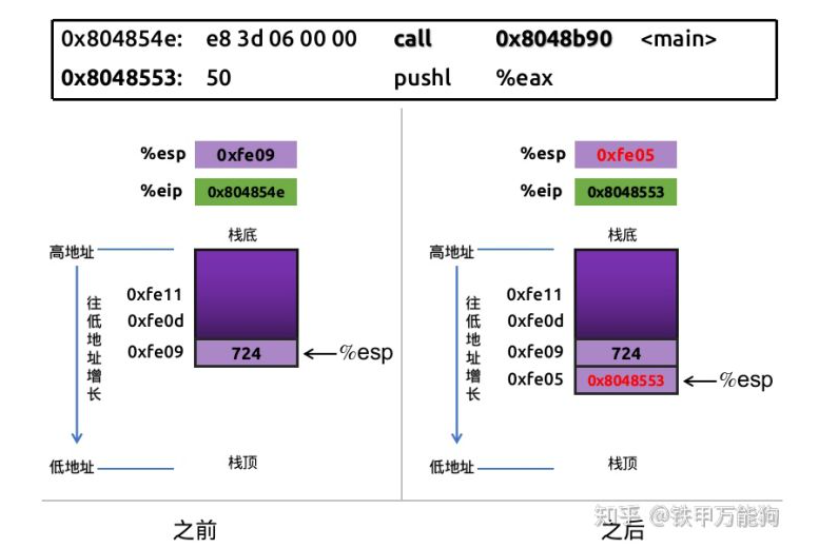
目前指令指针(又叫%eip指针)指示的地址是call指令所在行的地址是0x804854e,&esp指针指向栈顶位置保存着一个724的整数，可能是该过程的一个参数，该示例描述的是目前将要执行call指令(尚未执行)的程序状态

当执行call 0x8048b90这条语句,被调用者函数位于内存地址0x8048b90的位置，接下来会发生什么呢？  
由于此时已经读取了call 0x8048b90这条语句，但我们还没完成对call 0x8048b90的调用，此时%eip指针已经向前指向call指令语句的下一条指令，即会如图变化所示：**eip指针存储的内存地址更新为0x8048553**



接下来发生的事情，是将call 0x8048b90语句的下一条指令的地址，并将该地址值压入栈，此时栈的变化如下图所示：这里伴随着push指令的发生两个变化

* 将**返回地址**(即紧接着call指令的下一条指令所在行的地址)压入栈。
* 栈指针的向低地址递减4个字节，即此时%esp指针只想0xfe05.



接下来发生的事情，会跳转到8048b90这个地址即被调用者函数的所在行的起始地址。你必须要知道为什么是8048b90这个地址，而不是其他内存地址？生成这个地址的方式如下图,这种称为**相对寻址法**

https://pic3.zhimg.com/80/v2-d33878154447298c7595387aa503107a_720w.png

你要注意到指令中的常数是063d,当然示例用的小端机器的表示方法，所以在代码片段里显示为 3d06,但我们逻辑上的表示是063d,编译器会将该**内存地址常数**和eip指针的当前值作加法生成新的内存地址值，在这里即被调用者函数的内存地址。

**可能有人会问,为什么不让编译器自行决定任意一个可用的内存地址呢**？

对不起！我们不是编译器的设计者,作为C/C++程序员，只要理解到编译器使用的寻址原理，并在知道在生成汇编代码时，编译器已经在底层做了这些工作，我们没必要“打烂沙盘，问到底”。

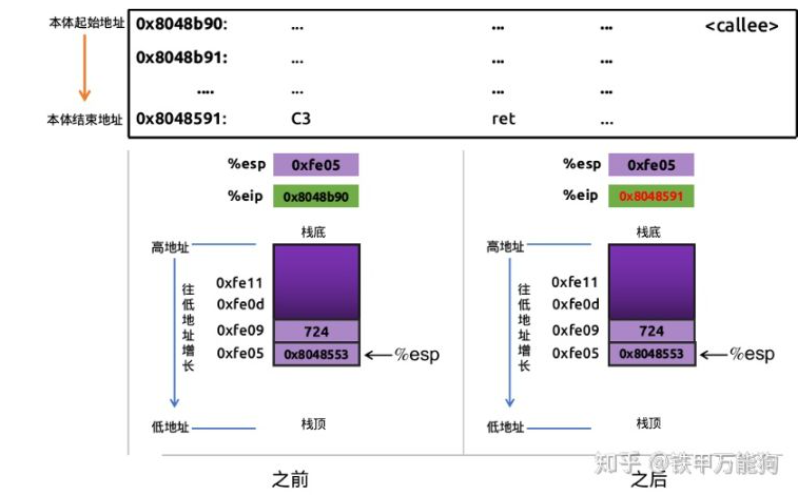
此时，我们真正在意的是,eip指针已经被替换为0x8048b90这个地址,换句话说,call指令告诉编译器可以执行jmp指令跳转到该地址即被调用者函数本体中的第一条指令，

如下图所示(左边的图例）:此时的程序状态包含了如下特征

* **对CPU/寄存器的控制权已经从调用者函数本体转移到被调用者函数的本体**。
* **当前eip指针指向的是被调用者函数的本体中的第一条指令的地址**，即0x8048b90。

下图被调用者函数的中间指令集不是本文讨论的内容，因此其中间的指令集我用“**...**”忽略了,当被调用函数**将**到达该本体的结尾之时，即eip指针指向ret指令所在行的地址0x8048591,如下图(右边的图例)所示,此时的程序状态是：

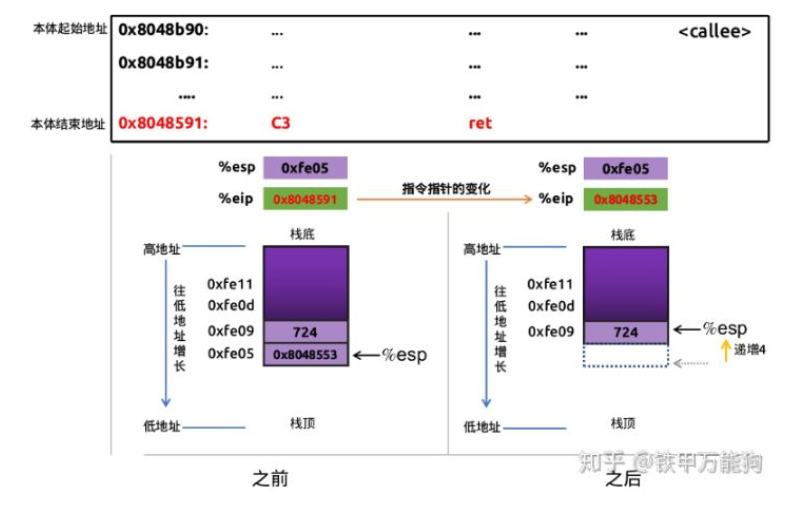
* %esp指针指向栈中的**返回地址**，但此时还没执行出栈操作。



跟接着,就弹出栈顶的**返回地址**(即pop操作),**返回地址**出栈是为了取得该地址,并跳转到该地址指向原来**调用者函数本体**中**紧接call 指令所在行**的**下一条指令**。此时程序的状态变化如下

* %esp指针会向高地址移动4个字节，即esp递增4,即指向0xfe09
* 被出栈的**返回地址**会被传入%eip指针,即0x8048553
* 控制权将从被调用者函数本体转移到调用者函数。

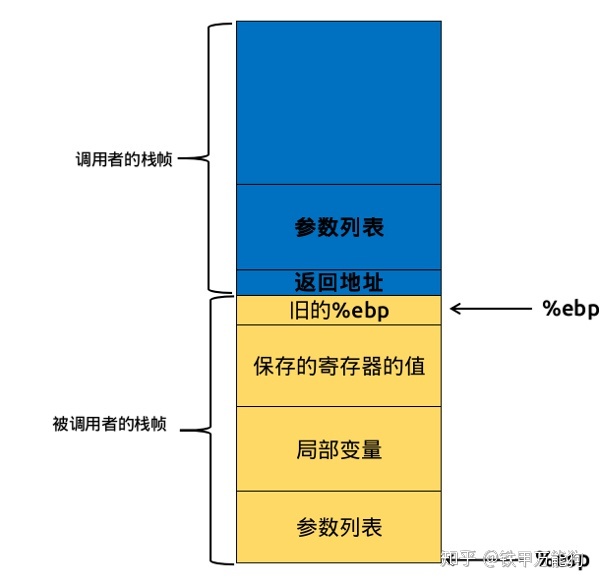
如下图变化所示



### ****返回值的处理****

从上面的示例中，我们都没有谈及到返回值是如何从被调用者传递给调用这函数的。是为了简化上面的示例分析，按照惯例，被调用者函数的返回值会放在eax寄存器中，eax的选择是相当随意的，可能是%ecx或%edx等，具体根据不同的C/C++编译器的实现而定。

* 调用者函数必须保证在调用可能返回一个值的被调用者函数之前,保存(eax)寄存器中的信息，因为被调用者函数和调用者函数共用同一个寄存器，被调用函数执行后会覆盖(eax)寄存器中的信息，这是寄存器保存操作中的约定。
* 被调用者函数在执行ret指令时,会将(计算过)的适合4个字节的任意类型的返回值保存到(通常是%eax)寄存器,也可能是其他寄存器, x86环境中的eax寄存器只有4个字节。
* 如果要返回大于4个字节的数据类型，最好的方式是返回一个自定义类型的对象的指针，而不是对象本身。
* 返回时，调用者函数在％eax寄存器(也可能是其他寄存器)中找到返回值。



**当前栈帧从**栈顶**到**栈底**如下构成**

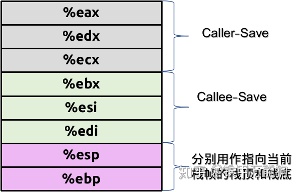
* **创建的参数表**为要调用的函数建立的参数
* **局部变量**:即在函数内部声明的变量(如果有)
* **保存的寄存器的上下文(如果有)**，当被调用函数返回之前，为调用者函数恢复原来的寄存器中的数据，如果当前架构有足够的空闲物理寄存器可供使用，一些寄存器信息可能不需要压入栈。
* 旧的帧指针，即前一栈帧的**ebp指针**\*，它指向前一帧的栈底。当被调用者函数结束时，寄存器ebp重新指向调用函数的栈底。

**调用者函数的栈帧**

* **参数列表**
* **本地变量**（如果调用者函数内部有声明局部变量）
* **返回地址**:我们知道调用者函数内部的执行到call指令时会将call指令所在行的下一条指令的内存地址压入栈,当**被调用者函数内部**执行到ret指令后,从栈中弹出**返回地址**,以便调用者函数回到该地址对应的指令继续执行余下的指令。

## **IA32平台的寄存器的使用约定**

C/C++编译器遵守如下约定的规则。



* 当该函数是处于**调用者**角色时，如果该函数执行过程中产生的临时数据会已存储在**%eax,%edx,%ecx**这些寄存器中，那么在其执行call指令之前会将这些寄存器的数据写入其栈帧内指定的内存区域,这个过程叫做**调用者保存约定(英文原名称:Caller Save)**。
* 当该函数是处于**被调用者**角色时，那么在其使用这些寄存器**%ebx,%esp,%edi**之前，那么该函数会保存这些寄存器中的信息到其栈帧指定的内存区域,这个过程叫**被调用者保存约定**。
* **%eax**总会被用作返回整数值。
* **%esp,%ebp**总被分别用着指向当前栈帧的**顶部**和**底部**,主要用于在当前函数推出时，将他们还原为原始值
* 对于全栈角度来说,%esp也叫做整个栈的**栈指针(Stack Pointer)**
* 对于某个函数的栈帧来说,%ebp也叫做该函数的**帧指针(Frame Pointer)**

## **C/C++ x86\_64的函数栈**

**x86 vs x86\_64 过程调用约定**

下表是对x86寄存器和x86\_64寄存器的一个**使用约定**的对比表，从该表可知，由于x86\_64中的物理寄存器数量比x86的多出了一倍，因此可以在很大程度上减轻了程序调用对栈的依赖，并且更好地利用寄存器交换临时数据,例如我们可以将参数直接缓存到寄存器中，并且也可以将局部变量直接缓存到寄存器，当然我们会耗尽所有可用的寄存器，然后又回到x86程序栈的约定的操作方式:参数先入栈，需要调用该参数时又加载到寄存器中。但是多数情况下，我们仍然尽最大努力尝试绕过函数栈直接利用寄存器,并避免过多地使用栈内存写入



回到我们上表，16个通用的寄存器,分别是8字节的位宽，我们知道

* 被调用者函数保存函数数据状态的寄存器,分别是RBX,RBP,R12,R13,  
  R14,R15.
* 用于**调用者函数传递参数**的寄存器是RCX,RDX,RSI,RDI,R8,R9,也就是说在理想情况下，我们编写的调用者函数的参数控制在6个参数以内的话，在x86\_64环境中,被调用者函数可以直接从这些寄存器中获取参数值，而且我们知道访问寄存器的速度远远快于从调用者栈桢中的参数域。但如果我们的被调用者函数的参数个数多于6个时，额外的参数不得不重新写入栈中，这又回到**x86架构的过程调用的约定**。
* 关于**返回值的处理**，在大多数情况下,x86\_64的过程调用仍然遵循x86的传统，仍然使用RAX寄存器作为返回值，只不过它是EAX的扩展版本而已。
* 关于**栈指针**的使用约定,x86\_64的过程调用仍然遵循x86的传统,使用RSP寄存器指向栈顶。
* 关于**局部变量**的使用约定,x86\_64的过程调用，若**存在闲置的寄存器**的话，局部变量可以直接缓存到寄存器中,若局部变量太多的情况下，再次回到x86过程调用的约定,将额外的局部变量写入栈中。
* callq指令在写入栈的返回地址是64位的尺寸,也就是说栈指针是隐含按照指令movq -8(%rsp),%rsp移动**栈指针**。
* 函数可以访问％rsp之后最多128个字节的内存：“红色区域”，意味着可以在通过%rsp的来在“红色区域”内存储一些临时数据。而不必使用使用多条指令。
* 关于**帧指针**的使用约定,如果深入研究过栈的话,可能大部分人会接受如下观点  
  x86\_64的过程调用取消了**帧指针**,所有对当前栈帧中的内存字段的访问引用,由%rsp进行相对寻址来实现,%rbp即被作为视为**通用寄存器**。

## **C++继承中虚表的内存布局**

### ****明确虚函数的目的****

* 我们要明确在类继承中使用虚函数的目的。在开发需求中,我们旨在让调用层代码保留相同的公共接口,因为调用层代码不需要关心被调用层的功能实现细节。那么虚函数就是\*\*让不同的派生类将继承自父类的同一个虚成员函数(接口)的根据派生类的功能需求进行不同行为的实现，以此达到不同的派生类提供调用层的决策代码同一个函数接口的不同实现版本，从而保持对调用层代码逻辑无需变动,而且隐藏了同一个函数接口的不同版本的实现细节。
* 示例导入

#include <iostream>

class Employee{

public:

bool iService=true;

virtual ~Employee(){};

virtual void add\_salary(){

std::cout<<"add\_salary method in Employee"<<std::endl;

}

};

class Teamer:public Employee{

public:

int idNo=1000;

virtual ~Teamer(){}

void add\_salary(){

std::cout<<"add\_salary method in Teamer"<<std::endl;

}

virtual void info(){

std::cout<<"Teamer info for Teamer"<<std::endl;

}

void show(){

std::cout<<"show method in Teamer"<<std::endl;

}

};

int main(void){

Employee \*tm1=new Teamer();

Employee \*tm2=new Teamer();

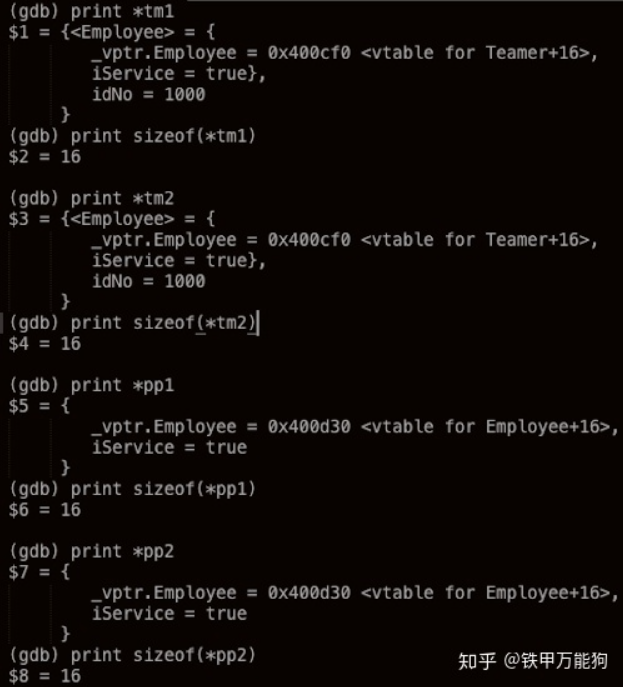
Employee \*pp1=new Employee();

Employee \*pp2=new Employee();

delete tm1,tm2,pp1,pp2;

}

* 在这里我们可以尝试打印**\*tm1,\*tm2,\*pp1和 \*pp2**，如下图所示



从上图的输出中,我们要引入一个**虚指针(\_vptr)**的概念

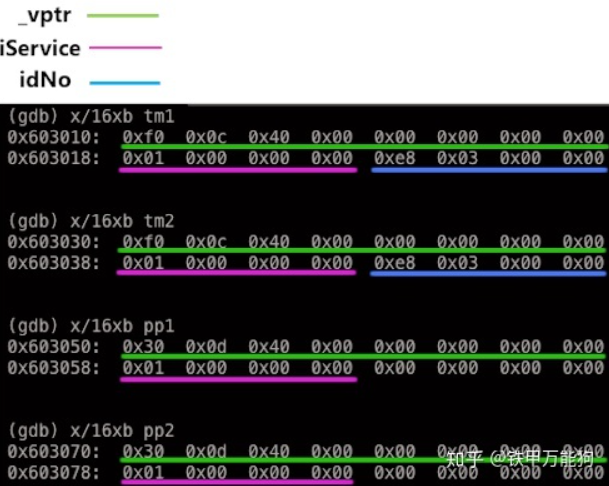
* **虚类的对象初始化时会自动创建一个隐藏的数据成员\_vptr指针指向虚表,此前声明该虚类的对象编译器也创建了该虚类的虚表**。
* **后续同一个虚类所有对象实例共享同一个虚表**,截图中的tm1和tm2的隐藏指针指向同一个地址0x400cf0,pp1和pp2的虚表是同理如是.
* **虚表表当前的地址是一个已经+16字节偏移后的内存地址**

另外我们还打印出所有Teamer对象和Employee对象,他们获得内存分配都为16个字节。因此我们不妨在查看我们刚才实例化的所有对象。

### ****查看对象的内存数据****

现在我们不妨看看刚才实例化的各个对象的内存布局，使用x命令,因为每个对象的堆内存块尺寸都为16个字节,因此我们使用x/16xb将他们的内存数据转存到屏幕中,如下图所示。

* \_vptr在虚类的对象中就占用8个字节,该\_vptr存储了指向该虚类的虚表的内存地址值。
* iService是一个bool类型仅占用1个字节，另外高位的3个字节空间由于内存对齐的原因都以0填充。
* idNo是一个4字节的int类型,对于Teamer的对象0x03e8的值就是十进制的1000，对于Employee的对象这里的4个字节由于按8字节内存对齐,仅作为填充位之用。



### ****探究虚表的内存布局****

我们从前文打印的第一个Teamer对象 tm1的信息中，可以知道其\_vptr指针指向0x400cf0,你是否发现“**<虚表 for Teamer+16>**”的字样。这个其实表明0x400cf0是已经+16字节偏移后的地址值

我们已经在前文提到在首个的新的虚类对象且初始化时,编译器会该类动态创建一个虚表,但为什么每个不同虚类的虚表都要额外偏移16个字节呢？ 在本示例中,我们不妨减去这个偏移量,也即得到0x400ce0这个地址，然后使用x命令，该命令将300字节的内存数据转储到屏幕。

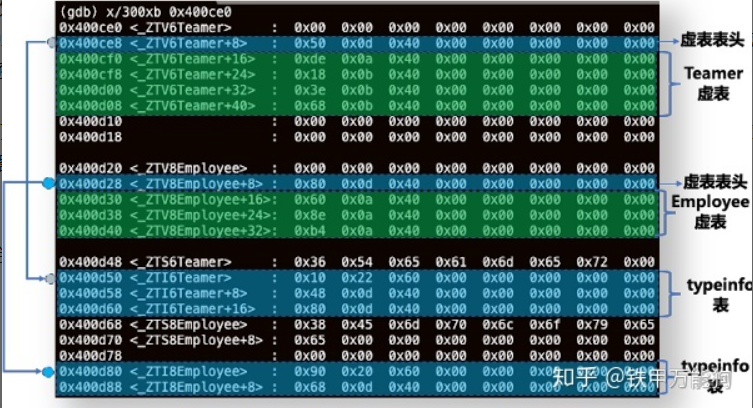
(gdb) x/300xb 0x400ce0

上面的命令以十六进制格式打印300字节,从0x400d00开始。 为什么要这个地址？ 因为在上面我们看到类Teamer的虚表指针指向0x400d10，该地址已经偏移0x10个字节,即减去0x10就能得到原本虚表的地址。

下图中**\_ZTV**是虚表的前缀，**\_ZTS**是type-string（名称）的前缀，**\_ZTI**是type-info的前缀。

我们从下图可以得到很多虚表的内存细节。

* 每个Teamer虚表存在一个虚表表头占用16个字节,前8个字节0填充,后8个字节包含一个指向与该类对应的typeinfo表的地址（没必要理会，只需知道他们占用16个字节即可）。
* 每个typeinfo表的前面也包含一个typeinfo name的信息(没必要理会,l罗列出来只是让你知道有这么一个描述字段)
* 绿色的部分就是不同虚类的虚表,虚表就是包含了该类定义的所有virtual成员函数的函数地址。

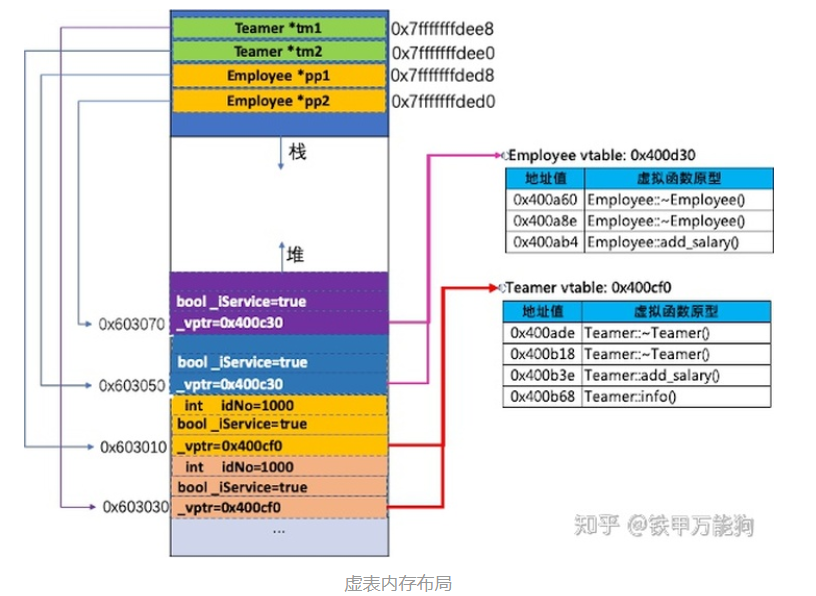


我们可以从上图中绿色部分的内存数据中即每行冒号之后的8字节空间提取有用的数据，例如

* 0x400cf0到0x400d08的内存区域中的内存数据,对应的是Teamer类类虚表中virtual成员函数地址的条目。
* 0x400d30到0x400d40的内存区域中的内存数据,对应的是Employee类虚表中virtual成员函数地址的条目

我们这两个内存区域的数据分别整理成如下表，注意写本文时使用的是CentOS 7的x64小端机器，因此读取图中的内存数据时，是从右向左读取，因此整理下表每个内存位置对应的值，并且分别是有**info symbol**命令 再次查看每个内存位置的值对应的具体含义。

结合整理如下表可知：**虚表中的地址值分别代表虚拟类中对应虚函数的地址**。



更简单获取虚类的虚表条目的另外一条命令就是**info vtbl**，这里就不展示了,我们看到上图的虚表中的虚解构函数都成对地出现,我们先暂不讨论为什么会这样，因为我日后会令起一文再阐述该问题。

* 第一个解构函数，称为**完整对象解构函数(complete object destructor)**，执行销毁操作时无需在对象上调用delete()。
* 第二个解构函数称**为删除析构函数**( deleting destructor),在销毁对象后调用delete()。
* 两者都摧毁了任何虚拟基类.一个独立的非虚函数称为**基类对象解构函数**(base object destructor)执行对象的销毁操作，但不执行其**虚拟基类子对象**的销毁操作，并且不调用delete()。
* 非虚函数是静态绑定的(**编译时绑定**)，因此在虚表中不存在任何非虚函数。

### ****虚表构建细节****

我们仍然使用上文的调用示例代码

int main(void){

//

Employee \*tm1=new Teamer();

Employee \*tm2=new Teamer();

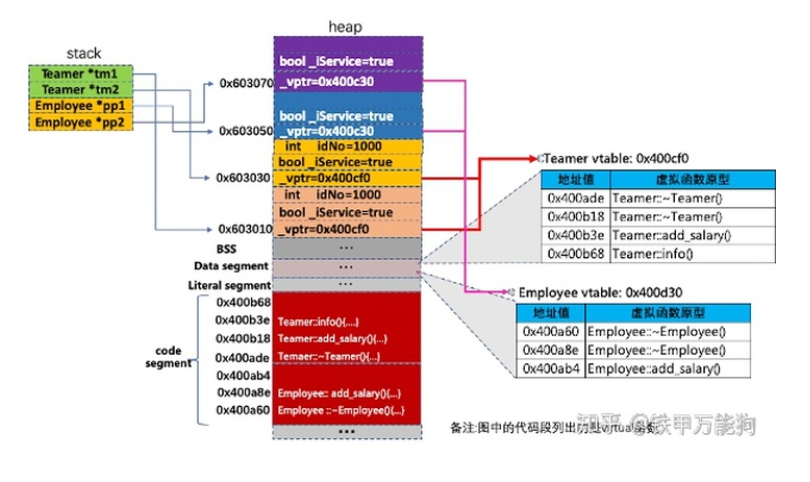
Employee \*pp1=new Employee();

Employee \*pp2=new Employee();

delete tm1,tm2,pp1,pp2;

}

从上面的示例代码中我们已经知道



* 首先，每个使用虚函数的类或从基类派生的虚函数的类都被赋予自己的虚表。该表只是C++编译器在“**编译时**”设置的**静态数组**。虚表包含当前类中所有虚成员函数的函数指针的相关条目，那么填入虚表的虚成员函数指针有四种来源。

1. **派生类本身原创定义的虚函数**,例如上图的Teamer::info()函数。
2. **从父类继承的虚成员函数,且该函数未被派生类重写**。
3. **从父类继承的虚成员函数,但该函数已被派生类重写**。值的注意的是,**虚表的虚成员函数指针始终指向该类中的最新的派生版本的虚成员函数**。理解这句话非常重要!举个例子Teamer类从Employee类继承了add\_salary()函数,但Teamer类重写(注意:不是重载)了该add\_salary()函数,对于Teamer虚表来说,填入表中的add\_salary()函数的地址是0x400b3e,而不是父类的add\_salary()的地址0x400ab4。
4. **若当前类定义了虚解构函数,那么该类的虚解构函数的解构函数的地址会“成双成对”地填入虚表中**。按照惯例,由于定义类时优先定义解构函数,再实现其他成员函数，因此该虚解构函数对的地址通常会出现在表中头两行,上图是很好的例证。

 然后，当类对象实例化时会将\*\_vptr设置为指向该类的虚表。例如，当创建类型为Teamer的对象时\*\_vptr设置为指向Teamer的虚表。构造类型为Employee对象时，\*\_vptr设置为指向的Employee的虚表。我们这里先不讨论virtual解构函数,目前只针对其他虚函数进行讨论。

* 对于基类Employee类型的对象,它只能访问Employee的成员,Employee类型的对象无法访问Teamer类的的成员函数,因为地址为0x400ab4的地址仅指向Employee::salary()
* 同理,Teamer类型的对象也只能访问Teamer::add\_salary()和Teamer::info()。