PvZ Mod Development in a Nutshell

Plants vs. Zombies

2024/11/1

00

绪论

数据在内存之中的表达

内存和地址

- 内存是计算机中存储数据的地方，所有程序在运行时需要被使用和运算的数据都需要在内存中
- 地址则指示了数据在内存中的位置，通过地址，计算机得以在内存中寻找到这个数据

简单数据类型的内存表达

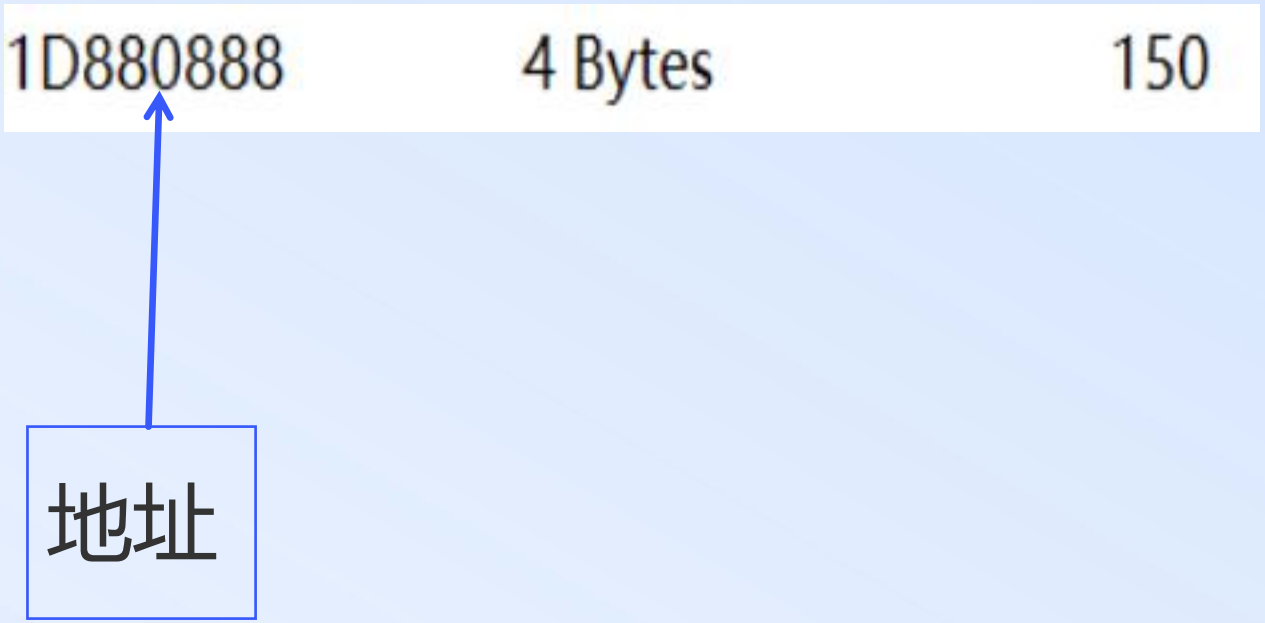
- int
- float

四字节

- double → 八字节
- short → 双字节

- bool
- char

单字节



复杂数据类型的内存表达 - 结构体和类

- 结构体与类

```
struct test{
    int a;
    int b;
};
```

成员顺序是固定的

如上图，a在内存中的位置一定在b之前

复杂数据类型的内存表达 - 结构体和类

```
struct test{
    int a;
    int b;
};
```

当该结构体被实例化时，将有一个初始内存地址，从这个内存地址开始的一些部分将存储这个结构体的成员

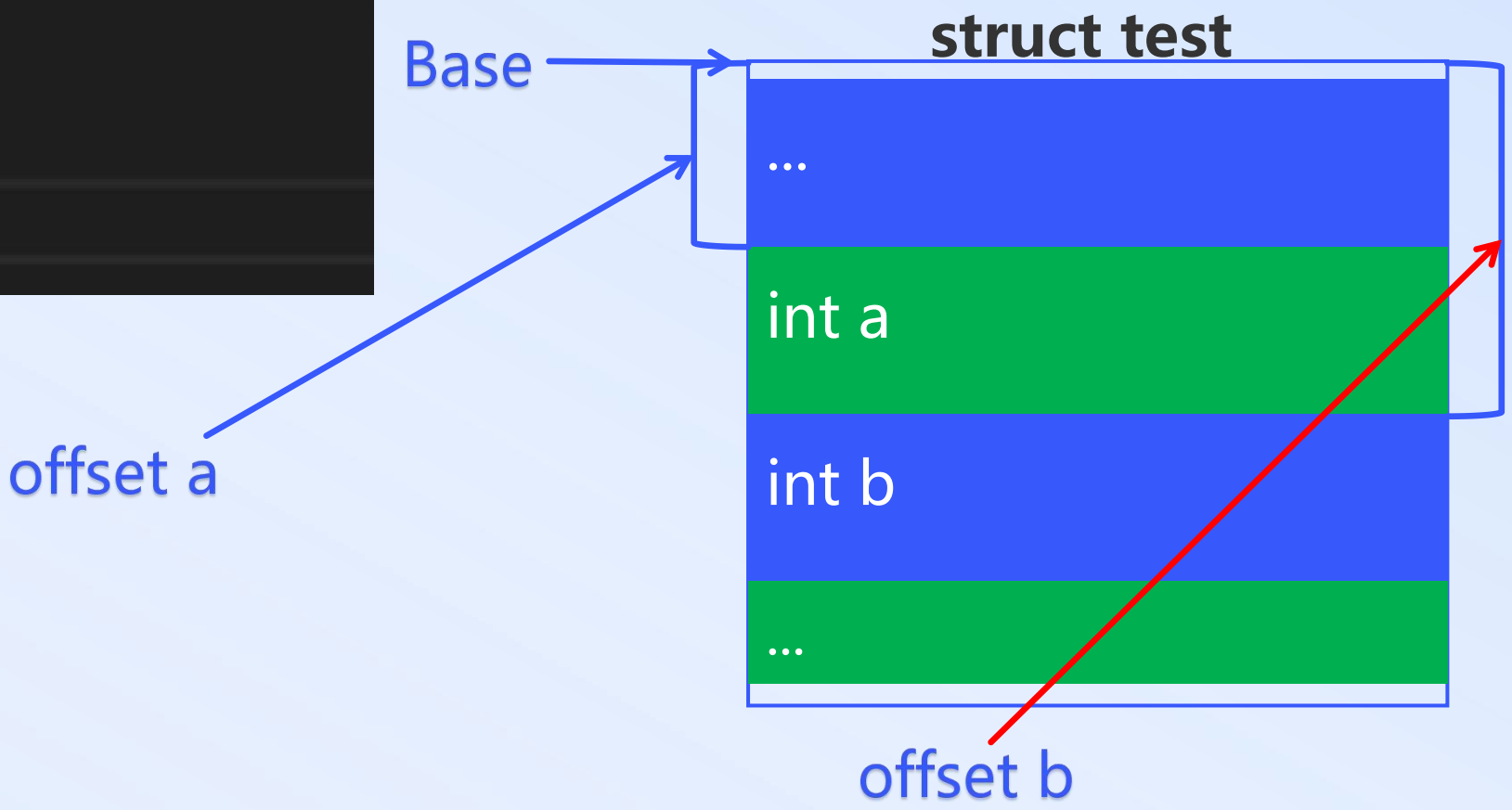
这个初始内存地址就是基址(base)

由于结构体的各类成员在内存中的顺序是完全固定的，所以，根据该结构体的基址，将这个基址加上一个固定的数值，就可以得到各个成员的起始内存地址

这个固定的数值就是该成员相对基址的偏移(offset)

复杂数据类型的内存表达 - 结构体和类

```
struct test{
    int a;
    int b;
};
```



复杂数据类型的内存表达 - 结构体和类

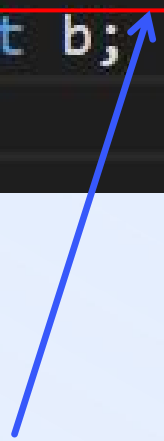
```
struct test{
    int a;
    int T(){
        return 0;
    }
    int b;
};
```

请注意，成员函数不参与偏移量的计算！

复杂数据类型的内存表达 - 结构体和类

```
struct test{
    int a;
    struct T {

    };
    int b;
};
```



没有定义T的实例（仅仅是声明了T），不参与偏移量计算

复杂数据类型的内存表达 - Union

- Union(联合体)

```
union test{
    bool a;
    int b;
};
```

Union在内存中的大小为Union中的成员最大内存长度(即所有成员公用一段内存)

如上图中的Union内存长度为4(即Int类型的长度)，而非4+1=5

01

PvZ中的指针

指针的内存表达

- 每一个数据在内存中体现为它的地址和它的值

指针的内存表达

- 一个变量拥有两层含义 —— 它的地址和它的值（只是使用变量名方便人来访问）
- 如：
- int a = 1919810; // address: 114514
- 则a这个变量拥有两层含义，其一为a的值为191810，其二为a这个变量存储的地址是114514
- **为了方便表达，我们称变量是一个av对，即<address, value>**

指针的内存表达

- 再次说明：

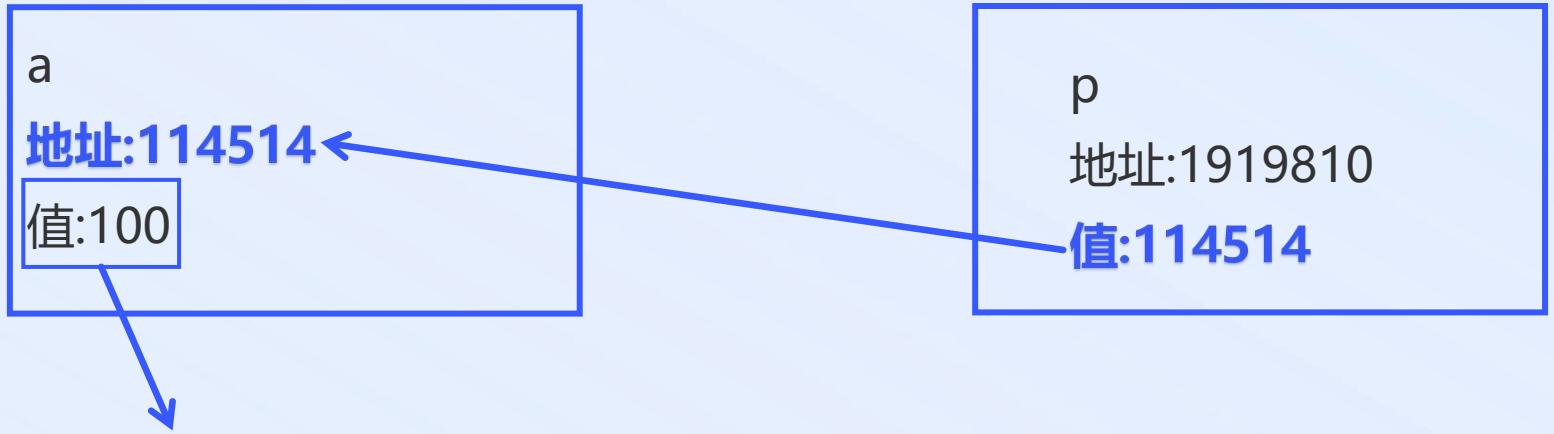
- **变量是一个av对，即<address, value>**

指针的内存表达

```
int a = 100;
int* p = &a;
//*p = ?
```

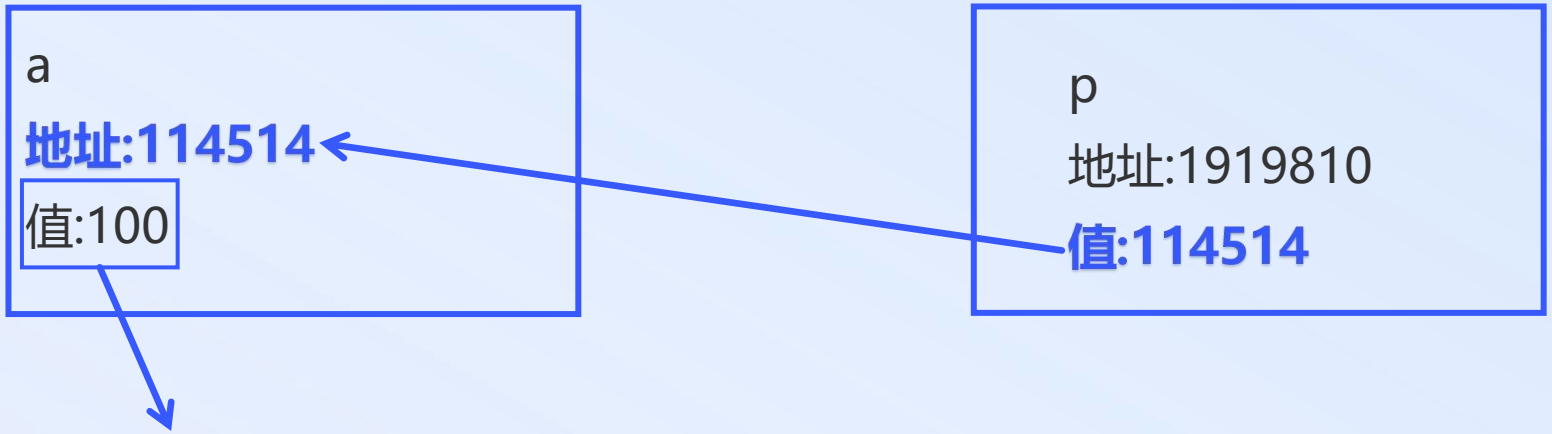
&a的作用是取出a的**地址**，即114514，并将其作为p的**值**

*p则是将**p的值**作为**地址**，获取该地址**存储的值**



***p = 100，p=114514**

指针的内存表达



***p = 100，p=114514**

事实上，*p在汇编语言中，相当于**[p]**

即[114514]=100

它的作用是根据p的av对中的**value**，将该**value**作为**地址**，

访问内存得到一个**新的av对(变量)**，并取出**这个av对的value**

类似的，&a的作用则是取出a这个av对的**address**值

阳光指针的理解 - 基址和偏移

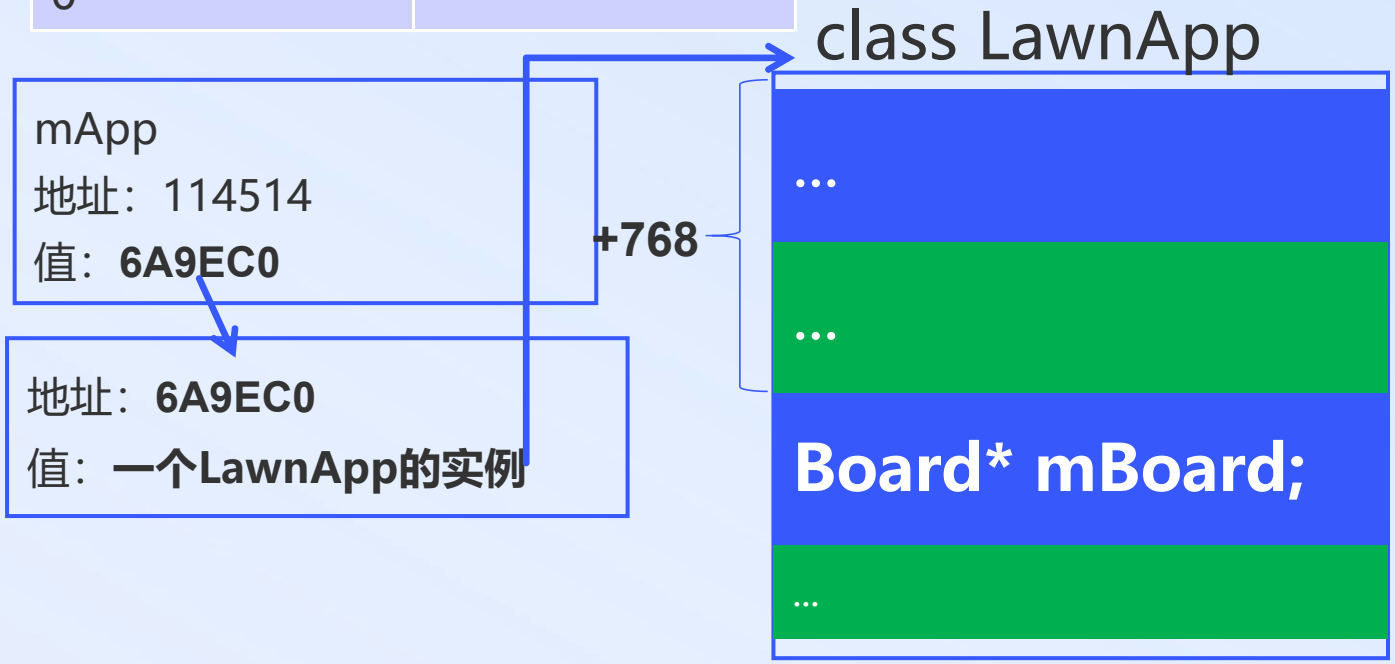
我们之前以结构体为例初步了解了基址和偏移的概念

而这是一个指针，它的基址和偏移各自意味着什么呢？

阳光指针的理解 - 基址和偏移

```
1  class Board {
2      int mSun;//+5560
3  };
4
5  class LawnApp {
6      Board* mBoard;//+768
7  };
8
9  LawnApp* mApp;//6A9EC0
10
```

地址	值
6A9EC0	314A290
314A290+768	28390010
28390010+5560	150



阳光指针的理解 - 基址和偏移

地址	值
6A9EC0	314A290
314A290+768	28390010
28390010+5560	150

```
1  class Board {
2      int mSun;//+5560
3  };
4
5  class LawnApp {
6      Board* mBoard;//+768
7  };
8
9  LawnApp* mApp;//6A9EC0
10
```

[[[6A9EC0]+768]+5560]

第一步：通过6A9EC0这个地址，找到mApp这个av对(adress = 6A9EC0, value = 314A290)

第二步：取出mApp的value，通过value访问内存得到一个新的av对(adress = 314A290)，这个av对是一个LawnApp类的实例

第三步：由于这个新的av对是一个实例，而类在内存中是连续存储的，所以我们只需要对它adress加上一个固定的偏移（768），就可以在内存中找到一个新的av对mBoard

第四步：根据mBoard这个av对的Address，在内存中找到了一个新的av对，这个av对是Board类的实例

第五步：同样，将这个新的av对的Address加上一个固定的偏移(5560)为地址，在内存中找到一个新的av对，而取出这个av对的value，就是阳光数

Practice Time 1

汇编语言初步

- 汇编语言是低级语言，是现阶段**大多数**程序员可接触到的相对而言最最最低级的语言
- 汇编语言的指令和数据类型一样，**将存储在内存中以供调用**
- 我们可以同数据类型一样理解汇编语言，其的每一段指令（或者说代码），都代表了**这段指令的地址**和**这段指令本身**

汇编语言初步

- 汇编语言的常用指令:
- mov a,b: a = b
- jmp address: 跳转至address, 从address地址继续执行汇编指令
- call address: 调用位于address地址的函数
- ret [optional](4的倍数): 返回
- add a,b: a += b
- sub a,b: a -= b

汇编语言初步

- push a：将a入**栈**
- 这里的栈是程序临时存储数据的地方（内存），将数据入栈将方便程序后续取用这个数据
- 当我们后续需要取用数据时，可以通过下列指令取用**栈顶**的数据（栈遵循先入后出的原则）
- **pop eax ：将栈顶的数据取出给eax 或者说：**

eax = stack[stack_size - 1] #stack_size指这个栈现有的数据个数

Practice Time 2

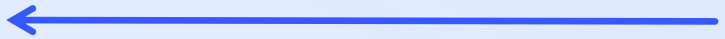
02

函数调用

形参入栈

- 对于一个函数调用，我们首先要将各个函数调用的参数压入栈中，以便后续函数体内取用它们
- 在汇编语言中，将参数压入栈的指令是push，调用函数的指令为call（前文已述）
- 请留意，函数实际上是很多汇编指令的集合体，**所以函数本身也有一个地址**，这个地址实际上是函数汇编指令在内存中的起始地址，通过这个地址计算机才能调用函数本身

形参入栈



```
int test_function(int a, int b, bool c){;;}
```

对于上述函数，假设test_function的**所在地址**为114514

当我们将实参入栈时，遵循**从右向左**的规则，即

```
push c

push b

push a

call 114514
```

附注：实际上，对于上述情况，形参入栈也有可能通过寄存器（见下页），即先将形参存储到寄存器，再将寄存器入栈，这与编译器有关，此处不予讨论。

取指针和取引用的入栈方式

```
int test_function(int a, int& b, int* c){;;}
```

在处理上述函数test_function入栈时，我们需要引入**寄存器**的概念

寄存器同内存一样，都是存储数据的场所，**但是寄存器是CPU的一部分**

常用的寄存器有

eax、ebx、ecx、edx 通用寄存器，用于临时存储数据

esp、ebp 维护栈指针

eip 指令寄存器

指针(*)和引用(&)的入栈方式

```
int test_function(int a, int& b, int* c){;;} //address:114514
```

对于int& b和int* c两个参数，我们需要先将它们给到**寄存器**，再将寄存器入栈

则函数调用操作为

```
mov eax,c
push eax
mov ebx,&b
push ebx
push a
call 114514
```

函数调用结果的存储

函数调用的结果一定存储在**eax**寄存器中

如 `int add(int a, int b){//返回a+b} //address:114514`

则:

```
push 2

push 1

call 114514
```

在调用函数完成后，`eax=3`

类成员函数的调用

对于类成员函数，我们都知道只有**类的实例**才能调用类的成员函数

如：

```
class TestClass{
   int add(int a, int b){//这是一个类成员函数}
}
```

则对于上述函数，在入栈时，我们需要额外入栈一个类的实例，这个类的实例将被存储在ecx这个寄存器中

类成员函数的调用

```
class TestClass{
    int add(int a, int b){//这是一个类成员函数}
}
```

而对于下列代码：

```
TestClass c;
c.add(1,2);
```

则ecx应为该实例c的地址

而对于

```
TestClass* c = new TestClass();
c->add(1,2);
```

则ecx应为c这个指针指向的实例的地址

类成员函数的调用

```
class TestClass{
    int add(int a, int b){//这是一个类成员函数}
}
```

然后，先将其他参数入栈，再将ecx入栈

即：

```
push b
push a
push ecx
call ...
```

call操作

- 所以，当我们call操作执行时，计算机又做了哪些操作呢?
- 则当call执行时，计算机会执行下面两步操作
- 第一步：将call下面一行指令的地址入栈（即push c这条指令的地址，**也就是将返回地址入栈**）
- 第二步：跳转到114514这个地址，执行函数内的指令

```
push a
push b
call 114514
push c
```

ret操作

- 在所有函数末尾，都将存在一个ret语句，该语句的作用是将上面call操作时压入的地址出栈，并跳转到这个地址（也就是返回地址）
- 另外，一般的编译器中，若函数的参数中**不存在变参**（也就是数量**不可变**的参数），**ret将跟有一个为4的倍数的参数**，如ret C(在十进制中是12)，12是所有**push到栈的参数**（不含返回地址）**的长度总和**（如push了3个int，就是ret C），**它的作用相当于将这12个字节的参数出栈，无需我们再手动管理栈**
- 而对于有**变参**的函数（如format函数），在ret时将**没有**这个额外的参数，而仅仅是一个ret指令，当函数返回后，若push进栈12字节的参数，将需要执行一段add esp,C，来手动管理栈

Practice Time 3

Materials of Pratice 3

```
    Board::AddCoin(CoinMotion
40C theCoinMotion, CoinType
B10 theCoinType, int theY, int theX,
    ecx = Board* this)
```

```
5AF
400 Sexy::Rand(eax = int range)
```

```
eax = Coin*
aCoin; ecx,
edx
```

```
eax = int
aRandVal; ecx,
edx
```

在 (theX, theY) 处创建并初始化以 theCoinMotion 方式运动的 theCoinType 物品。
一周目的 1-1 关卡中，额外创建点击拾取阳光的提示字幕。

取得一个 [0, range) 区间的随机整数返回给 eax。

self-study: esp & ebp

- 在函数调用中，一般在函数的开始会存在下面两个操作：
- `push ebp`
- `mov ebp, esp`
- 在函数末尾（ret前）也会相应做这个操作
- `mov esp, ebp`
- `pop ebp`
- 这是创建和销毁**栈帧**的操作，其作用是将**栈的基址指针在函数调用后修改到栈顶**，以方便取用函数调用的各个参数，感兴趣的可以自己去学习，这里不再赘述

特别鸣谢

- ueu4573(4573去) 舟批 mixed版作者
- Ghastasaucey Amadeus Vermeil(恶魂) GHTR系列(Ghtr、GHtr、GhTr作者)

广告

PvZ(1代) 16周年庆典

PVZ同人创作论坛
Plants vs. Zombies

03

互动环节

奖品

- Top 1: 豌豆玩偶
- Top 2: 坚果水杯
- Top 3: 向日葵卡扣
- 屏幕上的最后一名: 神秘奖品（x