



## 第七章 运行时环境—图解

冯 洋



# 一个简单的例子



```
int fun(int a, int b);  
int m=10;  
int main()  
{  
    int i = 4;  
    int j = 5;  
    m = fun(i,j);  
    return 0;  
}
```

```
int fun(int a, int b)  
{  
    int c = 0;  
    c = a + b;  
    return c;  
}
```

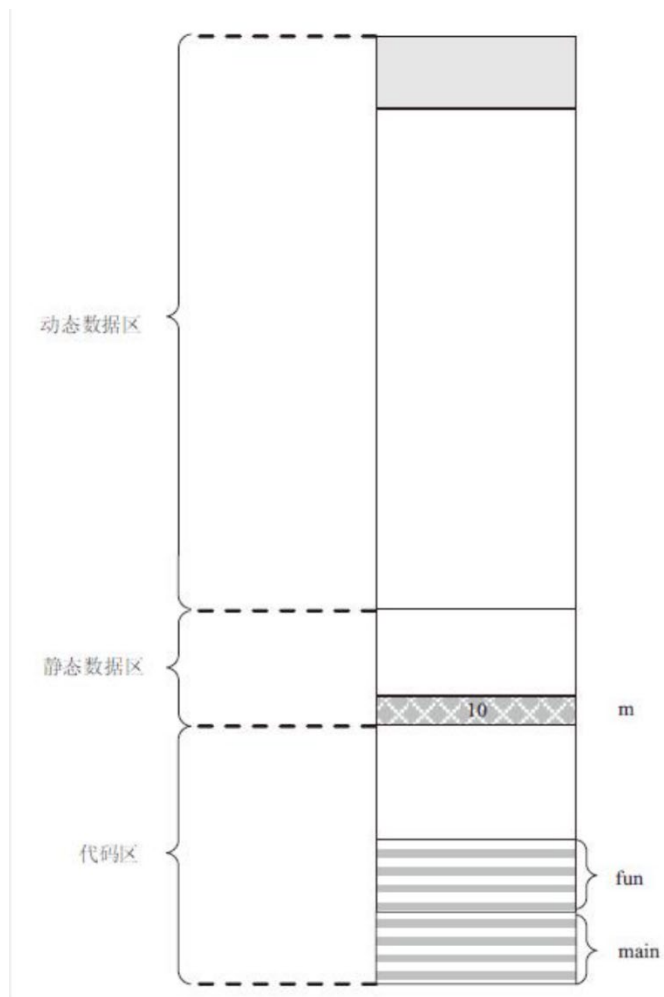
两个函数;  
一个全局变量m;  
若干局部变量i, j, c, a...  
一次函数调用;



# 一个简单的例子



整个程序放到了内存中的样子如右图。共有三个区域：代码区，静态数据区，动态数据区。



```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}

int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```



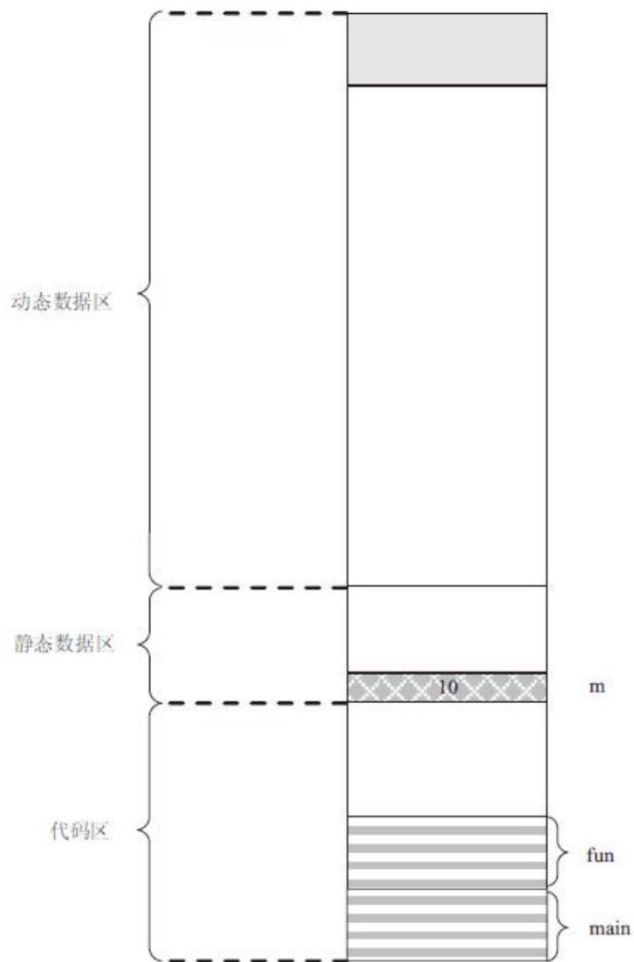
# 一个简单的例子



静态数据区：可以在编译时刻知道大小的数据对象，包括全局常量和编译器产生的数据（比如支持垃圾回收的信息等）

动态数据区：无法在编译时刻知道大小的数据对象

代码区：因为目标代码在编译时刻已经知道具体的大小，代码区也是一种静态区。

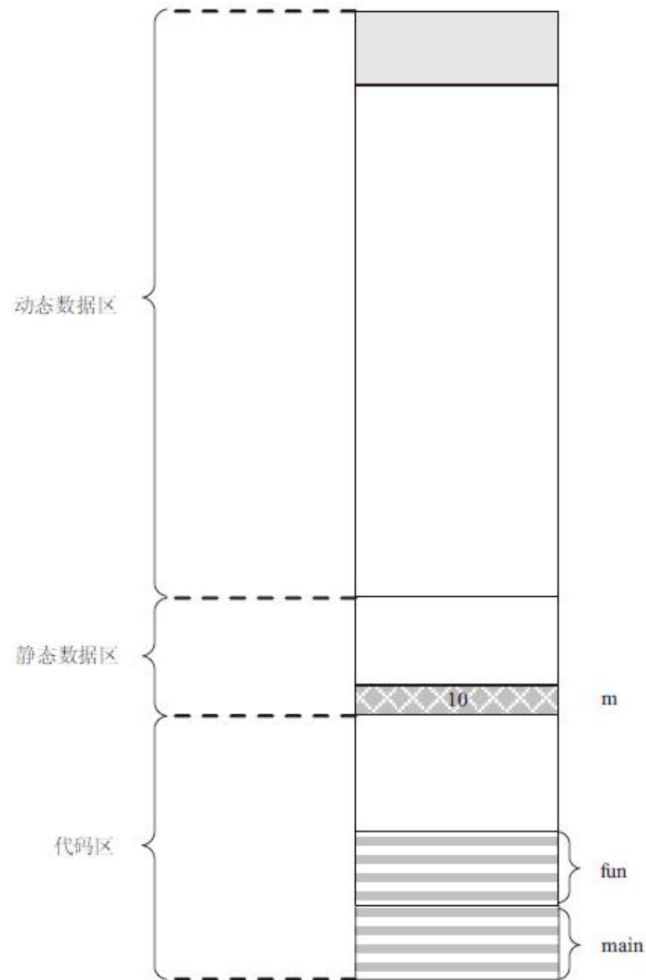


```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}

int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```



# 一个简单的例子

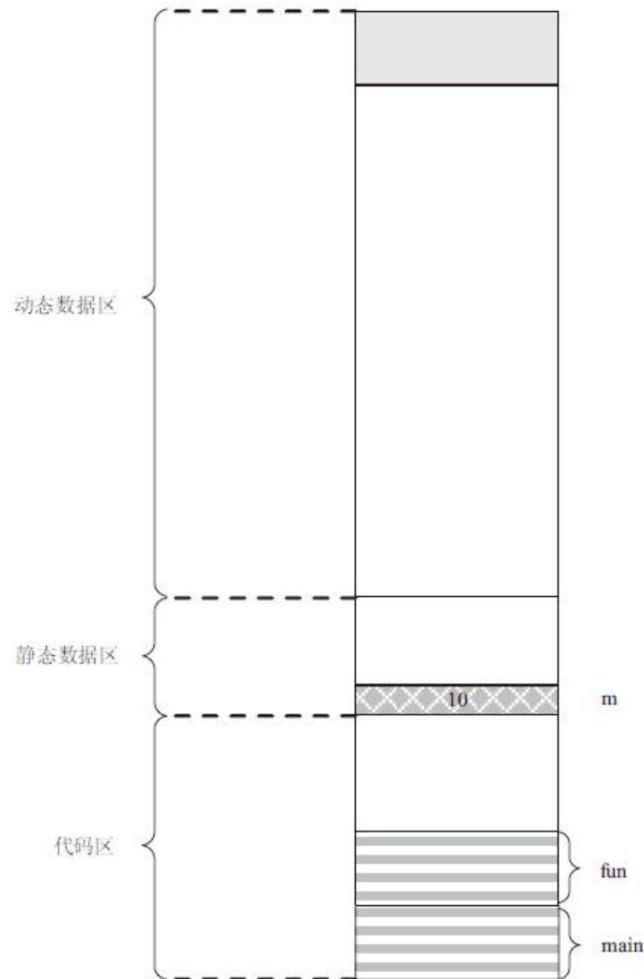


全局变量`m`的数值装载在静态数据区中；  
函数`fun`和`main`的代码装载在代码区中；  
程序开始执行前，动态数据区中没有数据；

```
int fun(int a,int b);  
int m=10;  
int main()  
{  
    int i=4;  
    int j=5;  
    m = fun(i,j);  
    return 0;  
}  
  
int fun(int a,int b)  
{  
    int c=0;  
    c=a+b;  
    return c;  
}
```



# 一个简单的例子

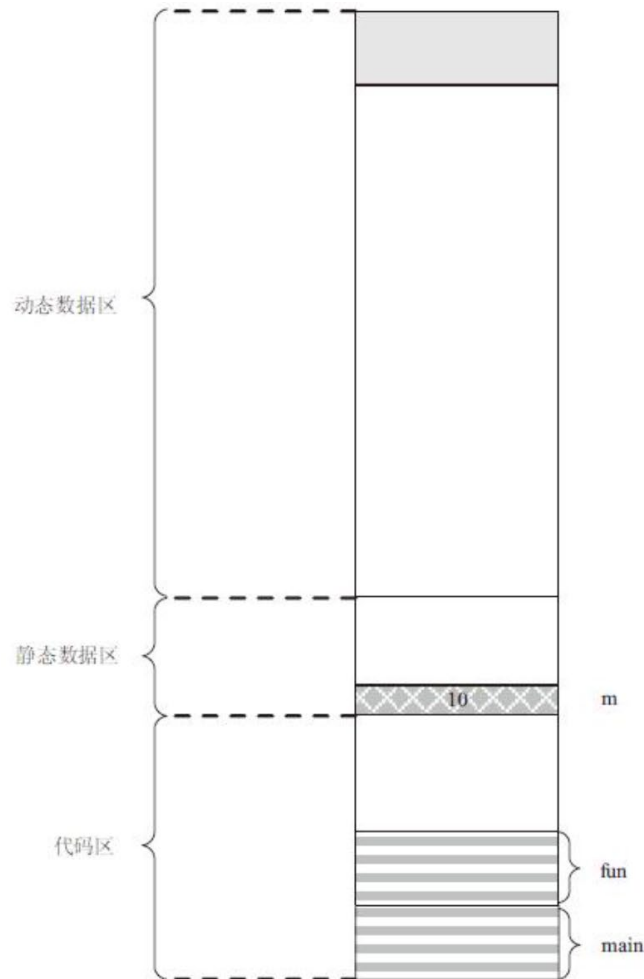


全局变量`m`的数值装载在静态数据区中。  
函数`fun`和`main`的代码装载在代码区中。  
程序开始执行前，动态数据区中没有数据；只有在程序开始执行后，在指令的驱动下，这一区域才会产生数据，压栈和清栈的工作就是在这一区域完成的。

```
int fun(int a,int b);  
int m=10;  
int main()  
{  
    int i=4;  
    int j=5;  
    m = fun(i,j);  
    return 0;  
}  
  
int fun(int a,int b)  
{  
    int c=0;  
    c=a+b;  
    return c;  
}
```



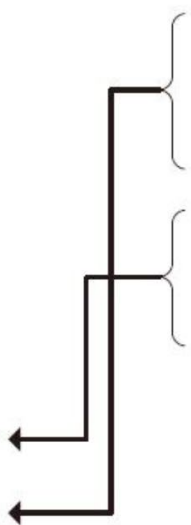
# 一个简单的例子



程序执行的本质，就是代码区的指令不断执行，驱使动态数据区和静态数据区产生数据变化。

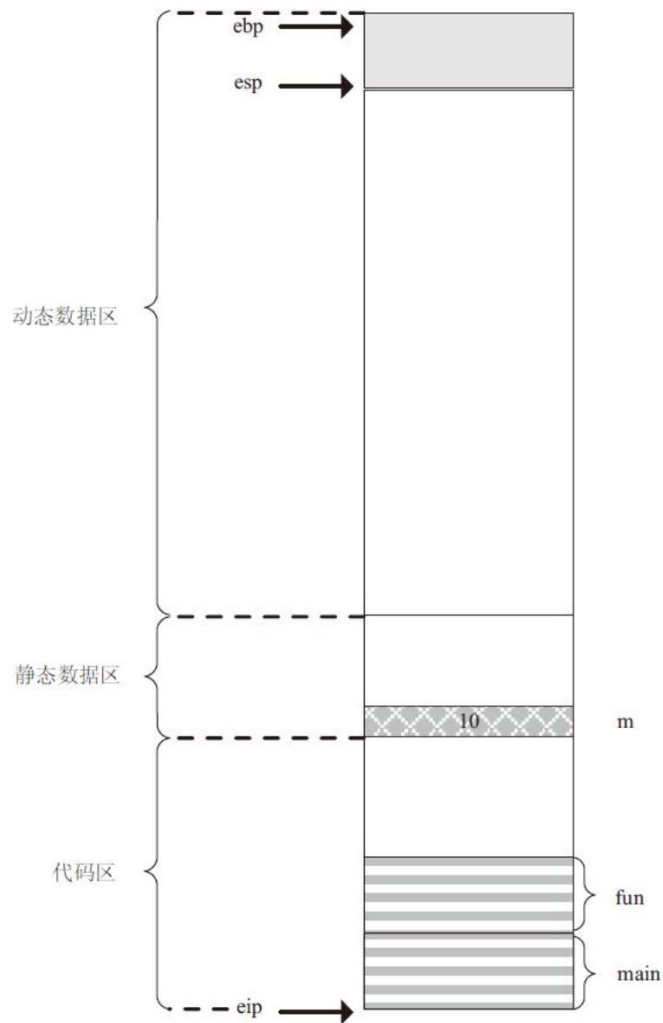
```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}

int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```





# 一个简单的例子



我们假定有三个CPU 中有三个寄存器，分别是 **eip**、**ebp**、**esp**。其中**eip**永远指向代码区要执行的下一条指令；**ebp**和**esp**用来管理栈空间，**ebp**指向栈底，**esp**指向栈顶，代码区中的程序执行，函数调用、返回，均存在不断的压栈与清栈。

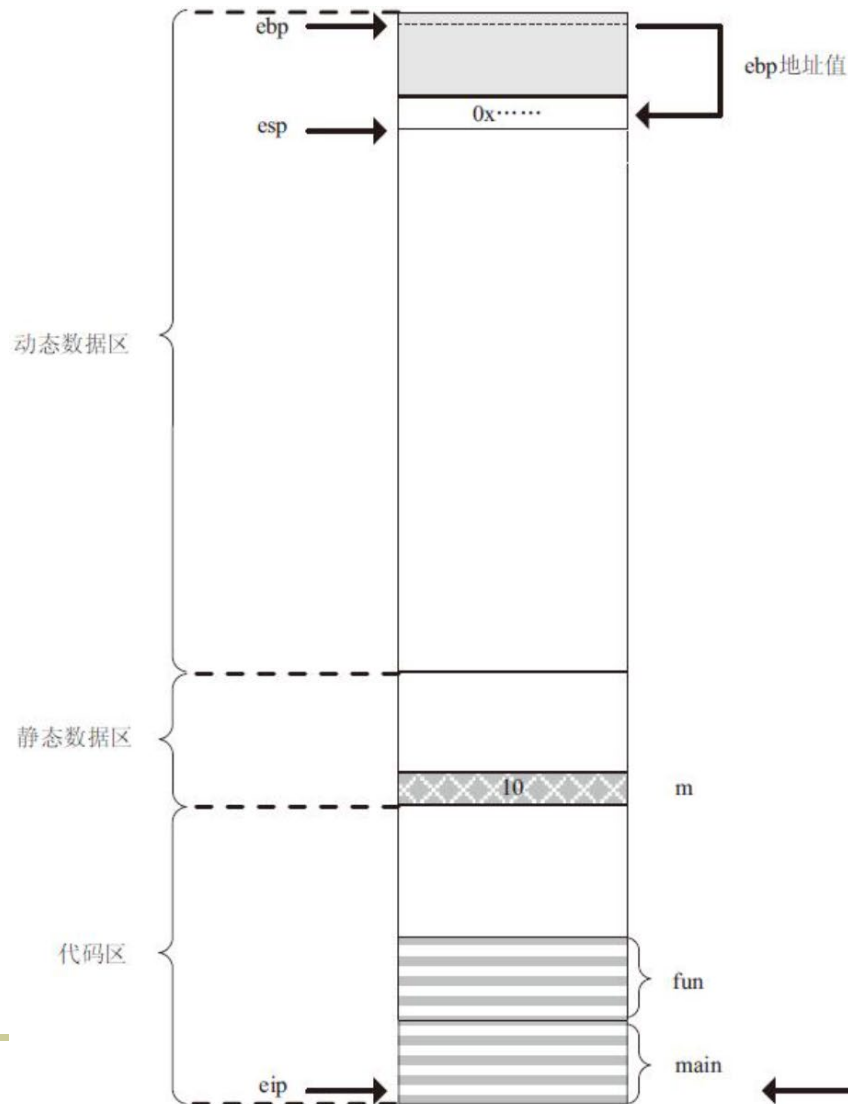
```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}

int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```





# 一个简单的例子



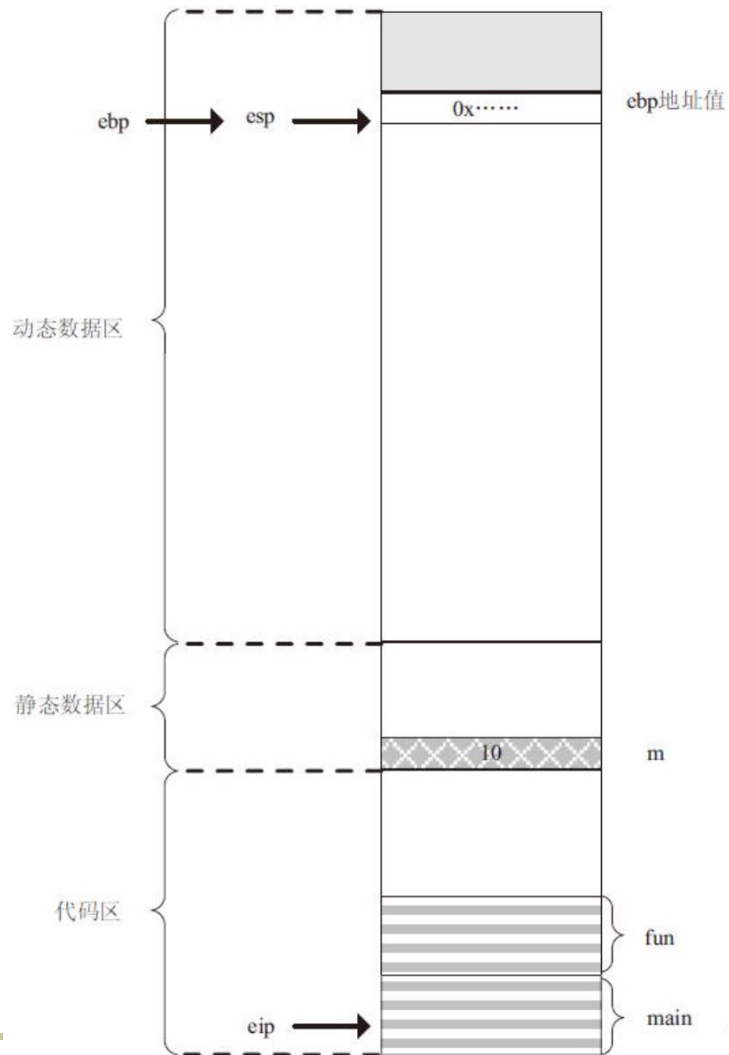
程序开始执行main函数第一条指令，**eip**自动指向下一条指令。第一条指令的执行，则使得**ebp**的地址值被保存到栈里面。保存的目的是本程序执行完毕后，**ebp**还能返回到现在的位置，复原现在的栈。**ebp**保存之后，**esp**则自动向栈顶移动，它永远指向栈顶。

```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}
```

```
int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```



# 一个简单的例子



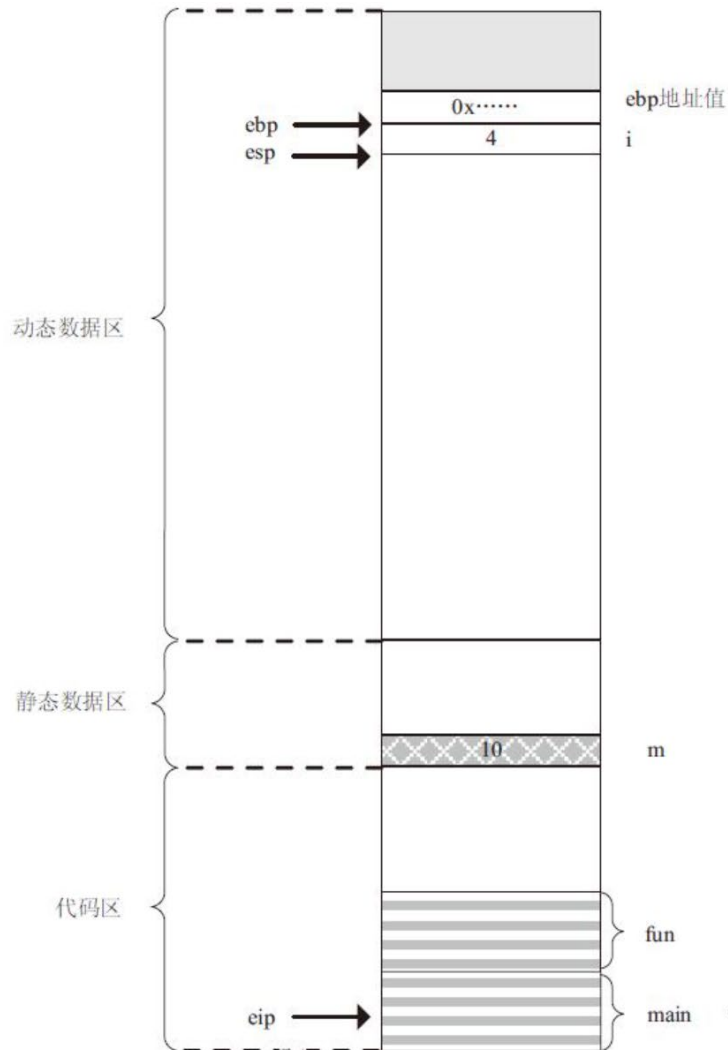
在对返回地址保存之后，开始为main构建栈，main的栈构建情况如右图所示。注意，此事ebp原来指向的值已经被存起来了，所以esp就自动向栈顶移动，同时，ebp需要用来指向main的栈底。此时由于没有数据在栈中，ebp和esp的位置其实是重叠的。

```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}

int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```



# 一个简单的例子



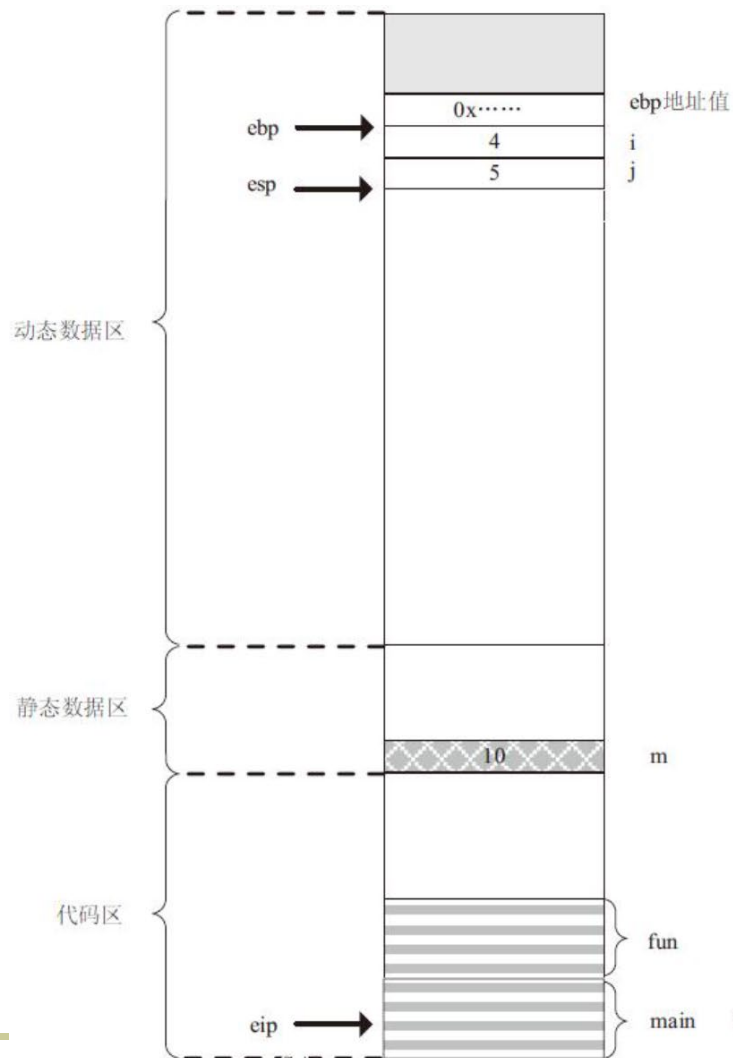
程序继续执行，第一个语句，**int i = 4** 被执行。注意到，对**main**来说，**i** 是一个局部变量，那么，其初始值**4**，被压入栈中。特别注意栈顶指针**esp**的变化。

```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}

int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```



# 一个简单的例子



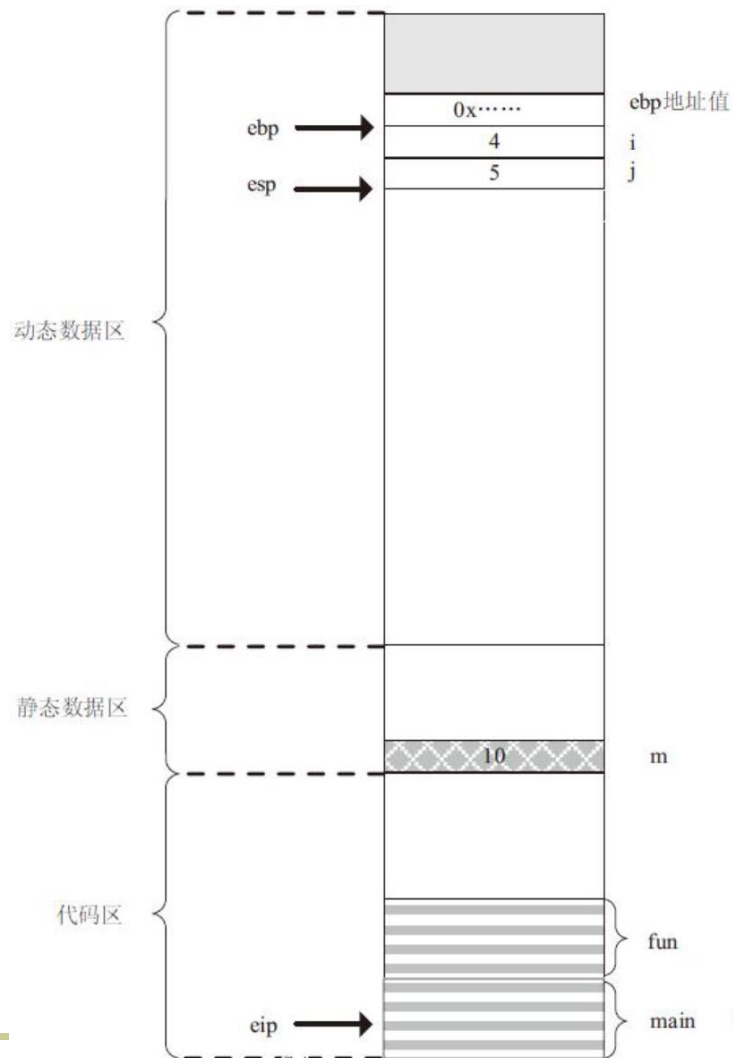
同样的道理， `int j = 5`被执行，5被压入到栈。

```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}

int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```



# 一个简单的例子



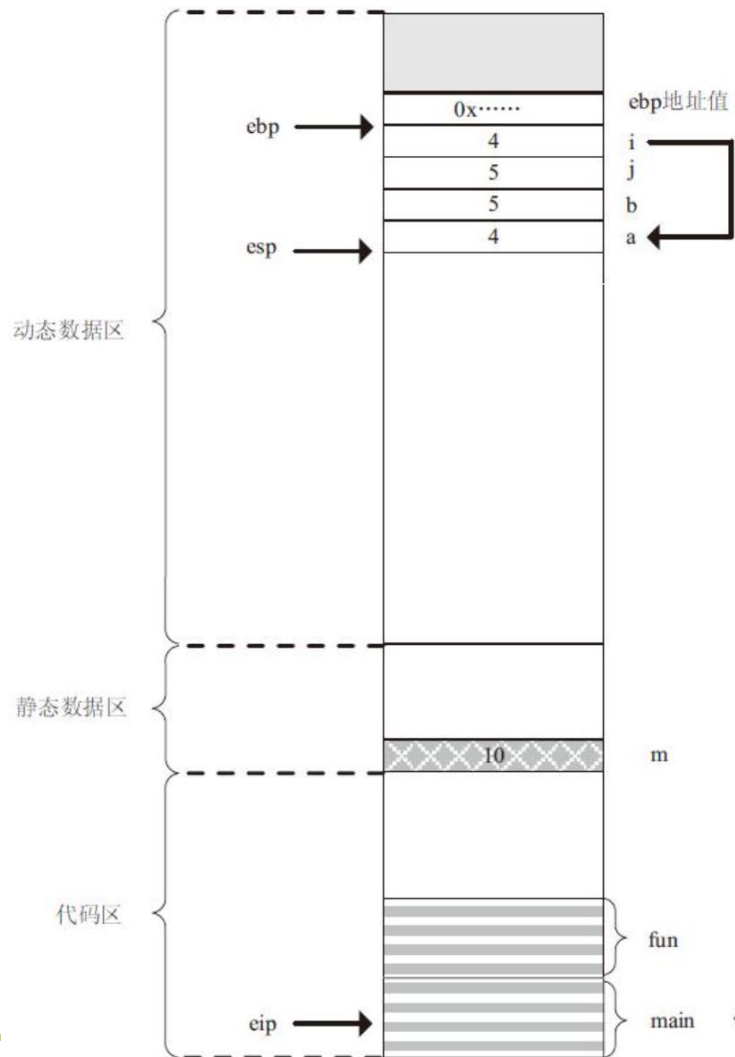
同样的道理， `int j = 5`被执行，5被压入到栈。

```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}

int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```



# 一个简单的例子



下一步，程序执行到调用函数**fun**的指令。特别注意，此处，我们对**fun**的调用，首先需要完成的，是传参指令。

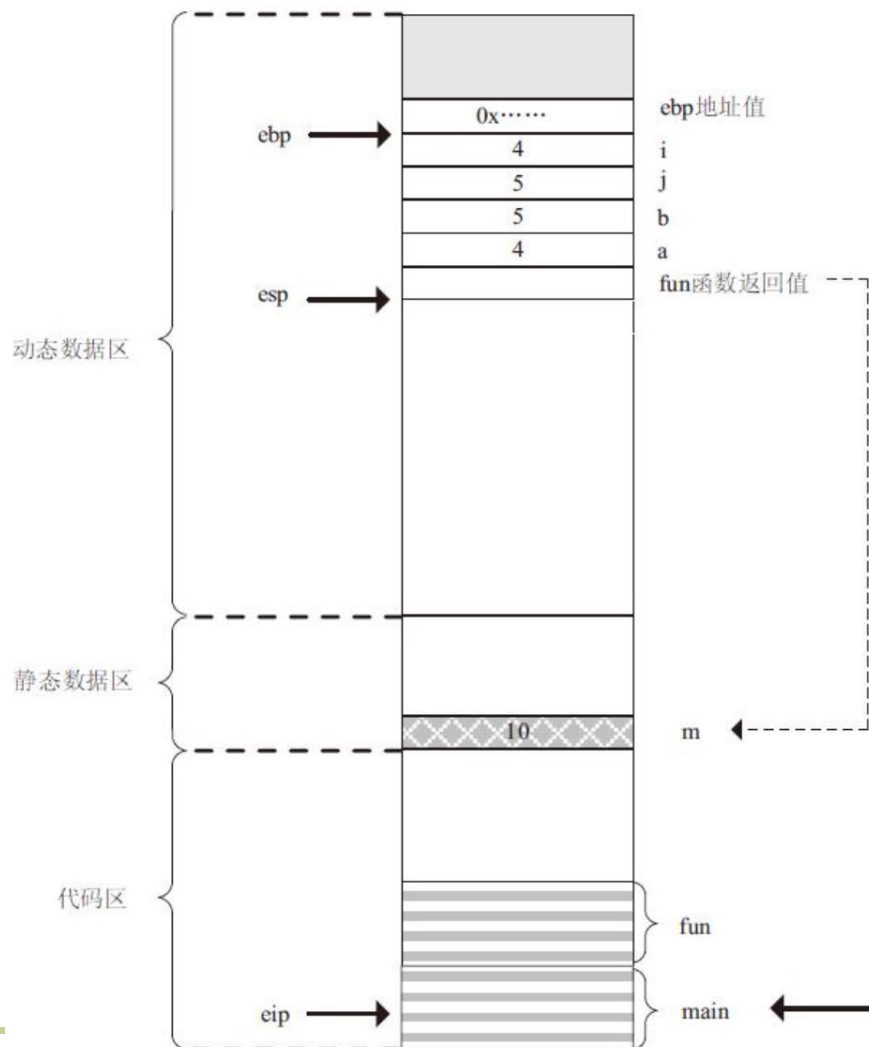
特别注意，在**C**语言中，参数入栈顺序与传参书写顺序正好相反。参数**b**先入栈，**b**的数值是局部变量**j**的数值**5**。因此，情况如右所示。

```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}
```

```
int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```



# 一个简单的例子



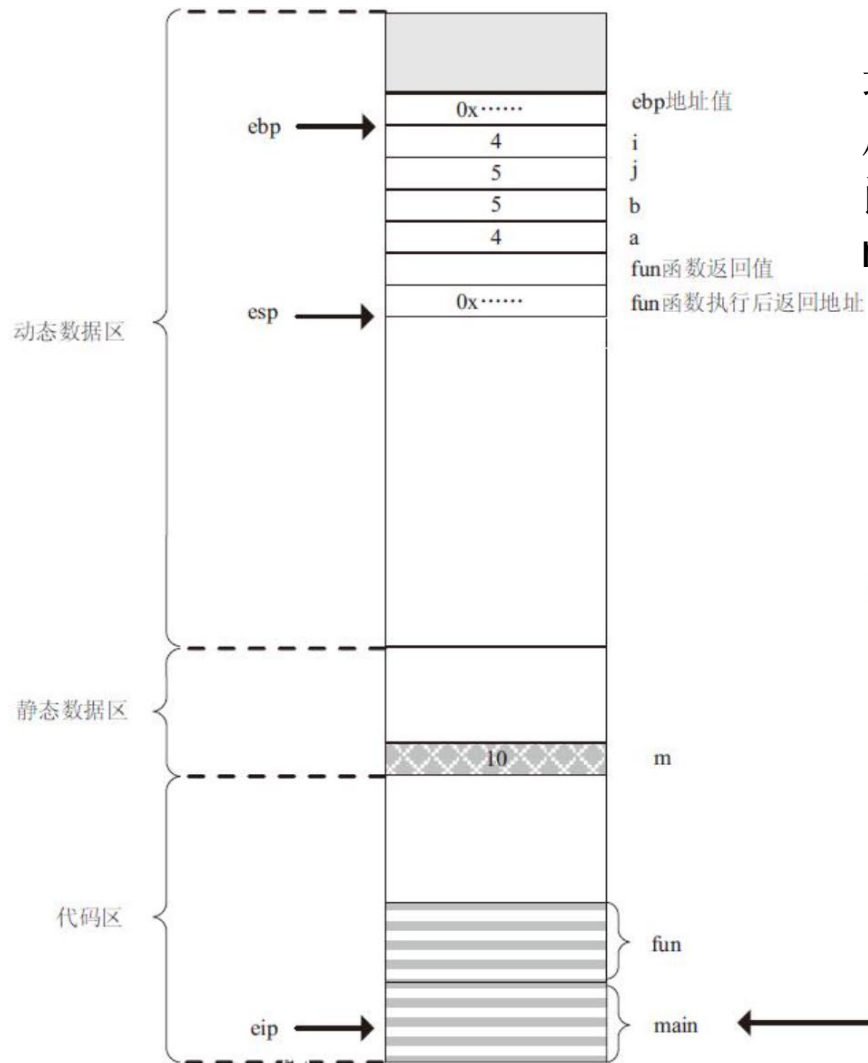
程序继续执行，那么应该压入的是**fun**函数的返回值，后面返回值应该是传递给**m**。

```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}

int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```



# 一个简单的例子



最后一步，就是把**fun**函数执行返回后的地址压入栈中，以便**fun**函数执行后能返回到**main**函数中继续正常执行。回想我们一开始执行**main**函数的情况。

```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}
```

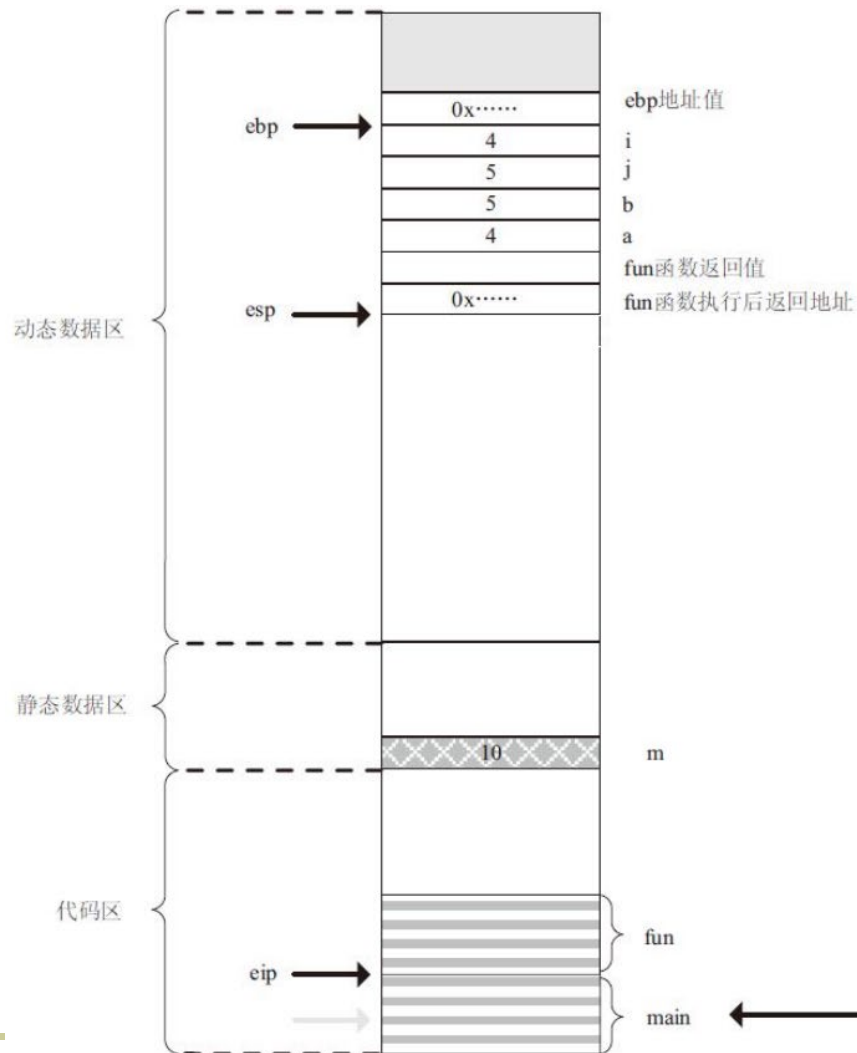
```
int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```







# 一个简单的例子

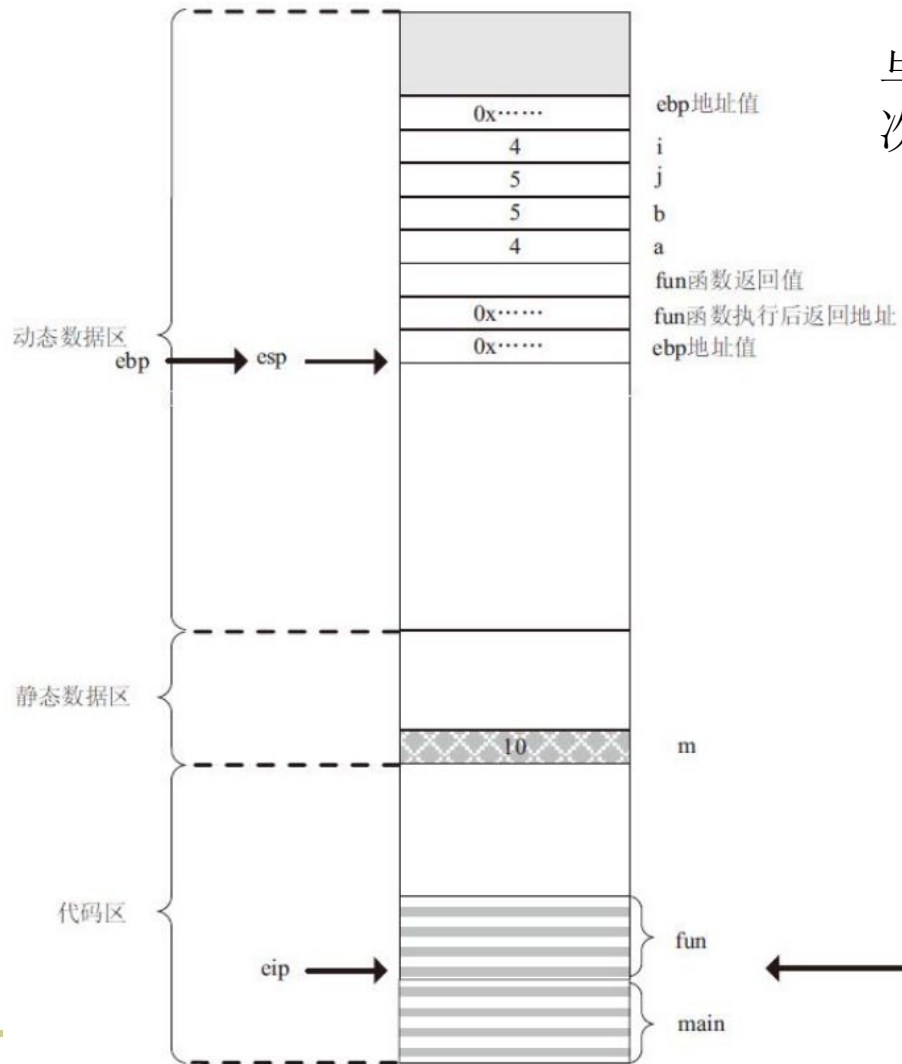


在保存好fun函数执行完毕的返回地址后，eip跳转到fun的代码区进行执行，如右图所示。  
后续则需要构建fun的栈。

```
int fun(int a,int b);  
int m=10;  
int main()  
{  
    int i=4;  
    int j=5;  
    m = fun(i,j);  
    return 0;  
}  
  
int fun(int a,int b)  
{  
    int c=0;  
    c=a+b;  
    return c;  
}
```



# 一个简单的例子



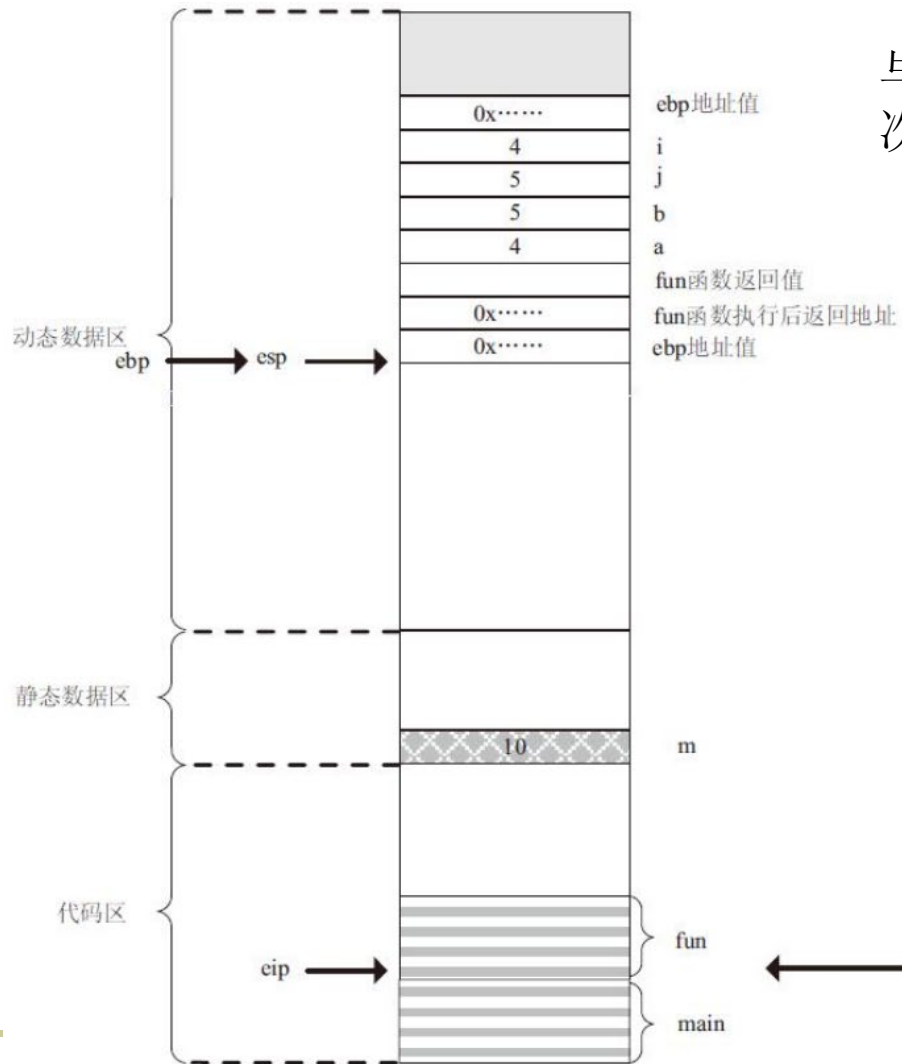
与前面我们调用main函数相似，ebp与esp再次指向同一个位置，此时fun的栈是空的。

```
int fun(int a,int b);  
int m=10;  
int main()  
{  
    int i=4;  
    int j=5;  
    m = fun(i,j);  
    return 0;  
}
```

```
int fun(int a,int b)  
{  
    int c=0;  
    c=a+b;  
    return c;  
}
```



# 一个简单的例子



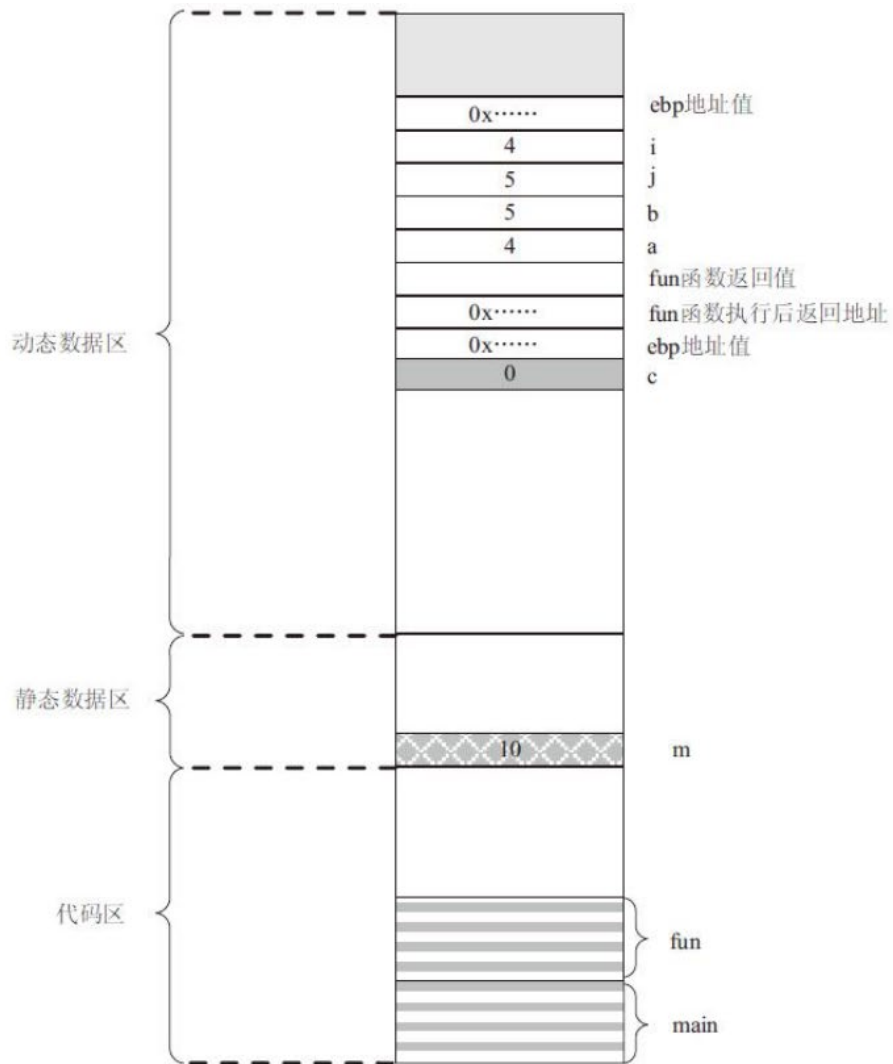
与前面我们调用main函数相似，ebp与esp再次指向同一个位置，此时fun的栈是空的。

```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}
```

```
int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```



# 一个简单的例子



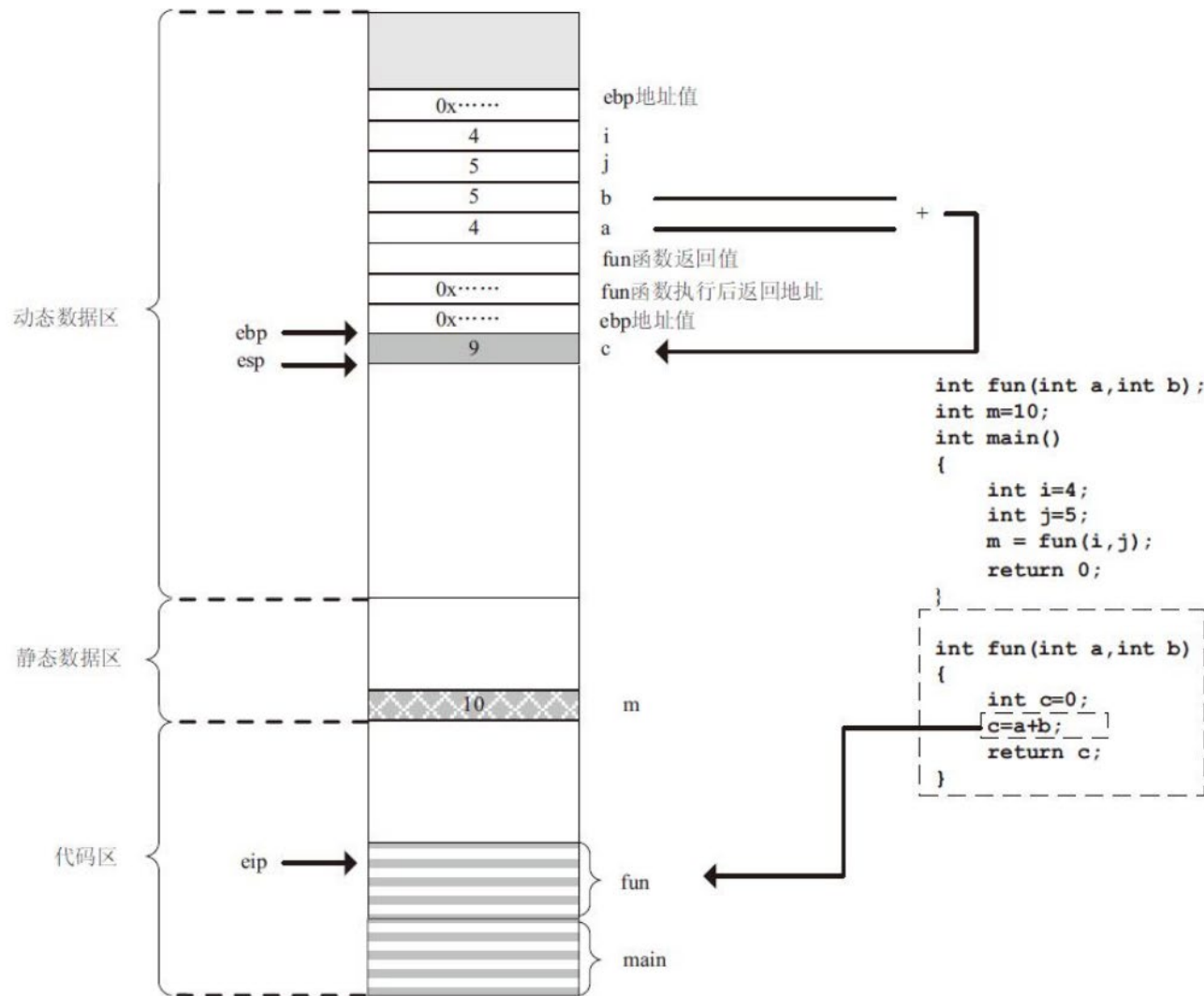
与前面的情况类似，**c**会被压入**fun**的栈中。后续是若干运算指令。

```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}

int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```



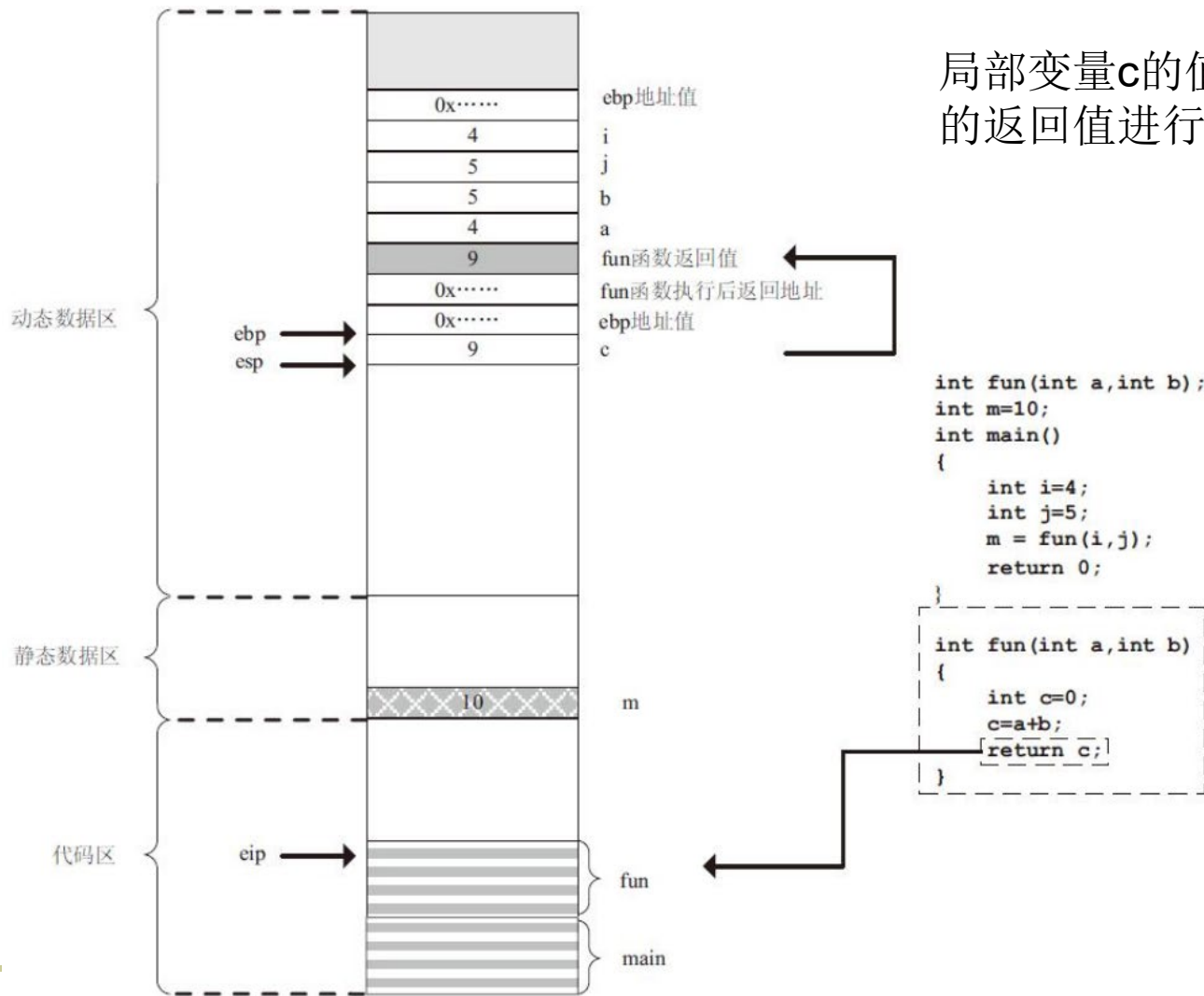
# 一个简单的例子



在执行`a+b`的过程中，我们可以看到，在运算指令的执行过程中，可以通过一系列的偏移计算，得到运算指令需要的数值。运算指令执行完毕后，赋值给局部变量`c`。



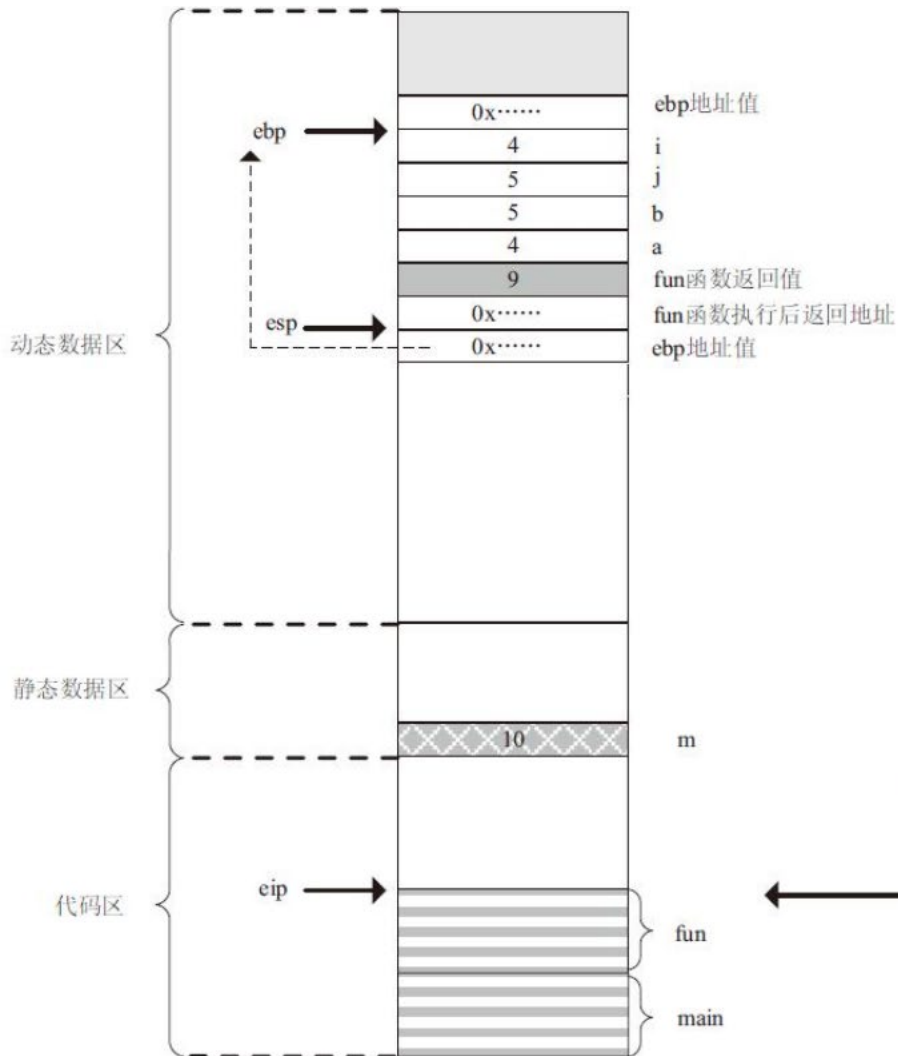
# 一个简单的例子



局部变量**c**的值作为返回值，对**fun**函数的返回值进行赋值。



# 一个简单的例子



Fun 函数执行完毕，要回复main函数调用fun函数的现场，这一现场包括两个部分，一部分是main函数的栈要回复，包括栈顶和栈底，另一部分是要找到fun函数执行后的返回地址，然后再跳转到那里去继续执行。

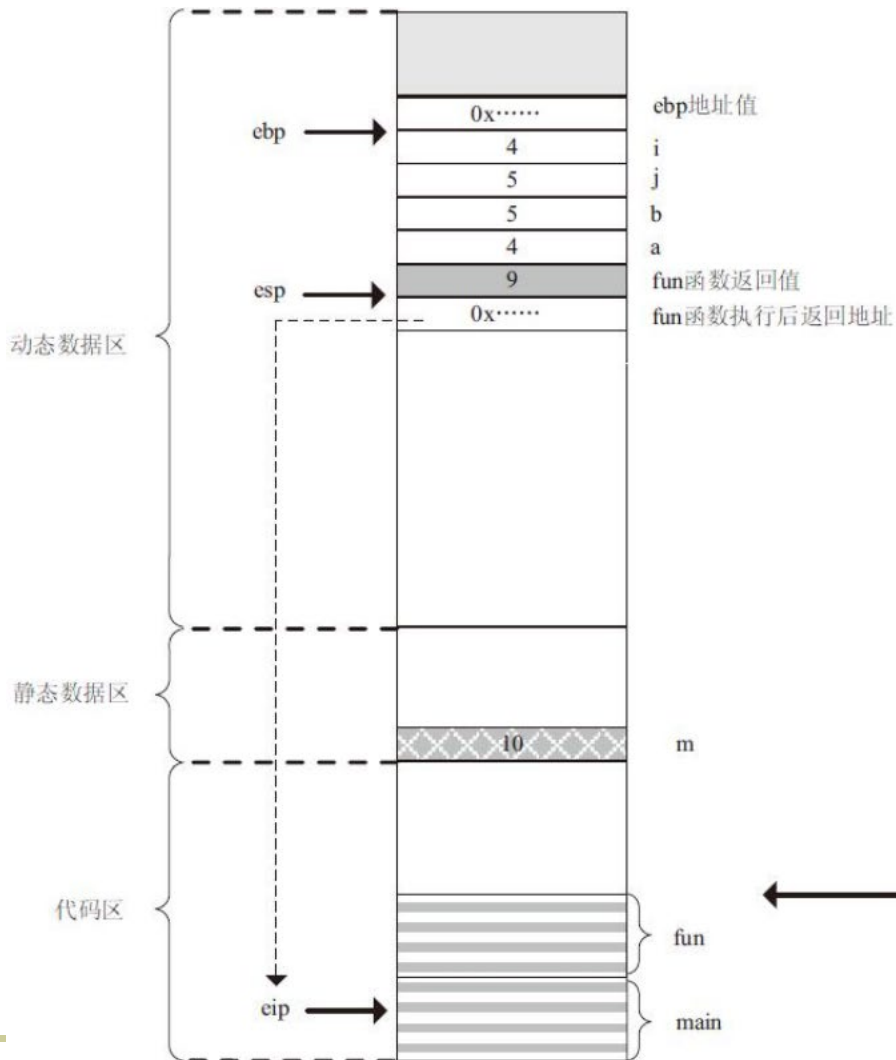
```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}
```

```
int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```





# 一个简单的例子



在ebp恢复后，在esp自动退栈。退栈后指向函数fun执行后的返回地址，然后执行ret指令，该指令将esp此时的值反馈给eip，eip则可以找到main中的下一条指令。

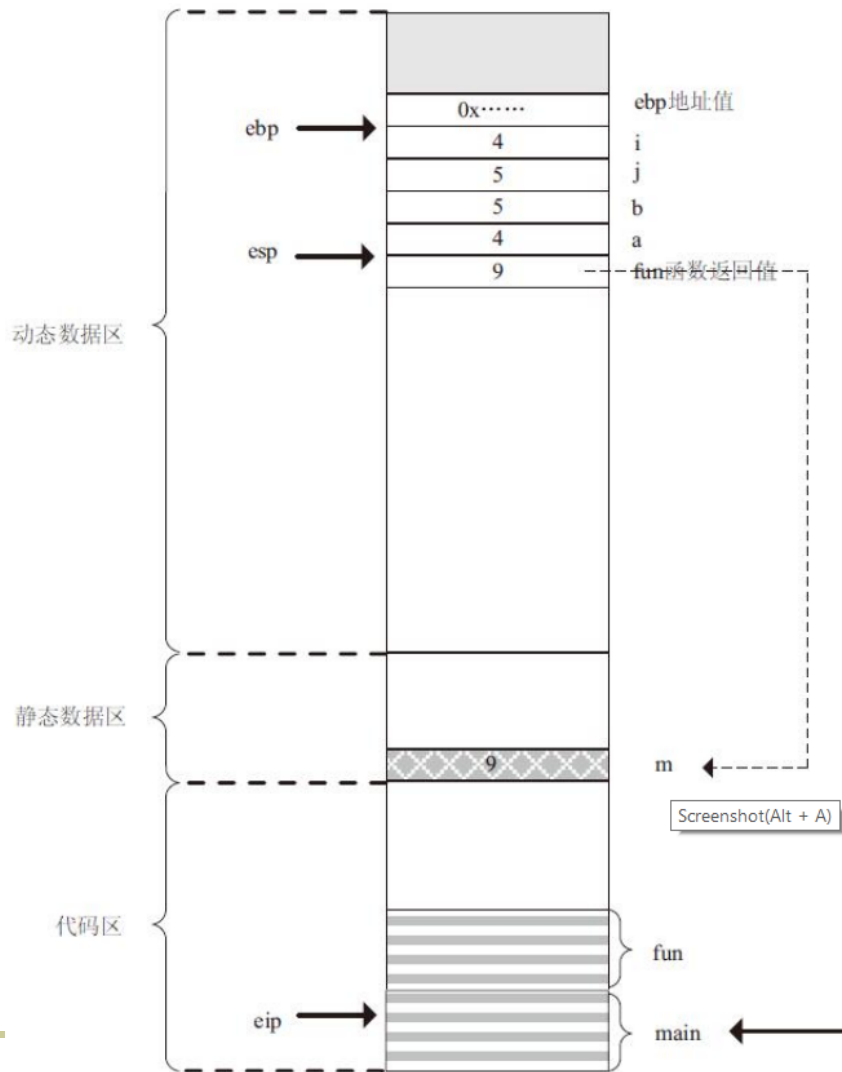
```
int fun(int a,int b);  
int m=10;  
int main()  
{  
    int i=4;  
    int j=5;  
    m = fun(i,j);  
    return 0;  
}
```

```
int fun(int a,int b)  
{  
    int c=0;  
    c=a+b;  
    return c;  
}
```





# 一个简单的例子



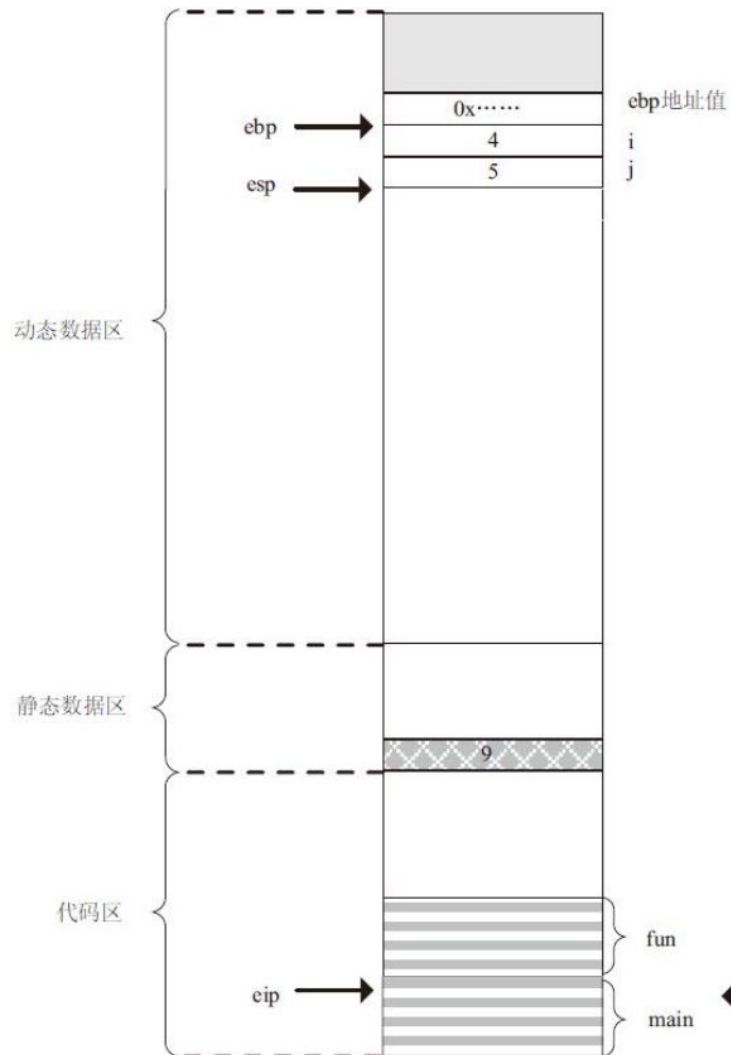
eip找到正确的后续执行指令之后，即将返回值赋值给正确的局部变量，即m。如右图所示。

```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}

int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```



# 一个简单的例子



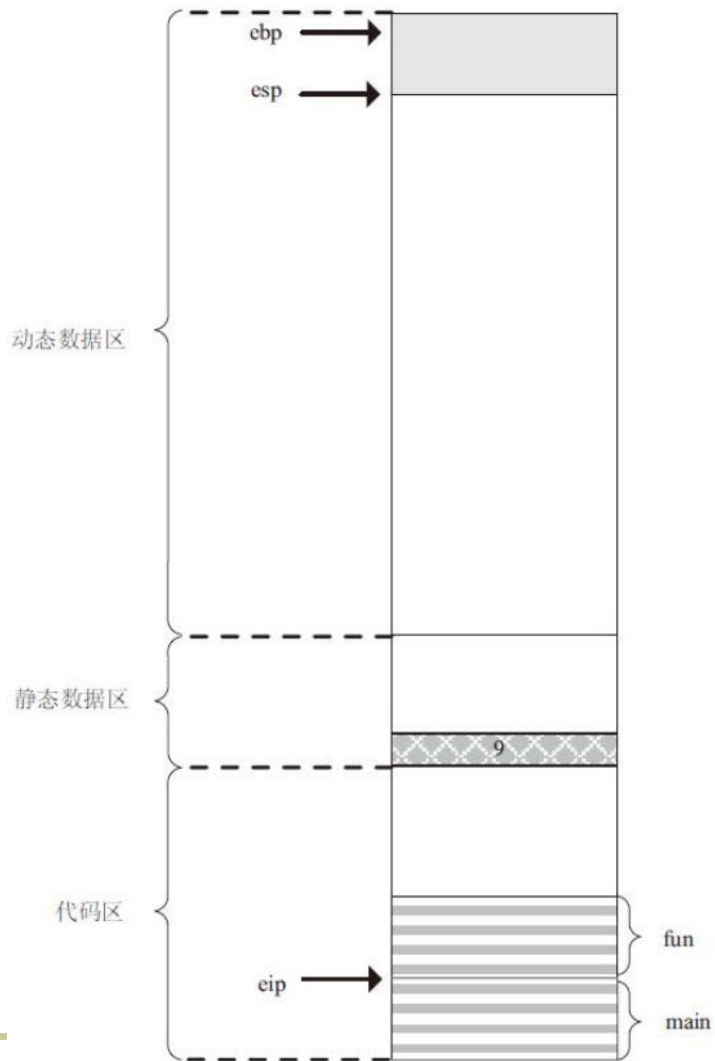
注意到，在完成给`m`的赋值之后，`fun`的调用相关的操作全部完毕，其栈已经没有任何存在的意义，全部清除。结果如右图所示。

```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}

int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```



# 一个简单的例子



同理，在执行完`main`相关的`ret`指令之后，也需要对`main`函数的栈进行清除，清楚后的结果如右图所示。至此，整个程序执行完毕。

```
int fun(int a,int b);
int m=10;
int main()
{
    int i=4;
    int j=5;
    m = fun(i,j);
    return 0;
}

int fun(int a,int b)
{
    int c=0;
    c=a+b;
    return c;
}
```