第三讲
递归算法
Lecture 3
Recursive Algorithms

明玉瑞 Yurui Ming

yrming@gmail.com

声明
Disclaimer

- 本讲义在准备过程中由于时间所限，所用材料来源并未规范标示引用来源。所引材料仅用于教学所用，作者无意侵犯原著者之知识版权，所引材料之知识版权均归原著者所有；若原著者介意之，请联系作者更正及删除。

  The time limit during the preparation of these slides incurs the situation that not all the sources of the used materials (texts or images) are properly referenced or clearly manifested. However, all materials in these slides are solely for teaching and the author is with no intention to infringe the copyright bestowed on the original authors or manufacturers. All credits go to corresponding IP holders. Please address the author for any concern for remedy including deletion.

分而治之
Divide and Conquer

- 为求一个大规模问题的解，将其分解为若干个子问题，例如两个，而这些子问题的规模大体相当；基于分别求出的这些子问题的解，得到初始问题的解。此即为分而治之的思想。

  To find a solution for a large-scale problem, a usual trick is to divide it into several sub-problems with comparable small scales, for example, two sub-problems to deal with them separately. The final solution for the original problem is built upon the solutions of these sub-problems.

- 根据分的方式，有时会做一些区分。当分成的两部分的量几乎等同的时候，就是传统的分而治之，当分的两部分的量很不对等时，可能会称为减而治之。

  There might exist different terminologies in describing the division strategy. Consider the case of dividing into two parts. When the scales of these two parts are almost equal, this is the traditional divide and conquer. If these two parts are obviously unbalanced, we call it reduce and conquer.

分而治之
Divide and Conquer

- 分而治之算法通常包括一个或者多个递归方法的调用，当这些调用将数据分隔成为独立的集合从而处理较小集合的时候，分而治之的策略将会有很高的效率，而在数据进行分解的时候，分而治之的策略可能会产生大量的重复计算，从而导致性能的降低。

  To find a solution for a large-scale problem, a usual trick is to divide it into several sub-problems with comparable small scales, for example, two sub-problems to deal with them separately. The final solution for the original problem is built upon the solutions of these sub-problems.

- 函数调用需要栈变量维护调用和返回时的相关信息，不恰当的递归调用可能会导致栈溢出，存在潜在的安全隐患。

  Calling a function needs stack variables to maintain the information prior to calling and upon return, and improper recursion might incur the possibility of stack overflow, and consequently introduce security risks.

递归
Recursion

- 那么什么是递归呢？

```
void func()
{
	printf("从前有座山，山上有座庙，庙里有个老和尚，老和尚在给小和尚讲故事：");
	func();
}
```

- 输出：

从前有座山，山上有座庙，庙里有个老和尚，老和尚在给小和尚讲故事：从前有座山，山上有座庙，庙里有个老和尚，老和尚在给小和尚讲故事：从前有座山，山上有座庙，庙里有个老和尚，老和尚在给小和尚讲故事：从前有座山，山上有座庙，庙里有个老和尚，老和尚在给小和尚讲故事：…

递归
Recursion

- What’s recursion?

```
void func()
{
    printf(“Long long time ago, in a temple built on a hill, there lived two monks, and the old monk was telling a story to the little monk: ”;
    func();
}
```

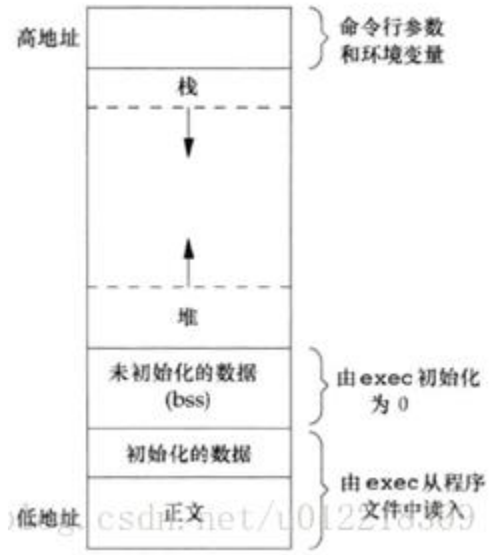
- Output:

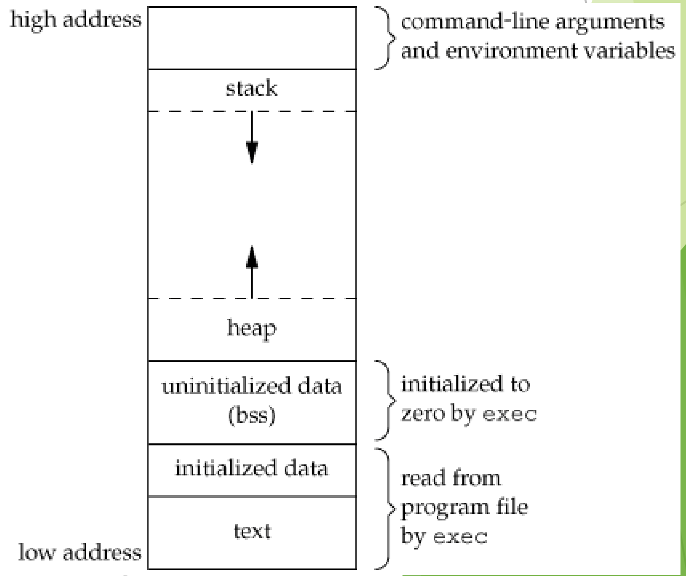
```
Long long time ago, in a temple built on a hill, there lived two monks, and the old monk was telling a story to the little monk: Long long time ago, in a temple built on a hill, there lived two monks, and the old monk was telling a story to the little monk: Long long time ago, in a temple built on a hill, there lived two monks, and the old monk was telling a story to the little monk: …
```

函数调用约定
Convention of Function Calling

- 当源代码被编译链接之后，会形成可执行程序，当被加载程序加载时，会根据可执行程序内容，加载进内存的不同地址，或内存段，如下图所示：

  After compiling and linking, the source code is converted into executable. Upon loading into the memory by linker and loader, the content of the executable will be loaded into different memory address or memory segment according to the property of the content, as illustrated below:





函数调用约定
Convention of Function Calling

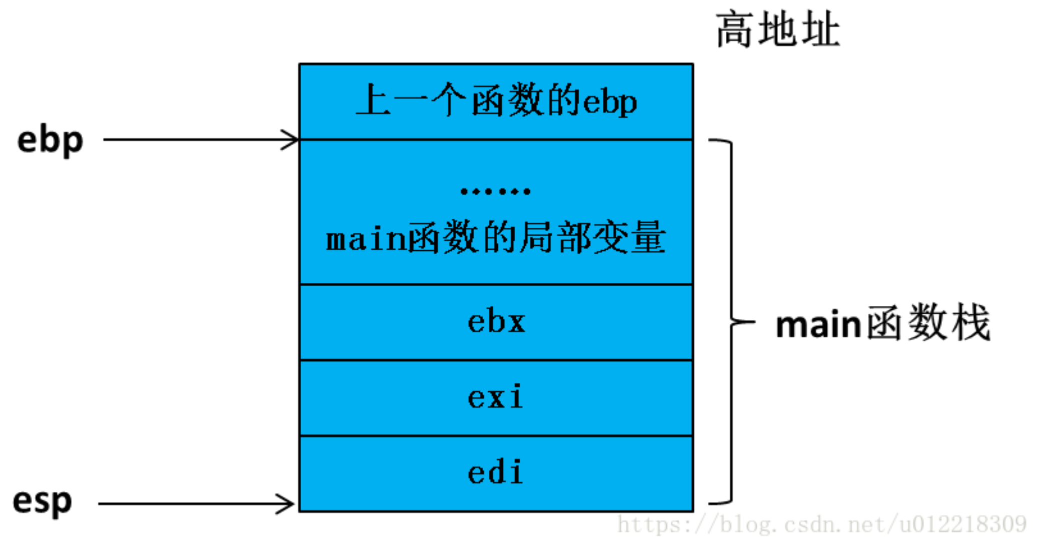
```
int sum(int a, int b)
{
    int c = a + b;
    return c;
}

int main()
{
    int a = 10, b = 20;
    int ret = sum(a, b);
    return 0;
}
```

函数调用约定
Convention of Function Calling

- main函数中在sum调用之前的栈如图：

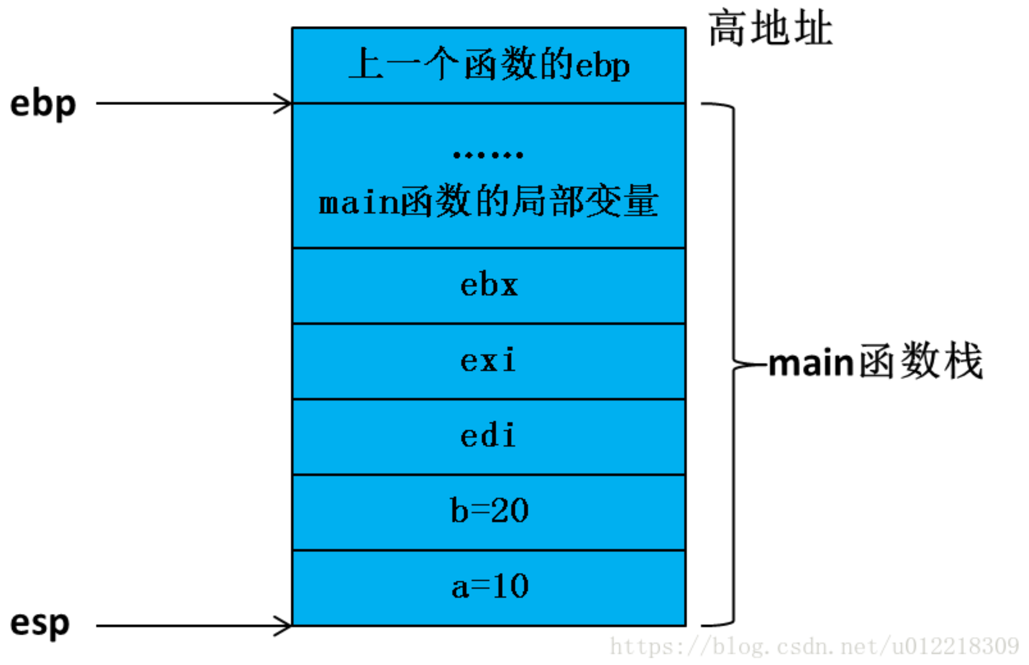
  Stack before calling the sum function in main:



函数调用约定
Convention of Function Calling

- 当准备sum调用时，第一步，将参数压入栈：

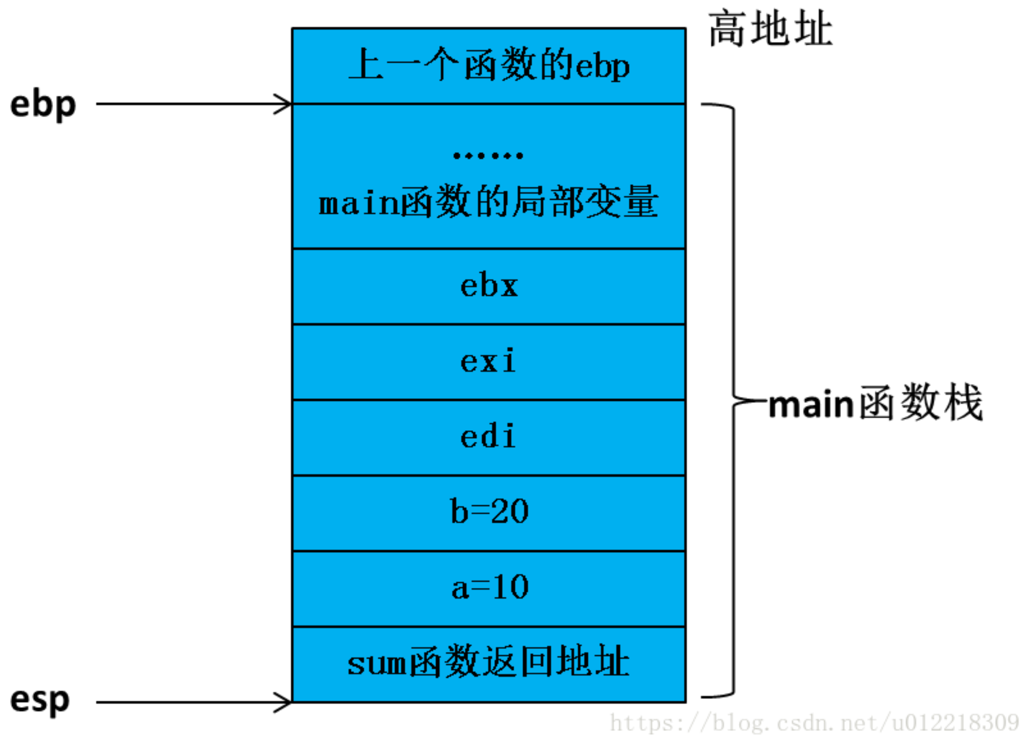
  Upon calling calling the sum function, step 1, to push the function parameters into stack:



函数调用约定
Convention of Function Calling

- 第二步，将返回地址压入栈：

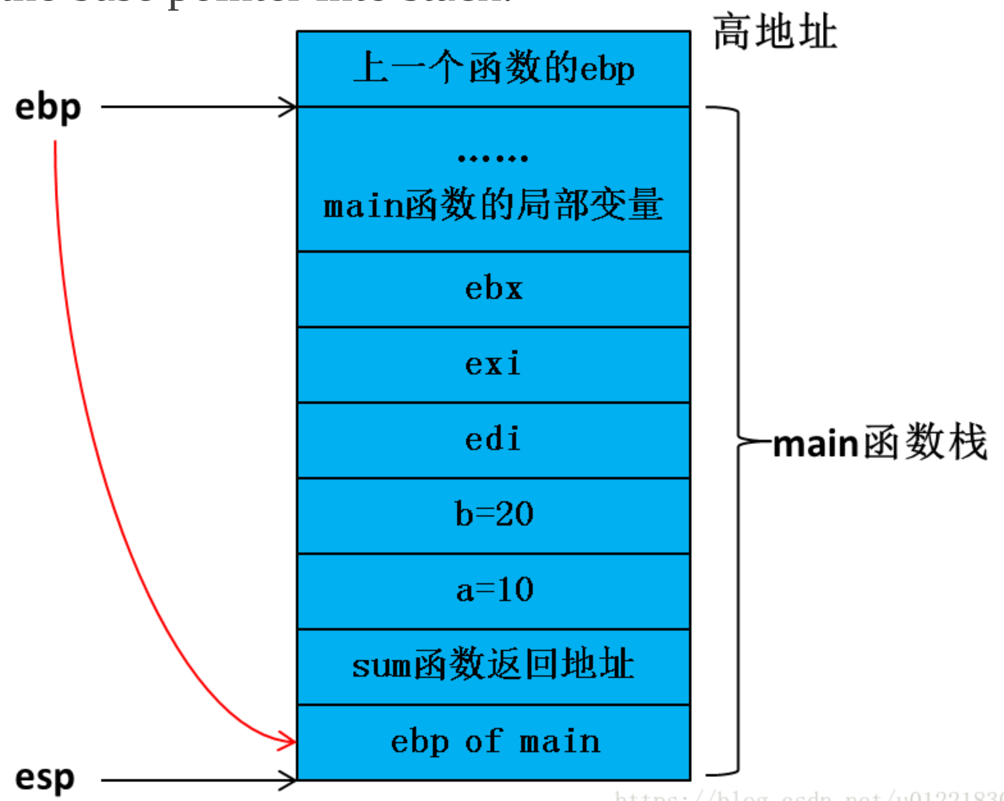
  Step 2, to push the return address into stack:



函数调用约定
Convention of Function Calling

- 第三步，将基指针压入栈：

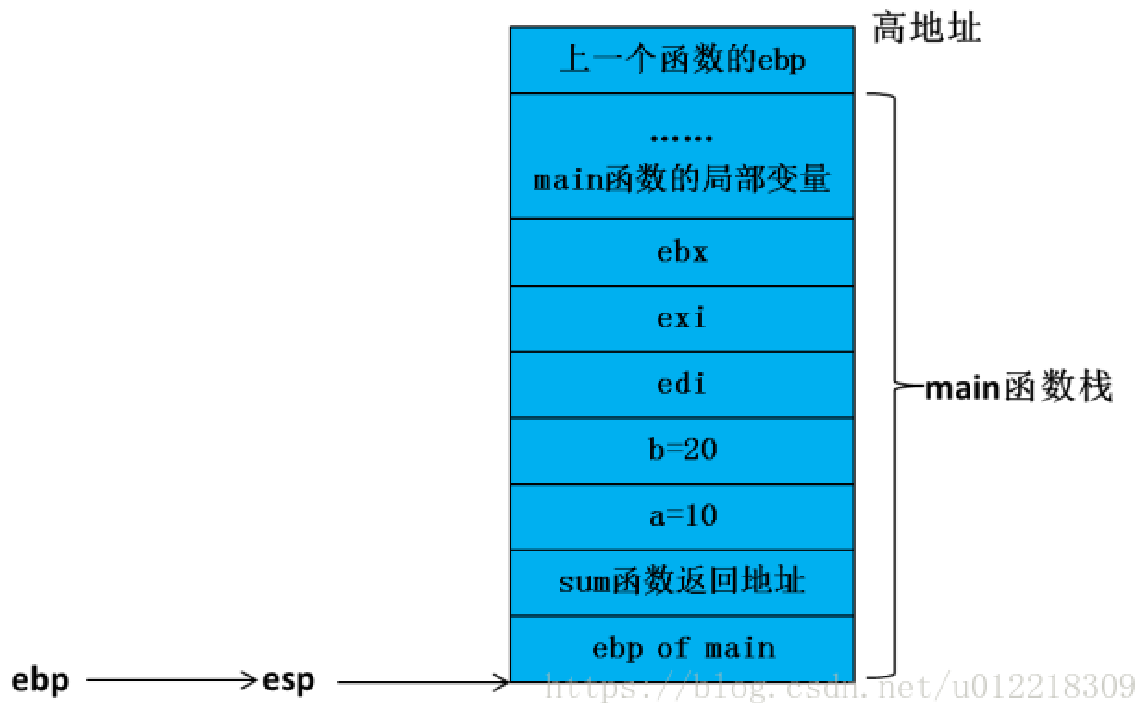
  Step 3, to push the base pointer into stack:



函数调用约定
Convention of Function Calling

- 第四步，将EBP的指针赋给ESP：

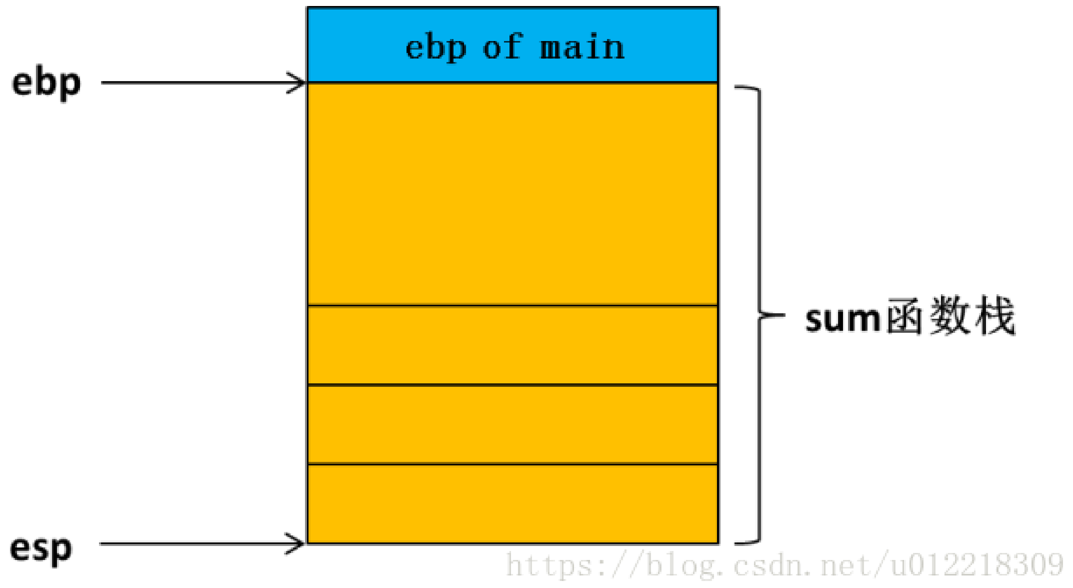
  Step 4, assign the value of EBP to ESP:



函数调用约定
Convention of Function Calling

- 第五步，将ESP下移动一段空间创建sum函数的栈的栈帧：

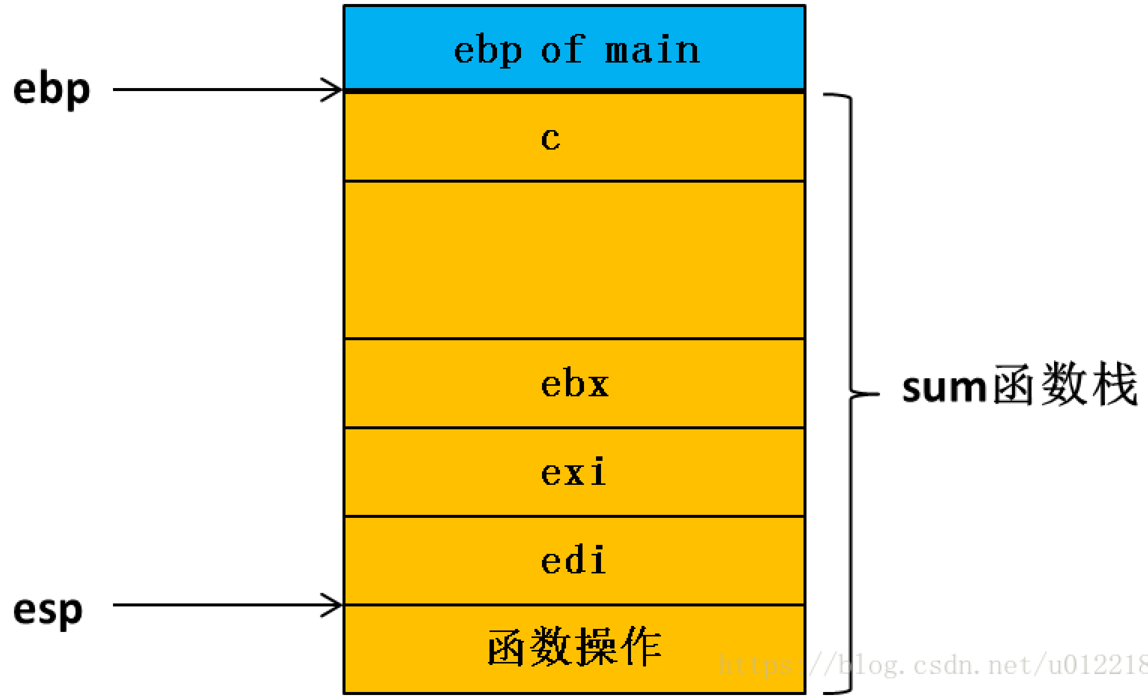
  Step 5, decrease ESP by some displacement to reserve space for stack frame:



函数调用约定
Convention of Function Calling

- 第六步，将进行加法操作，并将结果c放入[ebp-4]位置：

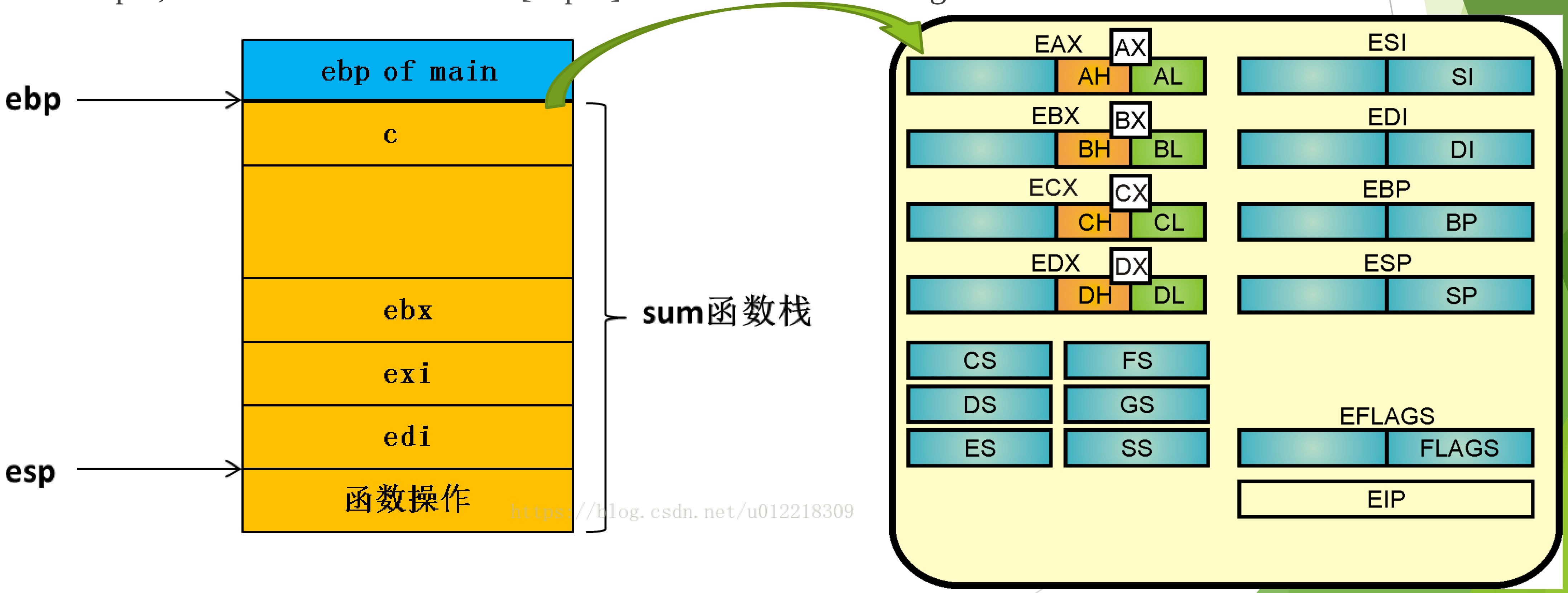
  Step 6, perform the addition and store the result to position [ebp-4]:



函数调用约定
Convention of Function Calling

- 第七步，将 [ebp-4]位置所存储之值放入EAX之中，准备返回：

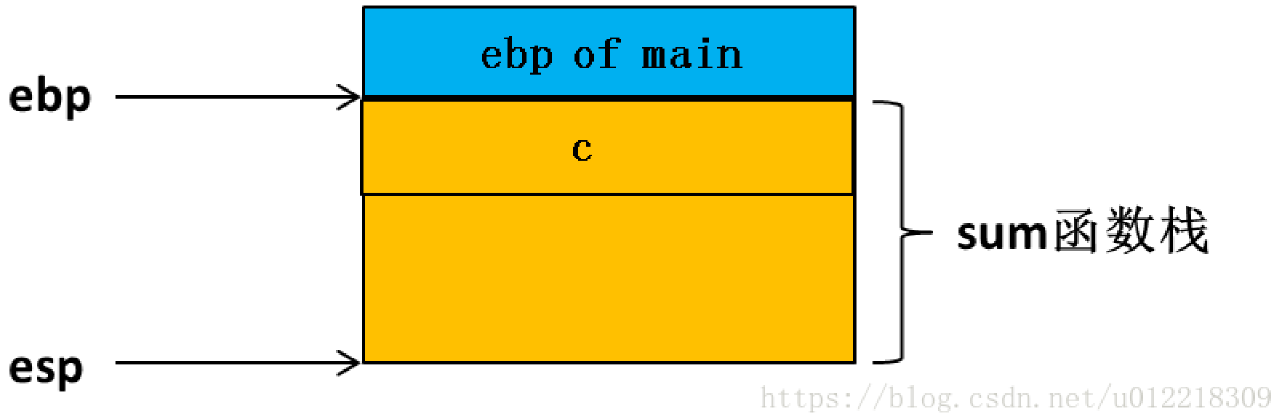
  Step 7, move the value stored at [ebp-4] into EAX for returning:



函数调用约定
Convention of Function Calling

- 第八步，依次弹出之前所压入之其它寄存器之值：

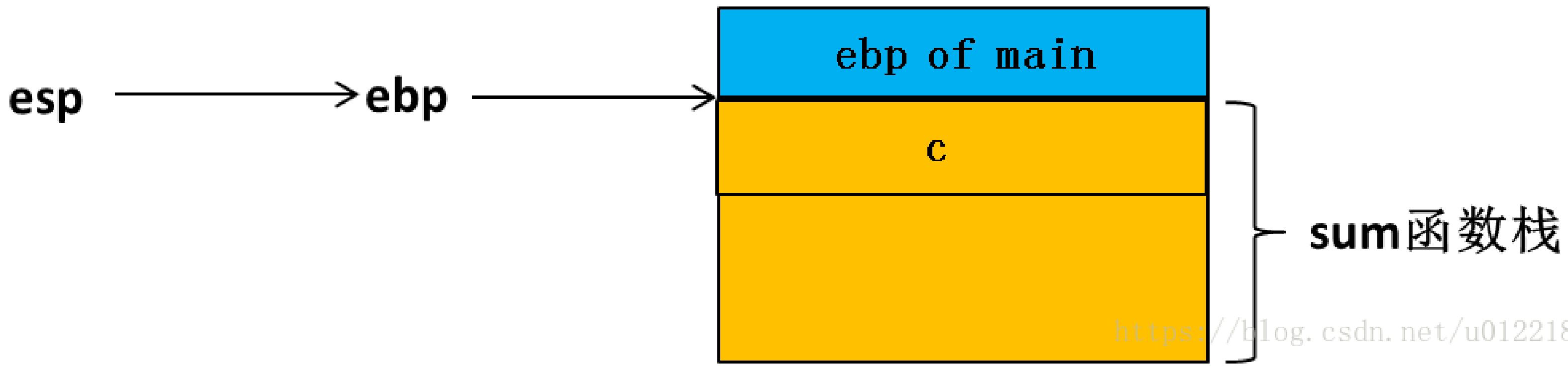
  Step 8, pop other register values that pushed previously:



函数调用约定
Convention of Function Calling

- 第九步，将ebp的值赋给esp，也就等于将esp指向ebp，销毁sum函数栈帧：

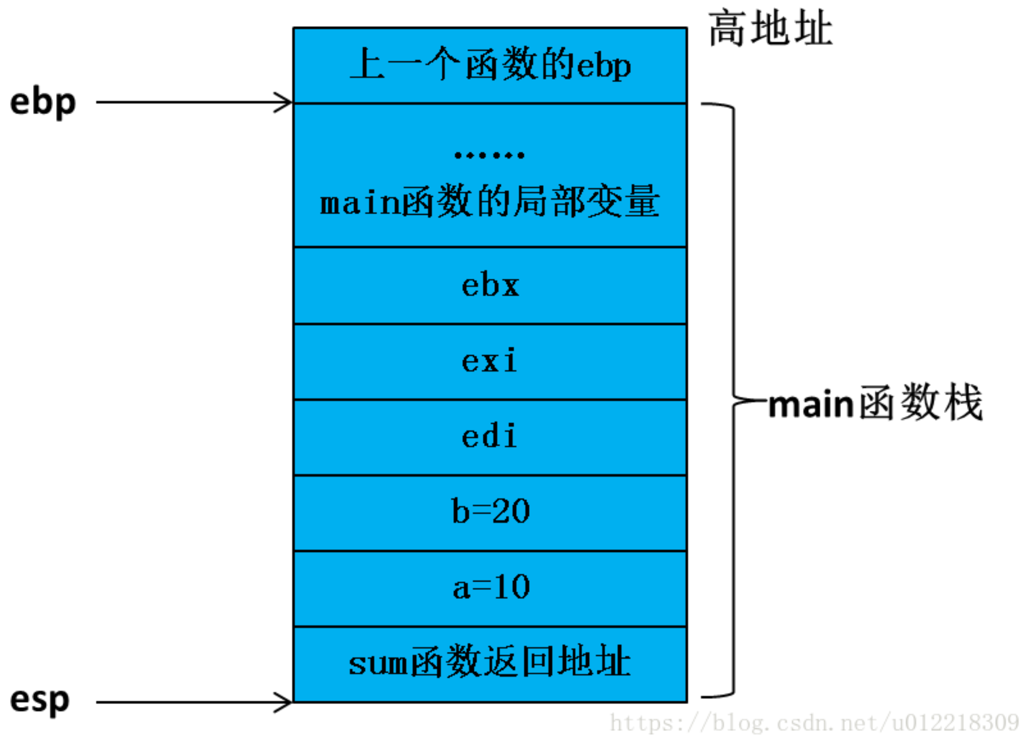
  Step 9, pop other register values that pushed previously:



函数调用约定
Convention of Function Calling

- 第十步，返回到main函数中，接着执行下一条语句：

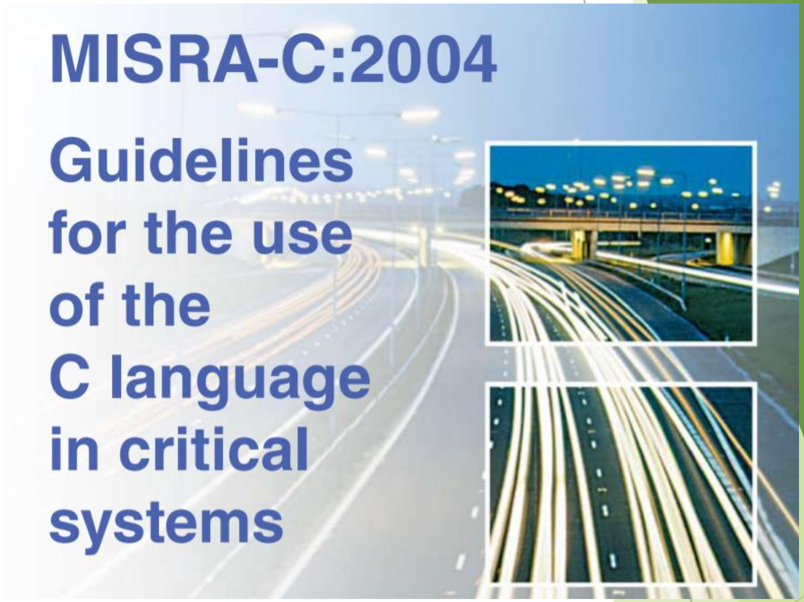
  Step 10, return to main function to continue execute the next instruction:

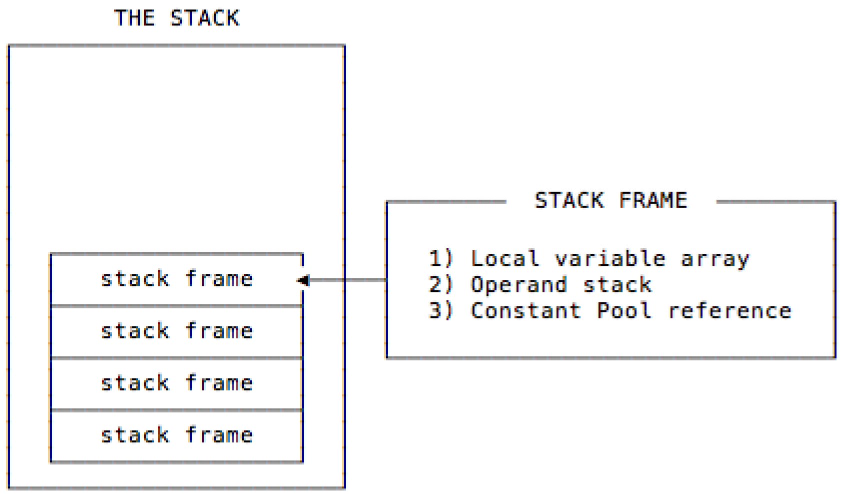


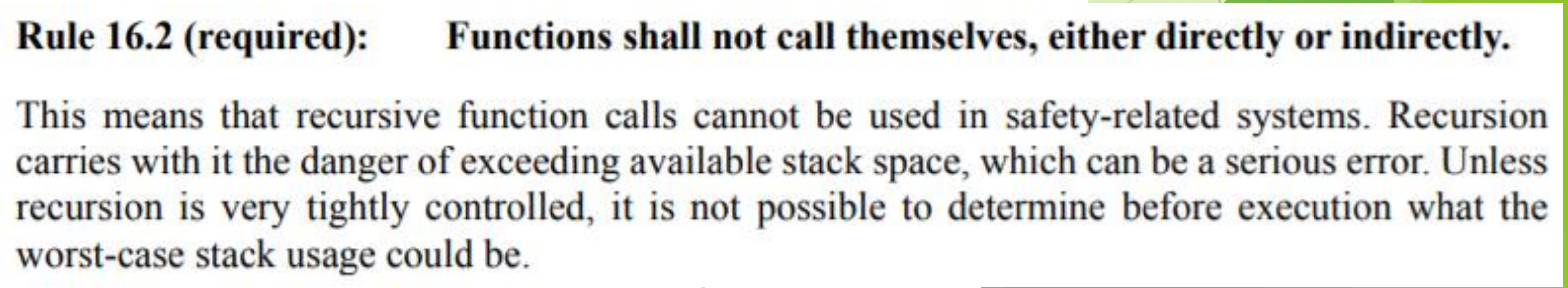
函数调用约定
Convention of Function Calling

- 如果递归调用，有可能导致很深的栈；因此在一些领域，比如汽车软件，考虑到安全性，会禁用递归。

  Recursion might incur exceeding depth of stacks, so in some domains, such as automotive software, recursion is prohibited from the security perspective.


MISRA-C:2004
Guidelines for the use of the C language in critical systems




Rule 16.2 (required): Functions shall not call themselves, either directly or indirectly.

This means that recursive function calls cannot be used in safety-related systems. Recursion carries with it the danger of exceeding available stack space, which can be a serious error. Unless recursion is very tightly controlled, it is not possible to determine before execution what the worst-case stack usage could be.

递归的类型
Types of Recursion

- 递归有两种类型，一种是常见的递归，一种是尾递归

  There are two types of recursion, the first is the normal recursion, and the second is called the tail recursion.

- 普通递归的例子（Example of normal recursion）

```
def recsum(x):
  if x == 1:
    return x
  else:
    return x + recsum(x - 1)
```

```
recsum(5)
5 + recsum(4)
5 + (4 + recsum(3))
5 + (4 + (3 + recsum(2)))
5 + (4 + (3 + (2 + recsum(1))))
5 + (4 + (3 + (2 + 1)))
5 + (4 + (3 + 3))
5 + (4 + 6)
5 + 10
15
```

递归的类型
Types of Recursion

- 尾递归的例子（Tail-recursion example）
- 尾递归的栈帧大小恒定，所以有时候可以考虑把普通递归改为尾巴递归。

  The stack frame of tail-recursion is fixed, so sometimes, it is good to rewrite the normal recursion as tail-recursion.

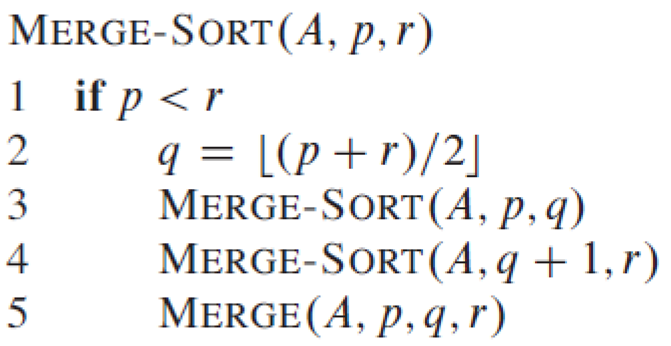
```python
def tailrecsum(x, running_total=0):
  if x == 0:
    return running_total
  else:
    return tailrecsum(x - 1, running_total + x)
```

```
tailrecsum(5, 0)
tailrecsum(4, 5)
tailrecsum(3, 9)
tailrecsum(2, 12)
tailrecsum(1, 14)
tailrecsum(0, 15)
15
```

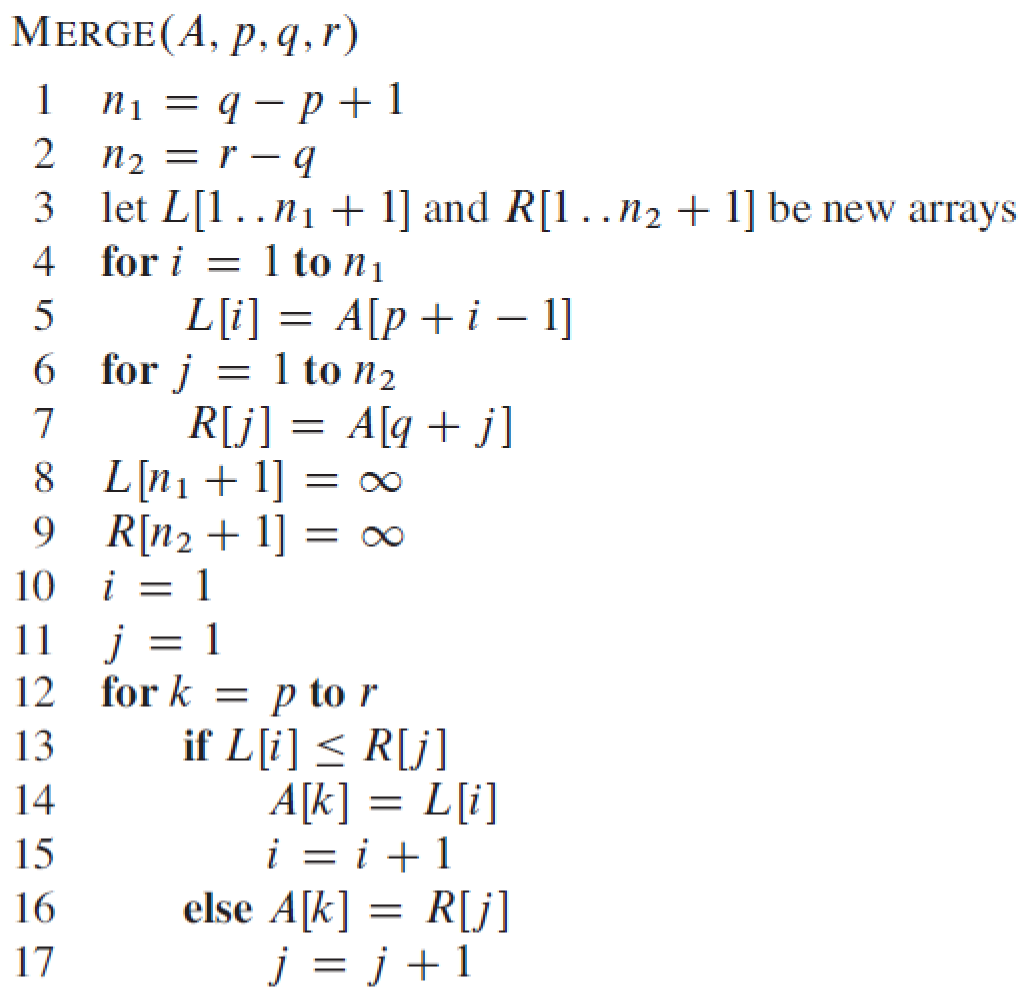
问题
Problems

1. 合并排序的伪代码如下图，请转化为基于STL的C++代码

   The pseudo-code of merge sort is as follows, please implement it in C++ based on STL



```
MERGE-SORT($A, p, r$)
if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT($A, p, q$)
    MERGE-SORT($A, q + 1, r$)
    MERGE($A, p, q, r$)
```



```
MERGE($A, p, q, r$)
$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1 \mathinner{..} n_1 + 1]$ and $R[1 \mathinner{..} n_2 + 1]$ be new arrays
for $i = 1$ to $n_1$
    $L[i] = A[p + i - 1]$
for $j = 1$ to $n_2$
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ to $r$
    if $L[i] \le R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$
```

问题
Problems

2. 对给定的数列，利用递归算法，求数列中的最大值。Python代码如下，请转化为C++代码。

For given array, find the maximal element based on recursive method. The Python script is already given, please re-implement it in C++

```
def findmax(arr):
    if len(arr) == 2: # base case
        return arr[0] if arr[0]>arr[1] else arr[1]
    # recursive case
    return arr[0] if arr[0] > findmax(arr[1:]) else findmax(arr[1:])
```