静态函数

背景： 普通的函数都是跨文件可见的，也就是比如a.c里面有个函数swap(), 在b.c中也可以调用该函数。

静态函数：只能够在当前定义的文件中使用，称为静态函数

语法：

```
static int func (int , int b )
{
 // 函数体

}
```

注意：

- 静态函数用来缩小可见范围，减少与其它文件中同名函数冲突的问题
- 静态函数一般会定义在头文件中， 然后被需要使用该函数的源文件包含即可。

递归函数：

递归函数的概念： 如果一个函数的内部，包含了对自己的调用则称为递归函数。

递归的问题：

1. 阶乘
2. 幂运算
3. 字符反转

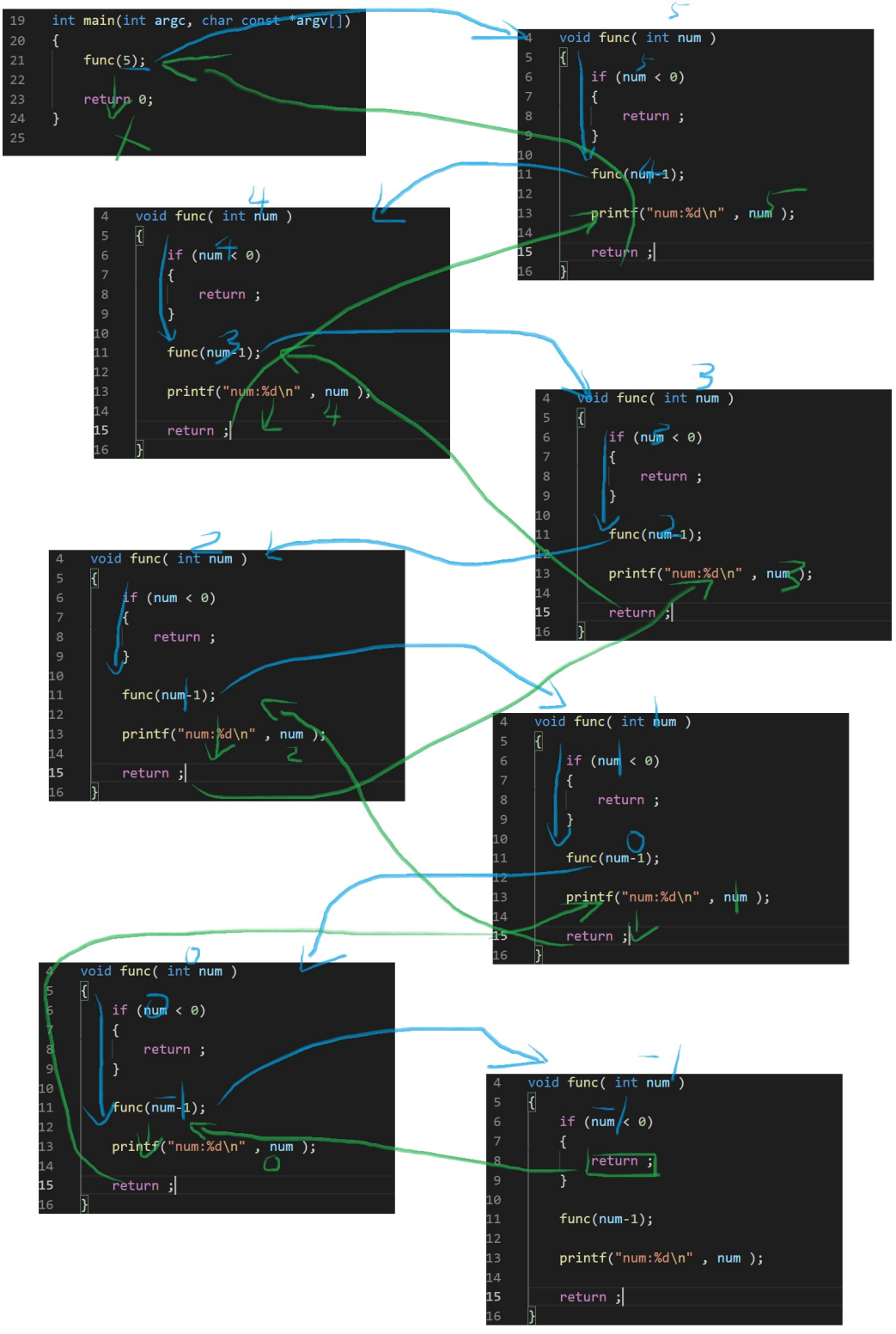
要点：

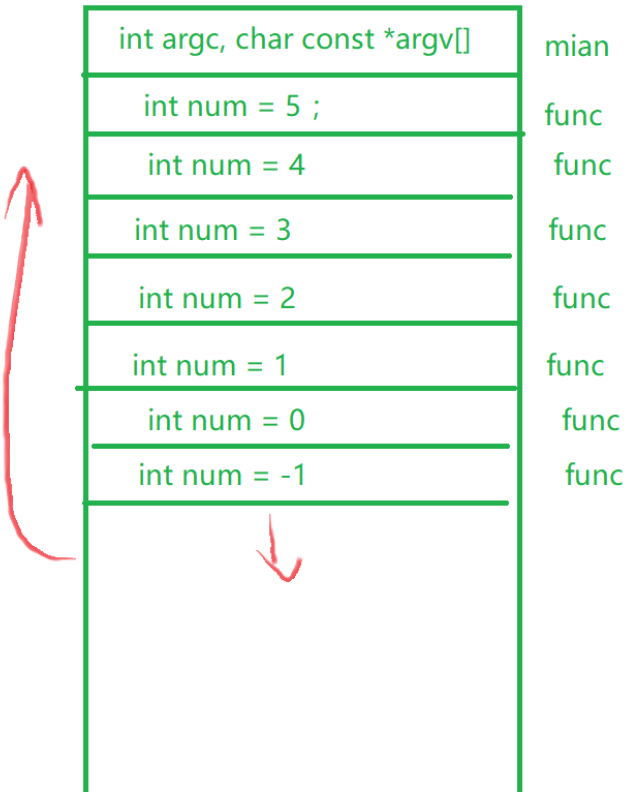
- 只有可以被递归的问题，才可以使用递归函数来实现。
- 递归函数必须有一个可以直接退出的条件，否则会进入无线递归，最后导致栈溢出。
- 递归函数包含了两个过程： 逐渐递进 ， 逐步回归。

例子：

要求使用递归来输出n个自然数

思路： 先输出n-1个自然数， 最后才输出N


int main(int argc, char const *argv[])
{
func(5);
return 0;
}
void func(int num)
{
if (num < 0)
{
return ;
}
func(num-1);
printf("num:%d\n" , num);
return ;
}



递归函数内存变化

总结：

递归的时候函数的栈空间，会随之从上往下不断增长，栈空间会越来越少， 直到耗尽或者开始返回。

在一层一层递进的时候，问题的规模会随之缩小，当达到某一个条件的时候，则开始一层一层回归。

函数指针：

指向一个函数的指针，称为函数指针；

定义语法：

```
int (*p) (int , float ) ;
```

例子：

```

int Printf( int a , float f )
{


    printf("a:%d , f:%f\n" , a , f );

```

```
    return 0 ;
}

int main(int argc, char const *argv[])
{
 // 定义一个函数指针， 名字为 p 。它指向的函数 有一个整型返回值
 // 需要 一个整型参数 以及 浮点参数
    int  (*p) (int , float ) ;
    p = Printf ;  //  函数名其实也是这个函数的入口地址

    Printf(10 , 3.14);

    p(56 ,9.8888);

    return 0;
}
```

指针函数：

一个返回指针的函数，就称为指针函数。

```
int * func(int a , int b)
{
 int kk ;
 /*
 ..
 ...
 */

 return &kk ;
}
```

回调函数：

概念：函数实现方不方便调用该函数，而由接口提供方来调用该函数，就称为回到函数。

例子：

```
#include <stdio.h>
#include <signal.h>


```

```
5
6  void func(int num )
7  {
8      printf("当前收到  3 号信号 ,  军师让我蹲下 !!\n");
9  }
10
11
12  int test( int num  ,  void (*p)(int)  )
13  {
14
15      for (size_t i = 0; i < num ; i++)
16      {
17          printf("num:%d\n" , num-- );
18          if (num == 50 )
19          {
20              p(1);
21          }
22
23
24      }
25
26      return 0 ;
27  }
28
29  int main(int argc, char const *argv[])
30  {
31      void (*p)(int); // 定义一个函数指针
32      p = func ; // 让指针p 指向函数  func
33
34
35      test( 100 , p  );
36
37
38      return 0;
39  }
40
```

```
1  #include <stdio.h>
2  #include <signal.h>
```

```
3
4
5
6  void func(int num )
7  {
8      printf("当前收到  3 号信号 ,  军师让我蹲下 !!\n");
9  }
10
11
12 int main(int argc, char const *argv[])
13 {
14     void (*p)(int); // 定义一个函数指针
15     p = func ; // 让指针p 指向函数  func
16
17     // 设置进程捕获信号 , 如果信号值 为 3的时候 ,  会自行调用 p 所指向的函
数 （代码/指令）
18     signal( 3 , p );
19
20     while(1);
21
22
23     return 0;
24 }
25
```

总结：

- signal 函数是一个用于捕获信号的函数， 当他捕获到指定信号的时候则会执行用户所提供的函数。
- 由于signal函数是内核提供的函数，修改内核的代码不现实， 因此它提供的接口是一个函数指针，只要某个条件满足则会自动执行我们所给的函数。
- 使得不同软件模块的开发者的工作进度可以独立出来，不受时空的限制，需要的时候通过约定好的接口（或者标准）相互契合在一起

内联函数：

不适用内联函数的情况下，有可能某一个函数被多次重复调用则会浪费一定的时间在调用的过程中（现场保护+恢复）

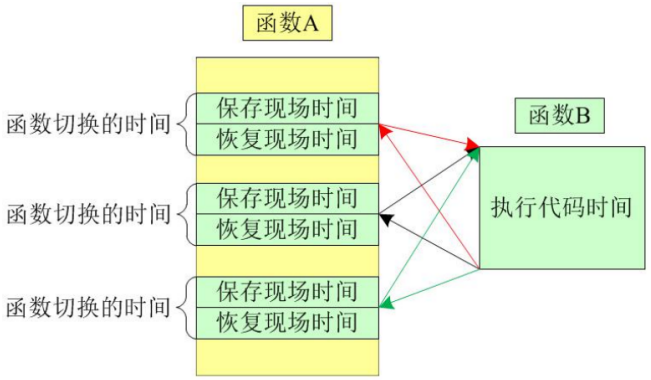


图 2-31 函数切换时高昂的时间成本

如果使用内联函数就相当与把需要调用的函数的内容（指令）拷贝到需要调用的位置，可以节省函数调用的过程中浪费的时间

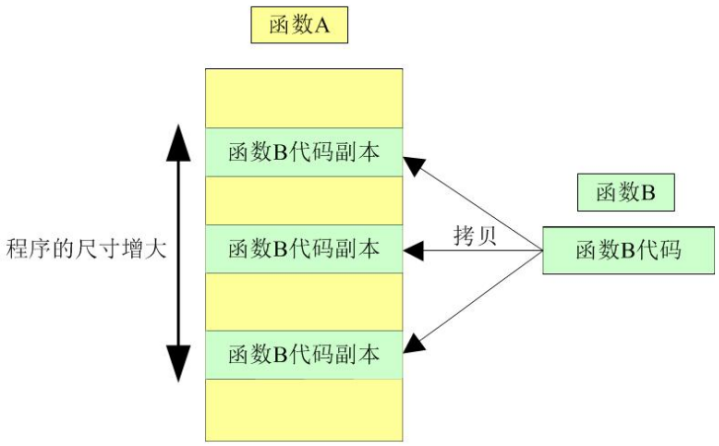


图 2-32 多处拷贝消除函数的切换时间

语法：

与普通函数区别不大， 只是在前面增加了函数的修饰 inline

```
inline int max_value(int x, int y);
```

示例：

```

list.h:

static inline void INIT_LIST_HEAD(struct list_head *list)
{
    list->next = list;
    list->prev = list;
}

main.c:
#include <list.h>
```

```
12
13 int main()
14 {
15  INIT_LIST_HEAD ( *p ) ;// 当前这一行代码在编译完成后会被该函数实际的代码进行替换
16  /*
17   list->next = list;
18      list->prev = list;
19  */
20  return 0 ;
21 }
22
```

注意：

内联函数在提高运行效率的过程中，消耗了更多的内存空间。

变参函数：

指参数的个数以及类型是可以根据实际y应用有所变化的。

分析打印函数如何实现变参：

```
1 printf("%d%c%lf", 100, ‘x’, 3.14);
```

如上代码中， 各个参数入栈的顺序是从右往左进行的。

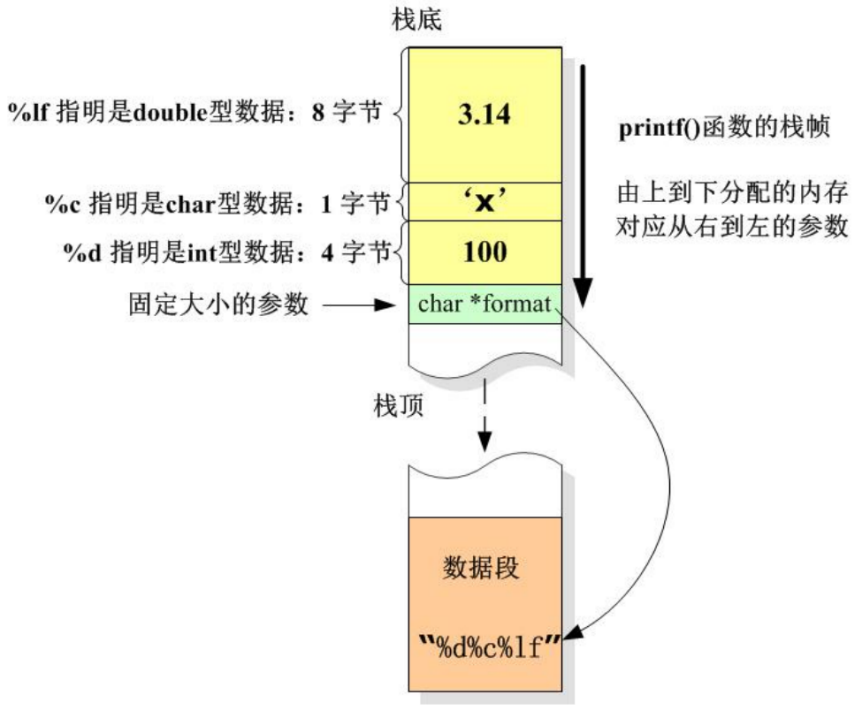


图 2-24 变参函数内幕

作业：

1. 尝试使用递归来方向输出一个字符串：

比如： Hello GZ2123

则输出 3213ZG olleH

2. 阶乘 --》 1*2*3*4*5*6，，， 10

```
int test( int n )
{
    // 如果 n 等于 1则 阶乘 为 1
    if (n == 1 )
    {
        return 1 ;
    }
    else
    {
         return  test( n-1 )*n;
```

```
    }
}
```

3. 幂运算

```
int mi( int a , int n )
{
    if (n == 0)
    {
        return 1 ;
    }
    else if (n == 1)
    {
         return a ;
     }
     else
     {
         return mi(  a , n - 1 ) * a ;
     }
}
```

4. 参透回调函数的概念
5. 理清楚函数指针

预习：

内联函数
变参函数 printf( )（拓展）
字符串处理函数
作用域/存储期 （内存区域的问题）