第 6 章 函数

（视频讲解：2 小时）

一个 C 程序可由一个主函数和若干个其他函数构成。一个较大的程序可分为若干个程序模块，每一个模块用来实现一个特定的功能。在高级语言中用子程序实现模块的功能。子程序由函数来完成。学习本章，你将掌握：

- 函数的声明，定义与调用
- 嵌套调用与递归调用
- 变量及函数的作用域

6.1 函数声明-定义-调用

6.1.1 函数的声明与定义

函数间的调用关系:由主函数调用其他函数，其他函数也可以互相调用。同一个函数可以被一个或多个函数调用任意多次（如图 6.1.1-1）。

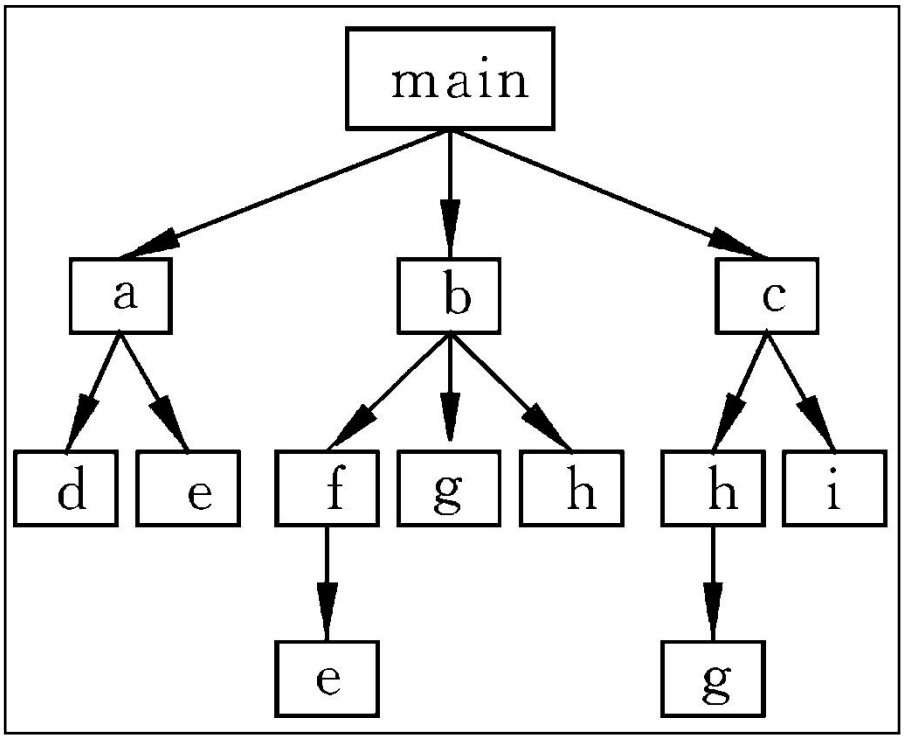


图 6.1.1-1

下面我们来看【例 6.1.1-1】，这次有两个 c 文件，func.c 是子函数 printstar 和 print_message 的**实现**，也可以叫**定义**，main.c 是 main 函数，func.h 中存放的是标准头文件的声明，还有 main 函数中调用的两个子函数的声明，如果不在头文件中对使用的函数进行声明，在编译时将出现警告。

【例 6.1.1-1】函数嵌套调用

func.h

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

```
#include <stdlib.h>

int printstar(int i);//函数声明
void print_message();
```

func.c

```
#include "func.h"

int printstar(int i)//i 即为形式参数
{
    printf("**********\n");
    printf("printstar %d\n",i);
    return i+3;
}
void print_message()//可以调用 printstar
{
    printf("how do you do\n");
    printstar(3);
}
```

main.c

```
#include "func.h"
int main()
{
    int a=10;
    a=printstar(a);
    print_message();
    printstar(a);
    system("pause");
    return 0;
}
```

下面我们来看下 C 语言编译及执行的一些特点：

1、一个 C 程序由一个或多个程序模块组成，每一个程序模块作为一个源程序文件。对于较大的程序，通常将程序内容分别放在若干个源文件中，再由若干源程序文件组成一个 C 程序。这样便于**分别编写、分别编译，提高调试效率**。一个源程序文件可以为多个 C 程序公用。

2、一个源程序文件由一个或多个函数以及其他有关内容（如命令行、数据定义等）组成。一个源程序文件是一个编译单位，在程序编译时是以源程序文件为单位进行编译的，而不是以函数为单位进行编译的。main.c 和 func.c 分别单独编译，在链接成为可执行文件 exe 时，main 中调用的函数 printstar 和 print_message 才会通过链接去找到函数定义的位置。

3、C 程序的执行是从 main 函数开始的，如果在 main 函数中调用其他函数，在调用后流程返回到 main 函数，在 main 函数中结束整个程序的运行。

4、所有函数都是平行的，即在定义函数时是分别进行的，是互相独立的。一个函数并不从属于另一函数，即**函数不能嵌套定义**。函数间可以互相调用，但不能调用 main 函数。main 函数是系统调用的。main 函数 print_message 函数，print_message 函数调用 printstar 函数，我们把这种调用称为嵌套调用。

声明与定义的差异：

1、函数的定义是指对函数功能的确立，包括指定函数名，函数值类型、形参及其类型、函数体等，它是一个完整的、独立的函数单位。
2、函数的声明的作用则是把函数的名字、函数类型以及形参的类型、个数和顺序通知编译系统，以便在调用该函数时编译系统能正确识别函数并检查调用是否合法。

隐式声明：

C 语言中有几种声明，它的类型名可以省略。例如，函数如果不显式地声明返回值的类型,它就默认返回整型。当你使用旧风格声明函数的形式参数时，如果省略了参数的类型，编译器默认它们为整型。依赖隐式声明不是好的写法，隐式声明会让读者的头脑留下疑问：是否有意遗漏类型名？还是不小心忘记了？显示声明能够清楚的表达意图！

6.1.2 函数的分类与调用

从用户角度来看，函数分为两种：

① 标准函数，即库函数。这是由系统提供的，用户不必自己定义这些函数，可以直接使用它们，例如 printf 函数，scanf 函数。不同的 C 系统提供的库函数的数量和功能会有一些不同，但许多基本的函数是共同的。

② 用户自己定义的函数。用以解决用户的专门需要。

从函数的形式看，函数分两类：

①无参函数。无参函数一般用来执行指定的一组操作。在调用无参函数时，主调函数不向被调用函数传递数据。

定义无参函数的一般形式为:

类型标识符　函数名（）
{
声明部分
语句部分
}

在前面的【例 6.1.1-1】例子中 print_message 即为无参函数。

②有参函数。主调函数在调用被调用函数时，通过参数向被调用函数传递数据。

类型标识符　函数名（形式参数表列）
{
声明部分
语句部分
}

在前面的【例 6.1.1-1】例子中 printstar 即为有参函数，int i 对应的 i 为形式参数。主调函数和被调用函数之间有数据传递的关系。在不同的函数之间传递数据，可以使用的方法有：

参数：通过形式参数和实际参数
返回值：用 return 语句返回计算结果
全局变量：外部变量

下面来看一个全局变量的实例【例 6.1.2-1】：

【例 6.1.2-1】全局变量的使用

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

int i=10;//全局变量
```

```
void print(int a)
{
    printf("print i=%d\n",i);
}
int main()
{
    printf("main i=%d\n",i);
    i=5;
    print(i);
    system("pause");
    return 0;
}
```

全局变量存储在哪里呢，如图 5.1.2-1，全局变量 i 在数据段，所以 main 函数和 print 函数都是可见的，全局变量不会因为某个函数执行结束而消失，在整个进程执行过程中始终有效，因此**工作中尽量避免使用全局变量！**之前几个章节，我们在函数内定义的变量都称为局部变量，局部变量存储在自己函数对应的栈空间内，在函数执行结束后，函数内的局部变量所分配的空间将会得到释放。如果局部变量与全局变量重名，那么将就近原则，实际获取和修改的值是局部变量的值。

图 5.1.2-1

思考题：如果把 print(int a)改为 print(int i)，那么 print 函数内的打印将会是多少？

关于形参与实参的一些说明：

1、在定义函数中指定的形参，在未出现函数调用时，它们并不占内存中的存储单元。只有

在发生函数调用时，函数 print 中的形参才被分配内存单元。在调用结束后，形参所占的内存单元也被释放。
2、实参可以是常量、变量或表达式，但要求它们有确定的值，例如 print(i+3)。在调用时将实参的值赋给形参。假如 print 函数有两个形参，如 print(int a,int b),实际调用 print 函数时，使用 print(i,i++)，是不合适的，因为 C 标准没有规定函数调用，是从左到右计算，还是从右到左计算，这样不通的编译各自制定自己的标准，就会造成代码在移植过程中发生非预期错误。
3、在被定义的函数中，必须指定形参的类型。如果实参表列包含多个实参，则各参数间用逗号隔开。实参与形参的个数应相等，类型应匹配。实参与形参按顺序对应，一一传递数据。
4、实参与形参的类型应相同或赋值兼容。
5、值传递:实参向形参的数据传递是单向“值传递”，只能由实参传给形参，而不能由形参传回来给实参。在调用函数时，给形参分配存储单元，并将实参对应的值传递给形参，调用结束后，形参单元被释放，实参单元仍保留并维持原值。
6、形参相当于局部变量，因此不能再定义局部变量与形参同名，否则会造成编译不通

6.2 嵌套调用

在代码实例代码实例 6.1.1-1 中，我们可以看到 print_message 调用 printstar，必须等 printstar 执行结束返回 print_message，再从 print_message 返回到 main 函数，那么是否可以从 printstar 直接到达 main 呢？采用 goto 语句是不行的，前面已经讲过 goto 只能在函数内使用，那么看【例 6.2-1】实例如何实现函数间跳转(考研机试的同学可以不用掌握本节，如果要做最后一个编译器项目应对复试面试，需要掌握本节)。
【例 6.2-1】setjmp 与 longjmp

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

jmp_buf envbuf;
void b()
{
    printf("I am b func\n");
    longjmp(envbuf,5);//回到 setjmp 位置
}
void a()
{
    printf("before b(),I am a func\n");
    b();
    printf("after b(),I am a func\n");
}
void main()
{
    int i;
    i=setjmp(envbuf);//保存进程执行的上下文
    if(i==0)
```

```
    {
        a();
    }
    system("pause");
    return;
}
```

从实例【例 6.2-1】中我们可以看到 main 函数调用 a 函数，a 函数调用 b 函数，正常情况下 printf("after b(),I am a func\n");这一句一定会得到执行，但是本实例执行后你会发现 after b(),I am a func 并没有得到打印，为什么呢，因为 b 中的 longjmp 直接跳转到 setjmp 的位置，这时 i 的值为 5，从而实现了从 b 函数直接跳转到 main 函数的操作，实现原理是什么呢？我们的程序运行起来后，称之为**进程**，启动 windows 的任务管理器你可以看到很多进程，我们的进程执行到 setjmp 位置时，我们通过 setjmp 函数保存了**进程的上下文**（处理器运行进程期间，运行进程的信息被存储在处理器的寄存器和高速缓存中，执行的进程被加载到寄存器的数据集被称为上下文），所以当执行到函数 b 时，我们可以通过 longjmp 实现恢复 envbuf 的现场。

是不是一定要掌握 setjmp 编程方法呢，其实不是，讲解这个编程的目的是为大家以后工作中一旦遇到协程编程，做好充分的理解准备。

针对不使用全局变量 jmp_buf envbuf，通过传参实现的代码，请加入前言 QQ 群获取代码集合即可。

6.3 递归调用

假如我们要求数字 n 的阶乘，当然你可能会说很简单，写个 for 循环就可以实现，但是我们要通过递归实现，为什么呢？其实是因为我们用递归在解决一些问题时，可以让问题变得简单，降低编程的难度，比如接下来的题目，假如有 n 个台阶，一次只能上 1 个台阶，或者 2 个台阶，请问走到第 n 个台阶，有几种走法？可能你不太理解题目，比如假如有 3 个台阶，总计就有 3 中走法，第一种，1,1,1，第二种先上 2 个台阶，再上一个台阶，第三种，先上一个台阶，再上两个台阶，请看实例【例 6.3-1】：

【例 6.3-1】n 的阶乘的递归调用实现

```
#include <stdio.h>
#include <stdlib.h>
//求 n 的阶乘
int f(int n)
{
    if(1==n)
    {
        return 1;
    }
    return n*f(n-1);
}
//走楼梯
int step(int n)
{
    if(1==n)
    {
```

```
        return 1;
    }
    if(2==n)
    {
        return 2;
    }
    return step(n-1)+step(n-2);
}
int main()
{
    int n;
    int ret;
    scanf("%d",&n);//请输入数字的大小
    ret=f(n);
    printf("%d\n",ret);
    scanf("%d",&n);//请输入台阶数
    ret=step(n);
    printf("%d\n",ret);
    system("pause");
    return 0;
}
```

思考题：对于上台阶，如果不用递归，具体应该怎么实现呢？对比非递归比递归有哪些优势呢？

前面的上台阶的递归难度不是很高，很多同学可以找到对应的规律，那么接下来来一个比较难的递归题目（如图 6.3-1），就是经典的汉诺塔问题，碟子都在 A 柱子上，借助 B 柱子，把碟子都放到 C 柱子上，一次只能移动一个碟子，同时要求只能是小碟子垒在大碟子之上。

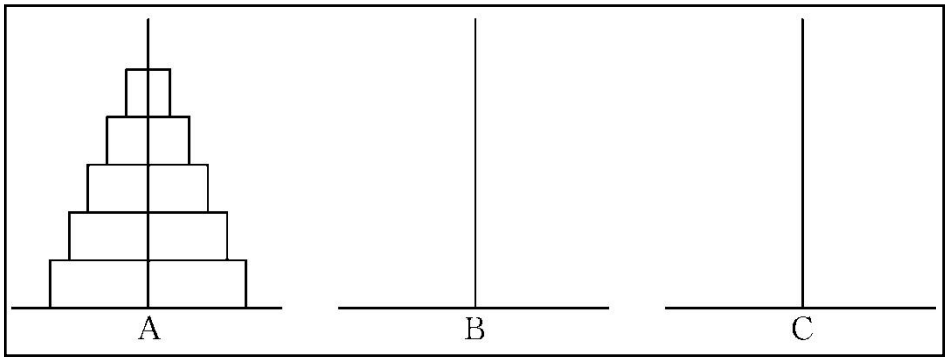


图 6.3-1

图解分析可以得出，我们可以分解为以下三个步骤：

1、将 A 上 n － 1 个盘借助 C 座先移到 B 座上。

2、把 A 座上剩下的一个盘移到 C 座上。

3、将 n － 1 个盘从 B 座借助于 A 座移到 C 座上。

根据描述请看答案【例 6.3-2】

【例 6.3-2】汉诺塔游戏递归实现

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

```
/*  定义 hanoi 函数,将 n 个盘从 one 座借助 two 座，移到 three 座 */
void hanoi(int n,char one,char two,char three)
{
    void move(char x,char y);      /* 对 move 函数的声明 */
    if(n==1) move(one,three);
    else
    {
        hanoi(n-1,one,three,two);
        move(one,three);
        hanoi(n-1,two,one,three);
    }
}
void move(char x,char y)            /* 定义 move 函数 */
{
    printf("%c-->%c\n",x,y);
}

void main()
{
    void hanoi(int n,char one,char two,char three);
    /* 对 hanoi 函数的声明 */
    int m;
    printf("input the number of diskes:");
    scanf("%d",&m);
    printf("The step to moveing %d diskes:\n",m);
    hanoi(m,'A','B','C');
    system("pause");
}
```

思考题：如何去理解答案，并让自己轻松写出呢？

6.4 变量及函数的作用域

6.4.1 局部变量与全局变量

内部变量：在一个函数内部定义的变量称内部变量。它只在本函数范围内有效，即：只有在本函数内才能使用这些变量，故称为“局部变量”

(1) 主函数中定义的变量只在主函数中有效,而不因为在主函数中定义而在整个文件或程序中有效。主函数也不能使用其他函数中定义的变量。

(2) 不同函数中可以使用相同名字的变量,它们代表不同的对象,互不干扰。

(3) 形式参数也是局部变量。

(4) 在一个函数内部,可以**在复合语句中定义变量,这些变量只在本复合语句中有效**,这种复合语句也称为“分程序”或“程序块”。代码实例 6.4.1-1 中的 int j=5 就是如此，只在离自己最近的大括号有效，离开大括号后在其下面使用会造成编译不通。

外部变量：函数之外定义的变量称为外部变量。外部变量可以为本文件中其他函数所共用。它的有效范围为从定义变量的位置开始到本源文件结束。所以也称全程变量。

(1) 全局变量在程序的全部执行过程中都占用存储单元，而不是仅在需要时才开辟单元。

(2) 使用全局变量过多，会降低程序的清晰性。在各个函数执行时都可能改变外部变量的值，程序容易出错。因此，要限制使用全局变量。

(3) 降低函数的通用性。因为函数在执行时要依赖于其所在的外部变量。如果将一个函数移到另一个文件中，还要将有关的外部变量及其值一起移过去。但若该外部变量与其他文件的变量同名时，就会出现问题，降低了程序的可靠性和通用性。一般要求把C程序中的函数做成一个封闭体，除了可以通过“实参——形参”的渠道与外界发生联系外，没有其他渠道。

请看下面实例【例 6.4.1-1】，该实例有 main.c，func.c，func.h 组成：

【例 6.4.1-1】局部变量与全局变量

main.c

```
#include "func.h"

extern int k;
void print1()
{
    printf("print1 k=%d\n",k);
}
int k=10;//static 修饰全局变量，该变量不能被其他文件借用
int main()
{
    int i=10;
    {
        int j=5;
    }//局部变量的有效范围是离自己最近的大括号
    printf("i=%d,k=%d\n",i,k);
    print();
    print();
    system("pause");
    return 0;
}
```

func.c

```
#include "func.h"

extern int k;//借用 main.c 文件中的全局变量 k
//static 修饰函数，函数只能在本函数本文件内使用
void print()
{
    static int t=0;//只初始化一次
    t++;
    printf("print execute %d\n",t);
    printf("print k=%d\n",k);
}
```

func.h

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

void print();
```

6.4.2 动态存储方式与静态存储方式

从变量的作用域（即从空间）角度来分，可以分为全局变量和局部变量。从变量值存在的时间角度来分，又可以分为静态存储方式和动态存储方式。**静态存储方式**：指在程序运行期间由系统分配固定的存储空间的方式。**动态存储方式**：则是在程序运行期间根据需要进行动态的分配存储空间的方式。

变量和函数有两个属性：数据类型和数据的存储类别。存储类别指的是数据在内存中存储的方式。

存储方式分为两大类：静态存储类和动态存储类。包含：

自动的（auto）；

静态的（static）；

寄存器的（register）；

外部的（extern）。

根据变量的存储类别，可以知道变量的作用域和生存期。

自动变量 auto：不专门声明为 static 存储类别的局部变量都是动态分配存储空间，在调用该函数时系统会给它们分配存储空间，在函数调用结束时就自动释放这些存储空间。因此这类局部变量称为自动变量。

函数中的形参和在函数中定义的变量(包括在复合语句中定义的变量)，都属此类。

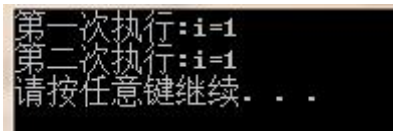
【例 6.4.2-1】自动变量

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

void add()
{
    auto int i=0;
    i=i+1;
    printf("i=%d\n",i);
}

int main()
{
    printf("第一次执行:");
    add();
    printf("第二次执行:");
    add();
    system("pause");
    return 0;
}
```

【例 6.4.2-1】代码执行结果如下：



【例 6.4.2-1】中，对于 add 函数，为什么第二次调用 i 依然等于 1，因为函数每次调用结束后，其内部使用的局部变量空间就会被释放，下次再次调用 add 函数时，i 被重新分配空间，因此再次初始化为零，然后加 1 后，得到的值仍为 1。

例如代码实例 6.4.1-1 中的 int i=10，可以修改为 auto int i=10，缺省为 auto，也就是 auto 可以省略，所以很多同学在编程中没有看到 auto 关键字。

1、对**静态局部变量**的说明：

（1） 静态局部变量属于静态存储类别，在静态存储区内分配存储单元。在程序整个运行期间都不释放。而自动变量（即动态局部变量）属于动态存储类别，占动态存储区空间而不占静态存储区空间，函数调用结束后即释放。

（2）对静态局部变量是在编译时赋初值的，即只赋初值一次，在程序运行时它已有初值。以后**每次调用函数时不再重新赋初值而只是保留上次函数调用结束时的值**。--例如代码实例 6.4.1-1 的 `static int t=0`，利用它我们可以统计 print 函数被调用的次数

（3）如在定义局部变量时不赋初值的话，则对**静态局部变量来说，编译时自动赋初值 0（对数值型变量）或空字符（对字符变量）**。而对自动变量来说，如果不赋初值则它的值是一个不确定的值。

（4）虽然静态局部变量在函数调用结束后仍然存在，但其他函数不能引用它。

2、**用 static 修饰全局变量，那么该全局变量将不能被其他文件引用**，例如在 main.c 的 int k 前加入 static 变为 static int k;那么 func.c 使用 extern int k 将无法编译通过。

3、**用 static 修饰函数，那么该函数将不能被其他文件引用。**

register 变量

如图 6.4.2-1 变量的值是存放在内存中的。当程序中用到哪一个变量的值时，由控制器发出指令将内存中该变量的值送到运算器中。 经过运算器进行运算，如果需要存数，再从运算器将数据送到内存存放。

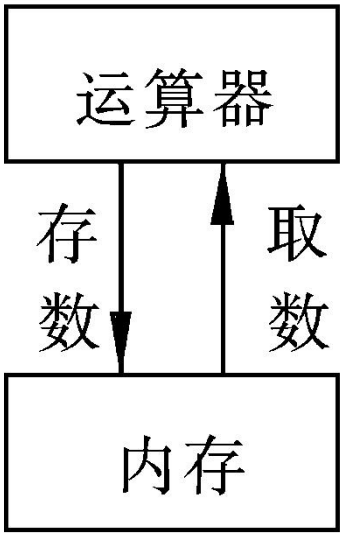


图 6.4.2-1

如果有一些变量使用频繁，则为存取变量的值要花费不少时间。为提高执行效率，C 语

言允许将局部变量的值放在 CPU 中的寄存器中，**需要用时直接从寄存器取出参加运算，不必再到内存中去存取。**由于对寄存器的存取速度远高于对内存的存取速度，因此这样做可以提高执行效率。这种变量叫做寄存器变量，用关键字 register 作声明。由于寄存器的数目是有限的，用 C 语言进行服务器编程时，性能提升采用 register 手法的非常少，嵌入式编程可能会使用到。

用 extern 声明外部变量

外部变量是在函数的外部定义的全局变量，它的作用域是从变量的定义处开始，到本程序文件的末尾。在此作用域内，全局变量可以为程序中各个函数所引用。编译时将外部变量分配在静态存储区。用 extern 来声明外部变量，以扩展外部变量的作用域。

如代码实例 6.4.1-1，在 func.c 中我们通过 extern int k 借用 main.c 的全局变量 k，从而可以获取值，进行打印，当然也可以修改。

所有函数默认都是 extern 类型的，因此任何一个文件的函数都可以被其他文件调用。思考题：代码实例 6.4.1-1 中，如果 print 函数不想让其他文件引用，如果修改？

6.5 函数调用原理详解

每一个函数使用的栈空间，我们可以称为函数帧，也称为栈帧，每一个函数使用的栈空间，我们可以称为一帧，分析一下函数在相互调用过程中栈帧的变化，还是想尽量以比较清晰的思路把这一过程描述出来（虽然前面指针章节初步解析了函数调用），对于 c 函数调用原理的理解是非常重要的，这样对于函数级代码级的优化将会比较清晰。(本节有一定难度，针对考研准备复试的同学，时间紧迫可以跳过，读研期间再研究本节内容)。

6.5.1 关于栈

首先必须明确一点也是非常重要的一点，栈是向下生长的，所谓向下生长是指从内存高地址->低地址的路径延伸，那么就很明显了，栈有栈底和栈顶，那么栈顶的地址要比栈底低。对 x86 体系的 CPU 而言，其中

寄存器 ebp（base pointer ）可称为“帧指针”或“基址指针”，其实语意是相同的。

寄存器 esp（stack pointer）可称为“ 栈指针”。

要知道的是：

---> ebp 在未受改变之前始终指向栈帧的开始，也就是栈底，所以 ebp 的用途是在堆栈中寻址用的。（寻址的作用会在下面详细介绍）

---> esp 是会随着数据的入栈和出栈移动的，也就是说，esp 始终指向栈顶。

见图 6.5.1-1，假设函数 A 调用函数 B，我们称 A 函数为"调用者",B 函数为“被调用者”则函数调用过程可以这么描述：

（1）先将调用者（A）的堆栈的基址（ebp）入栈，以保存之前任务的信息。

（2）然后将调用者（A）的栈顶指针（esp）的值赋给 ebp，作为新的基址（即被调用者 B 的栈底）。

（3）然后在这个基址（被调用者 B 的栈底）上开辟（一般用 sub 指令）相应的空间用作被调用者 B 的栈空间。

（4）函数 B 返回后，从当前栈帧的 ebp 即恢复为调用者 A 的栈顶（esp），使栈顶恢复函数 B 被调用前的位置；然后调用者 A 再从恢复后的栈顶可弹出之前的 ebp 值（**可以这么做是因为这个值在函数调用前一步被压入堆栈**）。这样，ebp 和 esp 就都恢复了调用函数 B 前的位置，也就是栈恢复函数 B 调用前的状态。

相当于：

```
mov     %ebp , %esp   //把 ebp 内的内容复制到 esp 寄存器中
pop     %ebp          //弹出栈顶元素，放置到 ebp 寄存器中
```

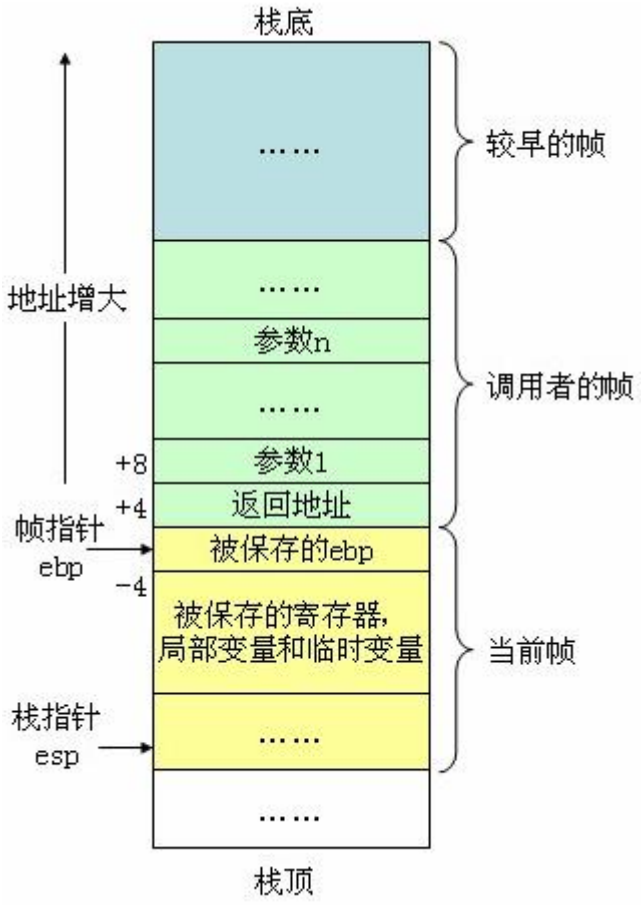


图 6.5.1-1

6.5.2 代码实例分析

有如下代码：

【例 6.5.2-1】函数调用过程代码

```
void swap(int *a,int *b)
{
   int c;
   c = *a;
```

```
    *a = *b;
    *b = c;
}

int main(void)
{
    int a,b,ret;
    a =16;
    b = 64;
    ret = 0;
    swap(&a,&b);
    ret = a - b;
    return ret;
}
```

【例 6.5.2-1】代码并不复杂，是 main 函数调用 swap 函数，实现整型变量 a 和 b 交换，图 6.5.2-1 是栈指针的实际变化过程。

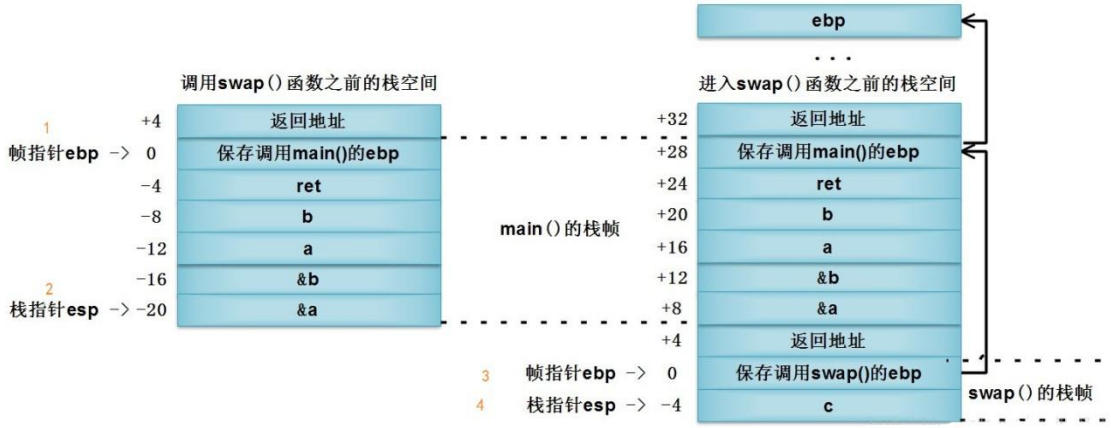


图 6.5.2-1 函数调用过程栈变化

首先需要知道可执行程序并不是自动到内存中的，而是其他程序调用的，因此我们的 main 函数虽然是入口函数，但是依然由其他函数调用它，所以图 6.5.2-1 左边部分是未调用 swap 函数时，栈空间的情况，这时帧指针 ebp 保存的是调用 main 的 ebp，不少同学不理解这时怎么回事，不用担心，原理与 main 调用 swap 的原理相同，接着当 main 函数调用 swap 函数时，我们首先 push %ebp，就是将 main 函数的帧指针 ebp（图 6.5.2-1 中标 1 的位置）进行压栈，这时将 main 函数的栈指针 esp（图 6.5.2-1 中标 2 的位置）作为新函数 swap 的帧指针 ebp（图 6.5.2-1 中标 3 的位置），也就是 main 函数的栈顶指针，作为新函数 swap 的基指针（也叫帧指针），新函数 swap 栈顶指针 esp 会随着定义变量数目依次增加（图 6.5.2-1 中 4 的位置）。

思考题：学习完函数调用原理后，函数调用的开销在哪里，直接在 main 函数内实现变量交换与使用 swap 进行交换有什么区别？