西北工业大学
NORTHWESTERN POLYTECHNICAL UNIVERSITY

C程序设计

Programming in C

1011014

主讲：姜学锋，计算机学院

调用函数 - 返回与参数传递

- 3、函数参数
- 4、如何设计参数

4.2 函数参数

- 本质上，函数参数是为了让主调函数与被调函数能够进行数据交换，如主调函数向被调函数传递一些数据，被调函数向主调函数返回一些数据。

4.2 函数参数

- 函数参数是实现函数时的重要内容，是函数接口的首要任务，围绕这个目标需要研究：
- ①形式参数的定义与实际参数的提供的对应关系，包括参数的类型、次序和数目。
- ②函数参数的数据传递机制，包括主调函数与被调函数的双向数据传递。

4.2.1 形式参数

- 函数定义中的形式参数列表（parameters），简称形参。例如：

```
1 int max(int a, int b)
2 {
3     return a>b ? a : b;
4 }
```

- 第1行a和b就是形参。

4.2.1 形式参数

- 函数定义时指定的形参，在未进行函数调用前，并不实际占用内存中的存储单元，这也是称它为形式参数的原因，即它们不是实际存在的。
- 只有在发生函数调用时，形参才分配实际的内存单元，接受从主调函数传来的数据，此刻形参是真实存在的，因而可以对它们进行各种操作。
- 当函数调用结束后，形参占用的内存单元被自动释放。此后，形参又是未实际存在的。

4.2.1 形式参数

- 形参的类型可以是任意数据类型，换言之，函数允许任意类型的数据传递到函数中。

- 但函数传递不同类型数据的机制不同，所以形参类型的设计一是依据实际需求，二是确保最佳的数据传递。

4.2.2 实际参数

- 函数调用时提供给被调函数的参数称为实际参数（arguments），简称实参。
- 实参必须有确定的值，因为调用函数会将它们传递给形参。实参可以是常量、变量或表达式，还可以是函数的返回值。

例如：

```
x = max(a,b); //max函数调用，实参为a,b
y = max(a+3,128); //max函数调用，实参为a+3,128
z = max(max(a,b),c); //max函数调用，实参为max(a,b),c
```

4.2.2 实际参数

- 实参是以形参为依据的，即实参的类型、次序和数目要与形参一致。如果参数数目不一致，则出现编译错误；如果参数次序不一致，则传递到被调函数中的数据就不合逻辑，难有正确的程序结果；
- 如果参数类型不一致时，则函数调用时按形参类型隐式类型转换实参；如果是不能进行隐式类型转换的类型，就会出现编译错误。

4.2.2 实际参数

- 更重要的是，实参的数据应与函数接口要求的数据物理意义是一致的，否则即使语法正确，程序的运行结果也是错的。例如调用数学库函数中的sin函数求正弦时，函数接口就要求实参必须是弧度的数据。

- 因此，实参数据传递给形参，必须满足语法和应用两方面的要求。

4.2.3 参数传递机制

- 程序通常有两种函数参数传递机制：
- 值传递和引用传递。

- 值传递（pass-by-value）过程中，形参作为被调函数的内部变量来处理，即开辟内存空间以存放由主调函数复制过来的实参的值，从而成为实参的一个副本。

- 值传递的特点是被调函数对形参的任何操作都是对内部变量进行，不会影响到主调函数的实参变量的值。

4.2.3 参数传递机制

- 例如：

```
void fun(int x, int y, int m) //x,y,m调用时是a,b,k的一个副本
{
    m = x>y ? x : y; //仅修改函数内部的m
}
void caller() //主调函数，调用者
{
    int a=10, b=5, k=1;
    fun(a,b,k); //实参值传递
}
```

- 在fun函数中对形参m的赋值不修改caller函数中的实参k。

4.2.3 参数传递机制

- 引用传递(pass-by-reference)过程中，被调函数的形参虽然也作为内部变量开辟了内存空间，但是这时存放的是由主调函数复制过来的实参的内存地址，从而使得形参为实参的一个别名（形参和实参内存地址相同，则它们实为同一个对象的两个名称）。被调函数对形参的任何操作实际上都是对主调函数的实参进行操作。

- C语言中，值传递是唯一的参数传递方式。C语言的后续C++，支持引用传递。

4.2.3 参数传递机制

- 值传递时，实参数据传递给形参是单向传递，即只能由实参传递给形参，而不能由形参传回给实参，这也是实参可以是常量和表达式的原因（这些数据不是左值）。

4.2.3 参数传递机制

- 值传递存在以下的局限性：
- （1）值传递做不到在被调函数中修改实参。
- （2）对于基本类型，例如整型、字符型，由于数据量不大，传递的时间和空间开销不是问题；但如果要传递的是大型数据对象时，会对函数调用效率产生影响。
- （3）当没有办法实现实参复制到形参时，不能值传递。
- 此时，有效的解决办法是使用指针。

4.2.4 函数调用栈

- 有必要了解，在函数调用过程中系统做了些什么，对这个问题的透彻理解有助于编写正确的函数，而且加深对函数调用与返回、参数传递机制、嵌套调用和递归调用的认识。

4.2.4 函数调用栈

- 函数调用时，为了能将参数传递到函数中、准确返回到调用点以及返回函数值，使用了“栈”来管理存储器。

- 栈是内存管理中的一种数据结构，是一种先进后出的数据表，即先进去的数据后出来。栈最常见操作有两种：进栈（push）和出栈（pop）。

4.2.4 函数调用栈

- 系统为每次函数调用在“栈”中建立独立的栈框架，称为函数调用栈帧（stack frame），其建立和撤销是自动维护的。

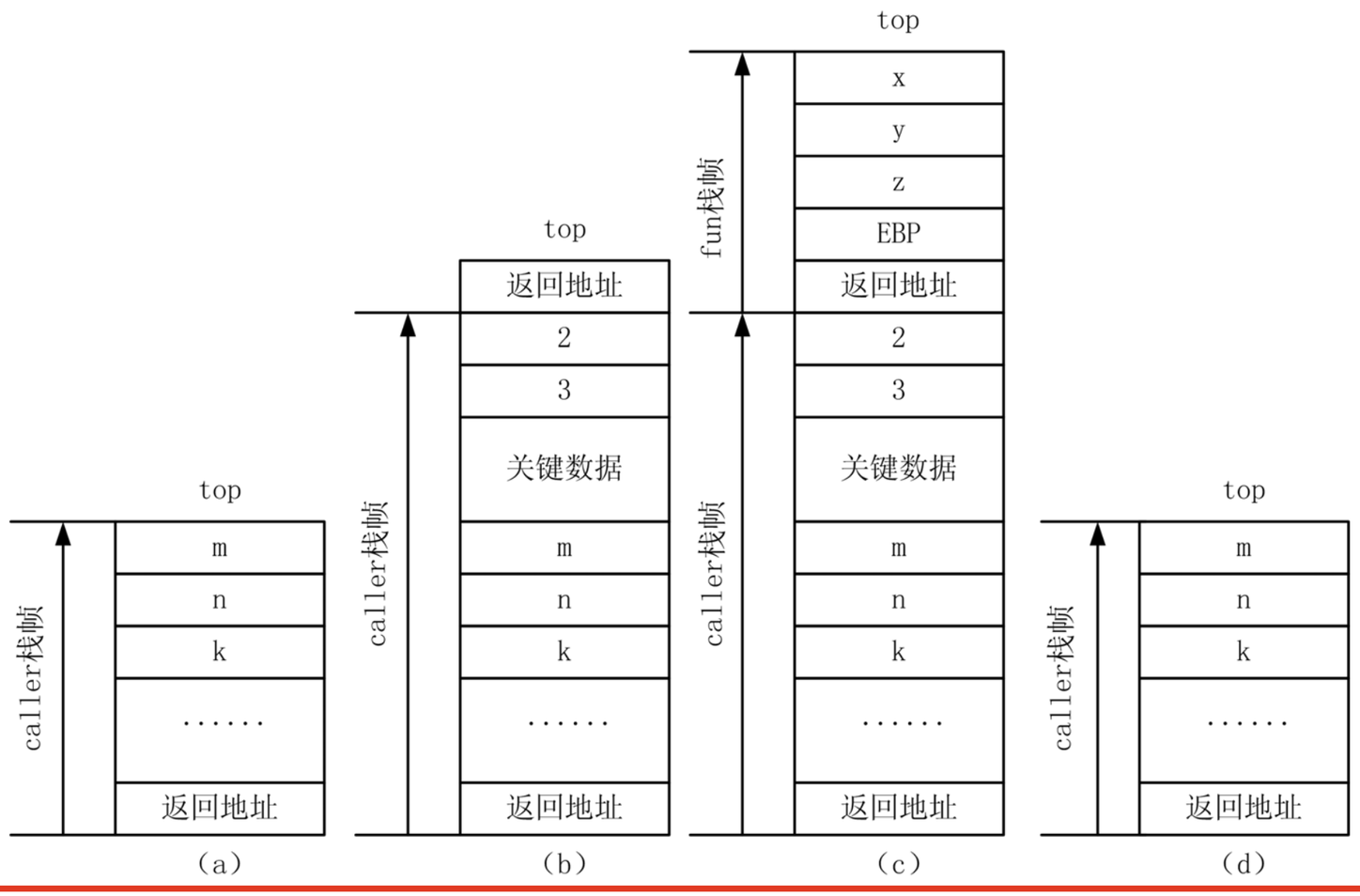
4.2.4 函数调用栈

- 下面结合具体的调用例子，来说明函数调用栈帧的工作原理。
- 假设有主调函数和被调函数如下：

```
int fun(int a, int b) //被调函数
{
    int x = 8, y=2, z;
    z = (a+b)*x +(a-b)*y;
    return z;
}
void caller() //主调函数，调用者
{
    int m=2 , n=3 , k;
    k = fun(m,n); //函数调用
}
```

4.2.4 函数调用栈

图4.1 函数调用栈



4.2.4 函数调用栈

- （1）当在caller函数中运行时，系统使用caller函数栈帧，如图（a）所示。

- 调用函数fun前，caller函数首先保护现场，将关键数据进栈，再将传递给fun的实参一一进栈。按调用约定，最右边的实参最先进栈，然后调用fun并将返回地址进栈，如图（b）所示。返回地址是caller函数中fun调用点的下一条指令位置，当fun以这个地址返回时，正好回到caller函数的下一条指令上。

4.2.4 函数调用栈

- （2）进入fun函数时，fun首先建立它自己的栈帧，保存caller函数栈帧记录值EBP，设置自己的EBP，然后在栈中为局部变量分配空间（只要在栈帧中移动栈顶top就留出空间给局部变量，称为分配），如果变量有初始化，fun还会——给它们赋初值，如图（c）所示。

4.2.4 函数调用栈

- （3）fun函数体开始执行了，这其中也许还有进栈、出栈的动作，也许还会调用别的函数，甚至递归地调用fun本身，但fun通过自己的EBP加上下偏移总是可以找到函数形参和局部变量的。

4.2.4 函数调用栈

- （4）当fun函数执行完后，fun首先释放局部变量空间（在栈帧中将栈顶top向栈底移动收回空间，称为释放），然后恢复caller函数EBP，回到fun栈底，取出返回地址返回。回到caller函数中，caller函数获得fun函数返回值，并且按调用约定将原先入栈的参数一一出栈，恢复现场，使栈回到原先的状态，达到栈平衡。如图（d）所示。

4.2.4 函数调用栈

- 从中可以看出：
- （1）实参是通过进栈传递到函数内部的，进栈时需要数据值，所以称为值传递。如图（b），分别将n、m的值3、2进栈成为函数fun的形参b和a。

- （2）因为进栈的内存单元长度是由数据类型决定的，所以实参与形参类型必须一致，否则会导致“栈溢出”错误，即超出实际栈空间长度。

4.2.4 函数调用栈

- （3）函数调用约定（calling convention）不仅决定了发生函数调用时函数参数的进栈顺序，还决定了是由主调函数还是被调函数负责清除栈中的参数。实际上，函数调用约定的方式有多种，C语言默认使用C调用约定，实参从右向左依次进栈。换言之，函数调用时实参的运算方向是自右向左。

- （4）函数内非静态局部变量是进入函数时才分配空间的，函数结束时自动释放。形参的情况与此相似。

4.2.5 const参数

- 函数定义时，允许在形参的类型前面加上const限定，语法形式为：

```
返回类型 函数名(const 类型 形式参数,.....)
{
    函数体
}
```

4.2.5 const参数

- const用来限制对一个对象的修改操作，即对象不允许被改变。出现在函数参数中的const 表示在函数体中不能对这个参数做修改。例如：

```
int strcmp(const char *str1 , const char *str2)
{
    ... //函数体
}
```

- 在strcmp函数中不应该有改变这两个参数的操作，否则编译出错。

4.2.5 const参数

- 函数参数使用const限定的目的是确保形参对应的实参对象在函数体中不会被修改。

- 通常，基本类型的参数，因为形参和实参本来就不是同一个内存单元，即使修改形参也不会影响到实参，因此没有必要const限定。

- 但如果是数组参数、指针参数就有必要了。

4.2.6 可变参数函数

- 仔细研究printf和scanf函数，会发现这两个函数的参数不像函数定义的形参列表，因为它们的参数可以有很多个，而且数目可变。
- C语言支持可变参数的函数，允许函数参数数目是不确定的。下面给出可变参数函数的定义方法和举例。

4.2.6 可变参数函数

- 可变参数函数的定义形式为：

```
返回类型 函数名(类型1 参数名1, 类型2 参数名2, ......)
{
   函数体
}
```

4.2.6 可变参数函数

- 形参可以分为两部分：个数确定的固定参数和个数可变的可选参数。
- 一般来说，至少需要第一个参数是普通的形参，后面用三个点“...”表示可变参数，且只能位于函数形参列表的最后。这里的三个点不是省略的意思，而是可变参数要求的写法。例如：

```
int fun(int a,...)
```

4.2.6 可变参数函数

- 如果没有任何一个普通的形参，定义成这样：

```
int fun(...)
```

- 那么在函数体中就无法使用任何参数了，因为无法通过宏来提取每个参数。所以除非函数体中的确没有用到参数表中的任何参数，否则在参数表中使用至少一个普通的形参。

4.2.6 可变参数函数

- 在函数体中可以使用stdarg.h头文件定义的几个va_*的宏来引用可变参数：

- （1）va_list arg_ptr：定义一个指向个数可变的参数列表指针；

4.2.6 可变参数函数

- （2）va_start(arg_ptr, argN)：使参数列表指针arg_ptr指向函数参数列表中第一个可选参数，argN是位于第一个可选参数之前的固定参数，即最后一个固定参数。例如有一个函数是int fun(char a,char b,char c,...)，则它的固定参数依次是a、b、c，最后一个固定参数argN即为c，因此就是va_start(arg_ptr ,c)。

4.2.6 可变参数函数

- （3）va_arg(arg_ptr, type)：返回参数列表中指针arg_ptr所指的参数，返回类型由type指定，并使指针arg_ptr指向参数列表中下一个参数。

4.2.6 可变参数函数

- （4）va_end(arg_ptr)：清空参数列表，并置参数指针arg_ptr无效。指针arg_ptr被置无效后，可以通过调用va_start恢复arg_ptr。每次调用va_start后，必须有相应的va_end与之匹配。参数指针可以在参数列表中随意地来回移动，但必须在va_start～va_end之间。

4.2.6 可变参数函数

【例4.2】

编写并调用计算若干整数平均值的函数。

4.2.6 可变参数函数

例4.2

```
1  #include <stdio.h>
2  #include <stdarg.h> //可变参数函数需要用到va_*的宏定义
3  double avg(int first, ...) //返回若干个整数平均值的函数
4  {
5    int count=0 ,sum=0, i;
6    va_list arg_ptr; //定义变参数列表指针
7    va_start(arg_ptr, first); //初始化
8    i=first; //取第1个参数
9    while( i!=-1 ) //调用时最后一个参数必须是-1，作为结束标记
10   {
11     sum += i; //累加多个整数值
12     count++; //计数
13     i = va_arg(arg_ptr, int); //取下一个参数
14   }
15   va_end(arg_ptr); //清空参数列表
```

4.2.6 可变参数函数

例4.2

```
16    return (count>0 ? (double)sum/count : 0); //返回平均值
17 }
18 int main()
19 {
20    printf("%lf\n", avg(1,2,3,-1)); //返回1-3的平均值
21    printf("%lf\n", avg(7,8,9,10,-1)); //返回7-10的平均值
22    printf("%lf\n", avg(-1)); //没有计算返回0
23    return 0;
24 }
```

4.2.6 可变参数函数

例4.2

```
1  #include <stdio.h>
2  #include <stdarg.h> //可变参数函数需要用到va_*的宏定义
3  double avg(int first, ...) //返回若干个整数平均值的函数
4  {
5    int count=0 ,sum=0, i;
6    va_list arg_ptr; //定义变参数列表指针
7    va_start(arg_ptr, first); //初始化
8    i=first; //取第1个参数
9    while( i!=-1 ) //调用时最后一个参数必须是-1，作为结束标记
10   {
11     sum += i; //累加多个整数值
12     count++; //计数
13     i = va_arg(arg_ptr, int); //取下一个参数
14   }
15   va_end(arg_ptr); //清空参数列表
```

4.2.6 可变参数函数

例4.2

```
1  #include <stdio.h>
2  #include <stdarg.h> //可变参数函数需要用到va_*的宏定义
3  double avg(int first, ...) //返回若干个整数平均值的函数
4  {
5    int count=0 ,sum=0, i;
6    va_list arg_ptr; //定义变参数列表指针
7    va_start(arg_ptr, first); //初始化
8    i=first; //取第1个参数
9    while( i!=-1 ) //调用时最后一个参数必须是-1，作为结束标记
10   {
11     sum += i; //累加多个整数值
12     count++; //计数
13     i = va_arg(arg_ptr, int); //取下一个参数
14   }
15   va_end(arg_ptr); //清空参数列表
```

4.2.6 可变参数函数

例4.2

```
1  #include <stdio.h>
2  #include <stdarg.h> //可变参数函数需要用到va_*的宏定义
3  double avg(int first, ...) //返回若干个整数平均值的函数
4  {
5    int count=0 ,sum=0, i;
6    va_list arg_ptr; //定义变参数列表指针
7    va_start(arg_ptr, first); //初始化
8    i=first; //取第1个参数
9    while( i!=-1 ) //调用时最后一个参数必须是-1，作为结束标记
10   {
11     sum += i; //累加多个整数值
12     count++; //计数
13     i = va_arg(arg_ptr, int); //取下一个参数
14   }
15   va_end(arg_ptr); //清空参数列表
```

4.2.6 可变参数函数

例4.2

```
1  #include <stdio.h>
2  #include <stdarg.h> //可变参数函数需要用到va_*的宏定义
3  double avg(int first, ...) //返回若干个整数平均值的函数
4  {
5    int count=0 ,sum=0, i;
6    va_list arg_ptr; //定义变参数列表指针
7    va_start(arg_ptr, first); //初始化
8    i=first; //取第1个参数
9    while( i!=-1 ) //调用时最后一个参数必须是-1，作为结束标记
10   {
11     sum += i; //累加多个整数值
12     count++; //计数
13     i = va_arg(arg_ptr, int); //取下一个参数
14   }
15   va_end(arg_ptr); //清空参数列表
```

4.2.6 可变参数函数

例4.2

```
1  #include <stdio.h>
2  #include <stdarg.h> //可变参数函数需要用到va_*的宏定义
3  double avg(int first, ...) //返回若干个整数平均值的函数
4  {
5    int count=0 ,sum=0, i;
6    va_list arg_ptr; //定义变参数列表指针
7    va_start(arg_ptr, first); //初始化
8    i=first; //取第1个参数
9    while( i!=-1 ) //调用时最后一个参数必须是-1，作为结束标记
10   {
11     sum += i; //累加多个整数值
12     count++; //计数
13     i = va_arg(arg_ptr, int); //取下一个参数
14   }
15   va_end(arg_ptr); //清空参数列表
```

4.2.6 可变参数函数

例4.2

```
1  #include <stdio.h>
2  #include <stdarg.h> //可变参数函数需要用到va_*的宏定义
3  double avg(int first, ...) //返回若干个整数平均值的函数
4  {
5    int count=0 ,sum=0, i;
6    va_list arg_ptr; //定义变参数列表指针
7    va_start(arg_ptr, first); //初始化
8    i=first; //取第1个参数
9    while( i!=-1 ) //调用时最后一个参数必须是-1，作为结束标记
10   {
11     sum += i; //累加多个整数值
12     count++; //计数
13     i = va_arg(arg_ptr, int); //取下一个参数
14   }
15   va_end(arg_ptr); //清空参数列表
```

4.2.6 可变参数函数

例4.2

```
16    return (count>0 ? (double)sum/count : 0); //返回平均值
17 }
18 int main()
19 {
20    printf("%lf\n", avg(1,2,3,-1)); //返回1-3的平均值
21    printf("%lf\n", avg(7,8,9,10,-1)); //返回7-10的平均值
22    printf("%lf\n", avg(-1)); //没有计算返回0
23    return 0;
24 }
```


程序设计