第3章 C语言基础 II

本章学习目标

- 掌握函数、指针的定义和使用。
- 理解编译预处理的作用和用法。
- 掌握结构体、共用体和枚举的定义和使用。
- 掌握零长数组、变长数组和动态数组的定义和使用。
- 掌握 indent 命令的使用。

3.1 函数

C 语言通过函数实现模块化程序设计，用函数实现功能模块的定义。一个 C 程序可以由一个主函数和若干个自定义函数构成，主函数调用其他函数，其他函数也可以互相调用，通过函数之间的调用实现程序功能。

3.1.1 C 语言库函数

C 语言提供了功能丰富的库函数，一般库函数的定义放在头(库)文件中。头文件是扩展名为.h 的文件。使用库函数时在程序开头用“#include <*.h>”或“#include "*.h"”将头文件包含进来。C 语言常用头文件及主要函数如表 3.1 所示。

3.1.2 函数定义和声明

C 程序的基本单位是函数，函数是一段封装好的、可以重复使用的代码，有助于程序的模块化设计。函数有库函数和用户自定义函数，一个实用的 C 程序都应该包含自定义函数，通常一个函数执行一个特定的任务。一个 C 程序至少包含一个主函数(main 函数)，并且主函数只能有一个，主函数的特殊之处在于函数名(main)是固定的，操作系统就认这个

表 3.1 C 语言常用头文件及主要函数

头 文 件	主 要 函 数
stdio.h	包含输入/输出函数的声明，共 6 大类别。①文件操作类：删除文件 remove、修改文件名称 rename、生成临时文件名称 tmpfile、得到临时文件路径 tmpnam、关闭文件 fclose、刷新缓冲区 fflush、打开文件 fopen、将已存在的流指针和新文件连接 freopen、设置磁盘缓冲区 setbuf、设置磁盘缓冲区 setvbuf；②格式化输入与输出类：格式输出 fprintf、格式输入 fscanf、格式输出(控制台)printf、格式输入(控制台)scanf、格式输出到缓冲区 sprintf、从缓冲区中按格式输入 sscanf、格式化输出 vfprintf、格式化输出 vprintf、格式化输出 vsprintf；③字符输入/输出类：输入一个字符 fgetc、字符串输入 fgets、字符输出 fputc、字符串输出 fputs、字符输入(控制台)getc、字符输入(控制台)getchar、字符串输入(控制台)gets、字符输出(控制台)putc、字符输出(控制台)putchar、字符串输出(控制台)puts、字符输出到流的头部 ungetc；④直接输入/输出类：直接流读操作 fread、直接流写操作 fwrite；⑤文件定位类：得到文件位置 fgetpos、文件位置移动 fseek、文件位置设置 fsetpos、得到文件位置 ftell、文件位置复零位 remind；⑥错误处理类：错误清除 clearerr、文件结尾判断 feof、文件错误检测 ferror、得到错误提示字符串 perror
stdlib.h	包含编程所必须要的实用工具函数的声明，共 4 大类别。①字符串转换类：字符串转换为整数 atoi、字符串转换为浮点数 atof、字符串转换为长整数 atol、字符串转换为浮点数 strtod、字符串转换为长整数 strtol、字符串转换为无符号长整型 strtoul、浮点数转换成字符串 ecvt；②产生伪随机序列类：产生随机数 rand、设置随机函数的起动数值 srand；③存储管理类：分配存储空间 malloc、释放存储空间 free、分配存储空间 calloc、重新分配存储空间 realloc；④环境通信类：中止程序 abort、退出程序执行并清除环境变量 atexit、退出程序执行 exit、读取环境参数 getenv、挂起当前进程去临时执行一个其他程序 system、二分查找已排序的数据 bsearch、快速排序 qsort、整数运算求绝对值 abs、得到除法运算底商和余数 div、求长整型的绝对值 labs、求长整型除法的商和余数 ldiv、得到多字节字符的字节数 mblen、得到多字节字符的字节数 mbtowc、多字节字符转换 wctomb、将多字节串转换为整数数组 mbstowcs
ctype.h	包含对单个字符进行处理的函数声明，包括字符类别测试和字母大小写转换：是否为字母和数字 isalnum、是否为字母 isalpha、isascii 判断字符 ASCII 码是否属于[0，127]、是否为控制字符 iscntrl、是否为数字 isdigit、是否为可显示字符(空格除外)isgraph、是否为可显示字符(包括空格)isprint、是否既不是空格又不是字母和数字的可显示字符 ispunct、是否为空格 isspace、是否为大写字母 isupper、是否为十六进制数字(0-9、a-f、A-F)字符 isxdigit、转换为大写字母 toupper、转换为小写字母 tolower
string.h	包含字符串处理函数的声明，包括对字符串进行合并、比较等操作：块复制(目的和源存储区不可重叠)memcpy、块复制(目的和源存储区可重叠)memmove、串复制 strcpy、按长度的串复制 strncpy、串连接 strcat、按长度的串连接 strncat、块比较 memcmp、字符串比较 strcmp、字符串比较(用于非英文字符)strcoll、按长度对字符串比较 strncmp、求字符串长度 strlen、字符串转换 strxfrm、块中字符查找 memchr、串中字符查找 strchr、字符串查找 strstr、字符串分解 strtok、字符串设置 memset、错误字符串映射 strerror
math.h	包含数学函数的声明：反余弦 acos、反正弦 asin、反正切 atan、反正切 2atan2、余弦 cos、正弦 sin、正切 tan、双曲余弦 cosh、双曲正弦 sinh、双曲正切 tanh、指数函数 exp、指数分解函数 frexp、自然对数 log、以 10 为底的对数 lg、浮点数分解函数 modf、幂函数 pow、平方根函数 sqrt

名字，程序执行时 main 函数自动被操作系统调用，除此之外，main 函数和其他自定义

函数没有区别，因此可以将主函数看成一个特殊的自定义函数。各个函数在定义时彼此独立，在执行时可以互相调用，其他函数不能调用主函数。下面主要介绍自定义函数。编程时不仅可以调用C标准库提供的库函数，也可以调用自定义函数。

用户自定义函数的一般形式是："函数类型　函数名(形参列表){函数体}"。一个函数包括函数头和函数体，"函数类型　函数名(形参列表)"为函数头。①函数类型是指函数返回值的数据类型，可以通过函数体中的return语句返回函数值，如果函数没有返回值，则函数类型应写为空类型void，如果省略函数类型，则默认为int型；②函数名是用户自定义的一个标识符，要符合标识符的命名规则；③定义有参函数时，函数名后一对圆括号内的形参列表至少一个形式参数(形参)，多个参数之间用逗号隔开，每个形参都应指定其类型；④定义无参函数时，函数名后圆括号内应该为空或void；⑤函数体是函数的主体部分，是由一对花括号{}括起来的一个语句块，即一个复合语句，可以由若干条语句和声明组成，C89要求所有声明写在所有语句之前，而C99允许语句和声明可以按任意顺序排列，只要每个标识符都遵循先声明后使用的原则即可。

函数(整型或void型函数除外)应该先定义后调用，或者先声明(也称说明)后调用。如果函数定义放在了调用它的函数之后，则一定要在调用它的函数之前对该函数进行声明。函数声明的一般形式是："函数类型　函数名(形参列表);"。函数声明和语句类似，都是以分号结尾，但是语句只能出现在函数体中，而函数声明既可以出现在函数体中，也可以出现在所有函数体之外。

3.1.3　函数调用及参数传递

函数调用就是使用已定义好的函数。函数调用的一般形式为："函数名(实参列表)"。通过调用函数完成已定义的任务。调用无参函数时实参列表为空，但是括号不能省略。当一个函数(主调函数)调用另一个函数(被调函数)时，程序控制权会从主调函数转移到被调函数，被调函数执行完已定义的任务后(当执行返回语句或到达函数体右花括号时)，会把程序控制权再次转移到主调函数。

C语言中有多种函数调用方式，示例如下。

```
1 a = b + max(x, y);              //函数表达式，max函数调用作为表达式中的一项
2 printf("%d", a);                //函数调用语句，printf函数调用作为一条单独的语句
3 printf("%d", max(x, y));        //函数参数，max函数调用作为printf函数调用的实参
```

Linux操作系统通过虚拟内存的方式为所有应用程序(进程)提供了统一的虚拟内存地址，如图3.1所示的用户地址空间部分(内核地址空间部分由所有应用程序共享)，从上到下分别是环境变量、命令行参数、栈、共享库和mmap内存映射区、堆、数据段、代码段和未用地址空间。①代码段存放着程序的机器码和只读数据，可执行指令就是从这里取得的。这个段在内存中一般标记为只读，任何对该区的写操作都会导致段错误。②数据段包括已初始化数据段和未初始化数据段，已初始化数据段用来存放全局的和静态的已初始化变量，未初始化数据段用来存放全局的和静态的未初始化变量。③堆用来存放程序运行时分配的变量。堆的大小并不固定，可动态扩张或缩减。其分配由malloc等函数实现，其释放由free等函数实现，通常一个malloc函数要对应一个free函数。堆内存由程序员负责分配和释放，如果程序员没有释放，那么在程序结束后由操作系统自动回收。④栈用来存放函数调用

时的临时信息，如函数调用所传递的参数、函数的返回地址、函数的局部变量等。栈通常称为先进后出队列。栈的基本操作是PUSH和POP，PUSH操作称为压栈，将数据放置在栈顶；POP操作相反，将栈顶元素移出，称为出栈。⑤命令行参数是指从命令行执行程序时传递给程序的参数。C语言总是从main函数开始执行，它的原型声明为“int main(int argc,char *argv[]);”，参数argc保存程序执行时命令行输入的参数个数。参数argv保存命令行参数。ISO C和POSIX都要求argv[argc]是一个空指针。历史上，大多数UNIX系统都是支持3个参数的main函数，原型声明为“int main(int argc,char *argv[],char *envp[]);”，其中第3个参数envp是环境变量列表。现在POSIX建议不使用第三个参数，而是使用getenv和putenv函数访问环境变量，getenv函数用来获取一个环境变量的内容，putenv函数用来改变或增加环境变量的内容。另外，也可以使用全局变量environ查看整个环境变量。命令行参数和环境变量示例如表3.2所示。

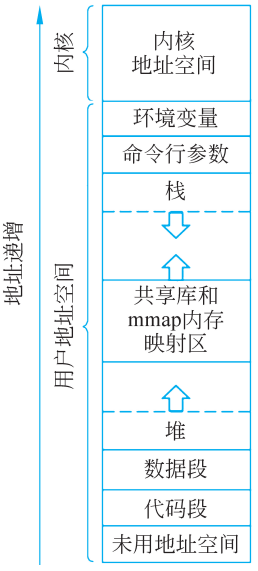

图 3.1 进程地址空间

注意 本章几乎所有源代码都在本书配套资源的“src/第2-3章/code.c”文件中。

扫一扫

表 3.2 命令行参数和环境变量示例

示例代码	运行结果
1 #include<stdio.h> 2 int main(int argc, char *argv[]) { 3 int i; printf("程序名: %s\n", argv[0]); 4 printf("argc=%d\n参数: ", argc); 5 for(i=1; i<argc; i++) 6 printf("%s ", argv[i]); printf("\n"); 7 for(i=0; NULL!=argv[i]; i++) 8 printf("%s ", argv[i]); 9 printf("\nargc=%d\n",i); 10 }	[root@ztg ~]# ./a.out a b c 程序名: ./a.out argc=4 参数: a b c ./a.out a b c argc=4
1 #include<stdio.h> 2 #include<stdlib.h> 3 extern char **environ; 4 int main(int argc,char *argv[],char *envp[]){ 5 char *var=NULL; 6 if((var=getenv("aaa"))) printf("aaa=%s\n",var); 7 putenv("aaa=vv89"); 8 if((var=getenv("aaa"))) printf("aaa=%s\n",var); 9 for(int i=0; NULL!=environ[i]; i++) 10 printf("%s\n", environ[i]); printf("\n"); 11 for(int i=0; NULL!=envp[i]; i++) 12 printf("%s\n", envp[i]); printf("\n"); 13 }	[root@ztg ~]# export aaa=123 [root@ztg ~]# ./a.out aaa=123 aaa=vv89 SHELL=/bin/bash SESSION_MANAGER=local/ztg:@/tmp QT_ACCESSIBILITY=1 COLORTERM=truecolor 省略大部分输出信息

参数是主调函数和被调函数进行信息通信的接口，函数在被调用之前，其形参在内存中是不存在的。函数只有被调用时，其形参才被定义且分配栈中的内存单元。调用函数时，实际参数(实参)与形参之间要进行数据传递，会将实参的值计算出来分别赋值给对应的形参，这一过程称为参数传递。C语言中函数参数有两种传递方式：值传递、地址传递。①值传递是将实参的值复制到形参相应的存储单元中，即形参和实参分别占用不同的存储单元。

值传递的特点是参数值的单向传递，即主调函数调用被调函数时把实参的值传递给形参，之后实参与形参之间不再有任何关系，形参值的任何变化都不会影响到实参的值，调用结束后，形参的存储单元被释放；②地址传递是将实参的地址复制到形参相应的存储单元中，即形参是指针类型的变量，指向实参的存储单元，可以通过形参（指针）读写实参占用的存储单元。地址传递的特点是双向传递，即对形参指向变量的改变也是对实参的改变。值传递和地址传递示例如表 3.3 所示。


扫一扫

表 3.3　值传递和地址传递示例

值传递示例代码	运行结果	地址传递示例代码	运行结果

1 #include<stdio.h>
2 void swap(int a, int b) {
3 int c=a; a=b; b=c;
4 printf("swap:\na=%d, b=%d\n",a,b);
5 }
6 int main() {
7 int a=1, b=2;
8 printf("main:\na=%d, b=%d\n",a,b);
9 swap(a,b); //值传递
10 printf("main:\na=%d, b=%d\n",a,b);
11 }
	main:
	a=1, b=2
	swap:
	a=2, b=1
	main:
	a=1, b=2
		1 #include<stdio.h>
		2 void swap(int *a, int *b) {
		3 int c=*a; *a=*b; *b=c;
		4 printf("swap:\na=%d, b=%d\n",*a,*b);
		5 }
		6 int main() {
		7 int a=1, b=2;
		8 printf("main:\na=%d, b=%d\n",a,b);
		9 swap(&a,&b); //地址传递
		10 printf("main:\na=%d, b=%d\n",a,b);
		11 }
			main:
			a=1, b=2
			swap:
			a=2, b=1
			main:
			a=2, b=1

函数调用是通过栈实现的，栈由高地址向低地址方向生长，栈有栈顶和栈底，入栈或出栈的位置叫栈顶。函数在被调用执行时都会在栈空间中开辟一段连续的空间供该函数使用，这一空间称为该函数的栈帧。栈帧就是一个函数执行的环境。每个函数的每次调用都有它自己独立的一个栈帧。当进行函数调用时，我们经常说先将函数压栈（局部变量从后向前、形参从右向左依次压入栈中），其实是为被调函数分配栈帧，当函数调用结束后，再将函数出栈，其实是释放该函数的栈帧。这一过程是系统帮我们自动完成的，在 x86 系统的 CPU 中，涉及三个寄存器：EIP、ESP、EBP。指令指针寄存器 EIP 中存储的是 CPU 下次要执行的指令的地址。基址指针寄存器 EBP 永远指向当前栈帧（栈中最上面的栈帧）的底部（高地址），栈指针寄存器 ESP 存储着栈顶地址，永远指向当前栈帧的顶部（低地址），因此函数栈帧主要由 EBP 和 ESP 这两个寄存器确定。当程序运行时，ESP 可以移动，元素的进栈或出栈操作通过 ESP 实现，EBP 是不移动的，访问栈帧里的元素可以通过 EBP 加减偏移量实现。表 3.3 中的示例代码的函数栈帧结构如图 3.2 所示。main 函数调用 swap 时，首先在自己的栈帧中压入返回地址（断点），然后分配 swap 的栈帧。在 swap 返回时，swap 的栈帧被弹出，main 函数栈帧作为当前栈帧，此时处于栈顶的返回地址被写入 EIP 中，处理器跳到 main 函数代码区的断点处继续执行。在程序实际运行过程中，main 函数并不是第一个被调用的函数，图 3.2 只是函数调用过程中栈变化的示意图。图 3.2(a)中，即使形参名称与实参名称一样，它们在内存的地址也不一样。图 3.2(b)中，即使形参名称与实参名称一样，它们的类型也不一样，是将整型变量的地址赋值给指针变量。

当实参列表有多个实参时，对实参的求值顺序是不确定的，有的系统按自左向右的顺序求值，有的系统按自右向左的顺序求值。不同编译系统以及同一编译系统的不同版本规定的求值顺序是不同的。有的编译器会进行优化，优化策略不同，求值顺序可能不同。实参求值顺序示例如表 3.4 所示。

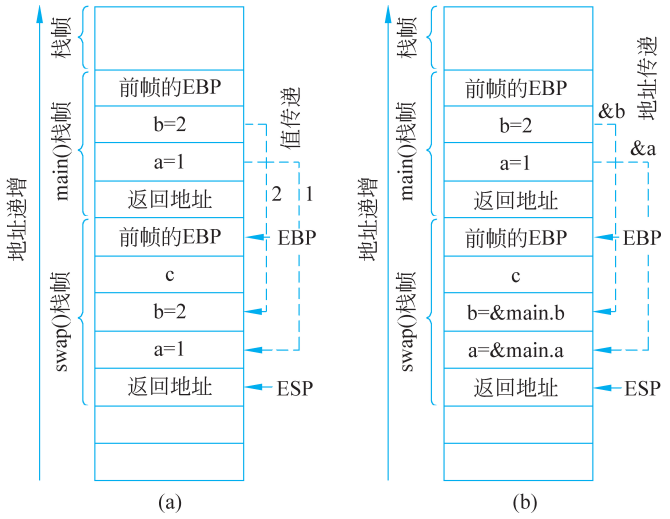

图 3.2 函数栈帧

表 3.4 实参求值顺序示例

扫一扫

示例代码	运行结果

1 #include<stdio.h>
2 void fun(int a,int b,int c) {
3 printf("%d,%d,%d\n",a,b,c);
4 }
5 int main() {
6 int i=1,a=6,b=9;
7 fun(i++,i++,i++); fun(a+3,a=b+2,a+b++);
8 }
	[root@ztg ~]# ./a.out
	3,2,1
	15,12,15
	
	从输出结果可知，该系统按自右向左的顺序进行求值。编程时不要使函数实参之间存在关联，尽量避免在函数参数表达式中对变量进行赋值操作

3.1.4 函数的嵌套与递归

C 语言中的函数定义都是互相平行、独立的，在定义一个函数时，不允许在其函数体内再定义另一个函数。C 语言不能嵌套定义函数，但是 C 语言可以嵌套调用函数，即在调用一个函数的过程中，在该函数的函数体内又调用另一个函数。例如，主函数调用函数 A，函数 A 又调用函数 B。

一个函数在它的函数体内直接或间接地调用该函数本身，这种函数称为递归函数，这种调用关系称为函数的递归调用。在函数内部又调用它本身称为直接递归，在函数 A 内部调用函数 B，在函数 B 中又调用函数 A 称为间接递归。函数的递归调用属于函数嵌套调用的一种。递归调用的实质就是将原来的问题分解为新的问题，而解决新问题时又用到了原有问题的算法。

递归函数执行时将反复调用其自身，每调用一次就进入新的一层(即在栈顶分配一个栈帧)，当最内层的函数(满足递归结束条件)执行完毕后，再一层一层地由里向外退出，每 return 一次就释放栈顶的栈帧。注意，如果无递归结束条件，则会无穷递归，直到程序栈空间耗尽导致程序崩溃(段错误)。下面通过求阶乘的例子理解递归函数的运行情况。n 的阶乘(Factorial)的定义为：n 的阶乘等于 n 乘以 n－1 的阶乘，0 的阶乘等于 1。具体地，0!＝

1,1!=1*(1-1)!=1*0!=1*1=1,n!=n*(n-1)!。阶乘示例如表 3.5 所示。


扫一扫

表 3.5 阶乘示例

示例代码	运行结果

1 #include<stdio.h>
2 int l;
3 long fac(int n) { //递归函数
4 l++;
5 if (n<=1) { //递归结束条件
6 printf("层数=%d n=%d return 1\n",l,n);
7 return 1;
8 } else {
9 printf("层数=%d n=%d return fac(%d)*%d\n",l,n,n-1,n);
10 return fac(n-1)*n; //递归调用
11 }
12 }
13 int main() {
14 int a; printf("输入一个整数: ");
15 scanf("%d", &a);
16 printf("fac(%d) = %ld\n", a, fac(a));
17 return 0;
18 }
	1 [root@ztg ~]# ./a.out
	2 输入一个整数: 5
	3 层数=1 n=5 return fac(4)*5
	4 层数=2 n=4 return fac(3)*4
	5 层数=3 n=3 return fac(2)*3
	6 层数=4 n=2 return fac(1)*2
	7 层数=5 n=1 return 1
	8 fac(5) = 120
	
	输入 5,求 5!。调用 fac(5),函数体中执行 fac(4)*5,调用 fac(4),函数体中执行 fac(3)*4,调用 fac(3),函数体中执行 fac(2)*3,调用 fac(2),函数体中执行 fac(1)*2,调用 fac(1),函数体中直接返回常量 1。n=1 时递归进到最内层,递归结束,开始逐层退出,也就是逐层执行 return 语句

3.1.5 回调函数

C 语言中,回调函数(Callback Function)就是一个通过函数指针调用的函数。回调函数允许用户把需要调用的函数的指针作为参数传递给一个函数,以便该函数在处理相似事件的时候可以灵活使用不同的方法。因为可以把调用者与被调用者分开,所以调用者不关心谁是被调用者,只需知道存在一个具有特定原型和限制条件的被调用函数。回调函数示例如表 3.6 所示。


扫一扫

表 3.6 回调函数示例

示例代码	运行结果

1 #include<stdio.h>
2 int add(int a, int b){ //回调函数
3 printf("this is add\n"); return a+b;
4 }
5 int sub(int a, int b){ //回调函数
6 printf("this is sub\n"); return a-b;
7 }
8 int handle(int x,int y,int (*callback)(int,int)){
9 printf("callback(%d,%d)=%d\n",x,y,callback(x,y));
10 }
11 int main(){
12 int a=5,b=6; handle(a, b, add); handle(2, 10, sub);
13 return 2;
14 }
	1 [root@ztg ~]# ./a.out
	2 this is add
	3 callback(5,6)=11
	4 this is sub
	5 callback(2,10)=-8
	6 [root@ztg ~]# echo $?
	7 2
	
	main 函数中调用 handle 函数时传递了不同的函数指针,进而通过 handle 函数调用了不同的函数,如 add、sub。输出结果中第 6 行,$? 的值就是上一条命令的返回值,也就是主函数的返回值

3.1.6 return 语句

函数有主调函数和被调函数,被调函数完成一定功能后可以通过 return 语句向主调函数返回一个确定的值,该值就是函数的返回值。return 语句表示把程序流程从被调函数转

向主调函数。return 语句有两种：带表达式的 return 语句“return 表达式;”、省略表达式的 return 语句“return;”。①带表达式的 return 语句结束函数的执行并把表达式的值返回给主调函数，定义函数时必须指定函数类型，函数类型和返回值类型要一致，如果不一致，系统规定返回值以函数类型为准进行自动转换，转换后的值为最终的函数值。②省略表达式的 return 语句结束函数的执行并返回到主调函数中该函数的调用处，定义函数时必须指定函数类型为空类型 void。③函数体中可以有多个 return 语句，执行到任何一条 return 语句都会结束函数的执行，返回到主调函数中该函数的调用处。④函数体中如果没有 return 语句，则执行完函数体的最后一条语句后返回主调函数。main 函数是整型函数，如果 main 函数体中没有 return 语句，则默认的返回值是 0。通过执行 echo $? 命令查看 main 函数的返回值，如表 3.6 所示。

3.1.7 全局变量、局部变量和作用域

作用域描述程序中可以访问标识符的区域，一旦离开其作用域，程序便不能再访问该标识符。C 语言中的变量的作用域可以是块作用域、函数作用域、函数原型作用域、文件作用域。块是用一对花括号括起来的代码区域，整个函数体是一个块，函数中的复合语句也是一个块，定义在块中的变量具有块作用域。变量在块作用域的可见范围从变量定义处到包含该定义的块的末尾。函数形参声明虽然在函数左花括号前，但是它们属于函数体这个块，具有函数作用域。函数原型作用域的范围从形参定义处到函数原型声明结束，这意味着编译器在处理函数原型中的形参时只关心它的类型，而形参名通常无关紧要，即使有形参名，也不必与函数定义中的形参名相同。定义在所有函数外的变量具有文件作用域，文件作用域变量(全局变量)从它定义处到该定义所在文件的末尾均可见。

全局变量也称为外部变量，是在函数外定义的变量，它不属于任何一个函数，其作用域从变量的定义处开始，到本程序文件的结尾。全局变量只能用常量表达式初始化。在作用域内全局变量可以被各个函数使用。全局变量也遵循先定义后使用的原则。如果在函数中使用该函数后面定义的全局变量，应在此函数内使用 extern 关键字声明该全局变量，语法为“extern 类型说明符 变量名;”，全局变量的声明表明在函数内要使用某全局变量，因此声明时不能进行赋初值等操作。在程序开始运行时为全局变量分配存储空间，在程序结束时释放全局变量占用的存储空间。当需要函数返回多个值时，除函数体中的 return 语句返回其中一个外，其他的返回值可以通过定义全局变量来处理。因为根据全局变量的特点，在被调用函数中改变了多个全局变量的值，相当于其主调函数全局变量的值也发生了变化，也就相当于返回了多个值。

局部变量也称为内部变量，是在函数内定义的变量，其作用域仅限于函数内(从该变量被定义开始到函数结束)，一个函数中定义的局部变量不能被其他函数使用。函数内定义的变量、函数的形参变量都属于局部变量。注意，在复合语句(由花括弧括起来)内定义的局部变量，其作用域仅限于该复合语句块内。局部变量仅在其作用域内可见，在作用域外不能被访问。局部变量的存储空间在每次函数调用时分配(存在于该函数的栈帧)，在函数返回时释放。

如果函数内定义的局部变量与全局变量重名，则函数在使用该变量时会用到同名的局部变量，而不会用到全局变量，也就是会用局部变量覆盖全局变量，只有局部变量起效果。

3.1.8 变量的存储类别及生存期

变量的数据类型说明它占用多大的内存空间、可以对其进行什么样的操作。除了数据类型，变量还有一个属性，称为存储类别。存储类别决定变量的作用域和生存期。存储类别是指变量占用内存空间的方式，也称为存储方式。存储类别分为两种：静态存储、动态存储。①静态存储变量通常在变量定义时就分定存储单元并一直保持不变，直至整个程序结束。全局变量属于此类存储方式。②动态存储变量是在程序执行过程中定义它时才分配存储单元，使用完毕立即释放。比如函数的形参，在函数定义时并不给形参分配存储单元，只在函数被调用时才予以分配，调用函数完毕立即释放。如果一个函数被多次调用，则反复地分配、释放形参的存储单元。

生存期表示变量存在的时间，是程序运行过程中变量从创建到销毁的一段时间。生存期的长短取决于变量的存储类别，也就是它所在的内存区域。因此，知道变量的存储类别，就可以知道变量的生存期。静态存储变量是一直存在的，而动态存储变量则因程序执行需要而时而存在时而消失。生存期和作用域是从时间和空间这两个不同的角度描述变量的特性，这两者既有联系，又有区别。

如图 3.1 所示，在进程地址空间中，常量区(代码段)、数据区和栈区可用来存放变量的值。①常量区和数据区的内存在程序启动时就已经由操作系统分配好了，占用的空间固定，程序运行期间不再改变，程序运行结束后才由操作系统释放；它可以存放全局变量、静态变量、一般常量和字符串常量。②栈区的存储单元在程序运行期间由系统根据需要分配，占用的空间实时改变，使用完毕后立即释放，不必等到程序运行结束；它可以存放局部变量、函数参数等。

C 语言有 4 个关键字用来控制变量在内存中的存放区域：auto(自动变量)、static(静态变量)、register(寄存器变量)、extern(外部变量)。自动变量和寄存器变量属于动态存储方式，外部变量和静态变量属于静态存储方式。

auto：自动变量的作用域仅限于定义该变量的函数或复合语句内。自动变量属于动态存储方式，只有在定义该变量的函数被调用时才给它分配存储单元，开始它的生存期。函数调用结束，释放存储单元，结束生存期。因此，函数调用结束后，自动变量的值不能保留。在复合语句中定义的自动变量，在退出复合语句后也不再使用，否则将引起错误。由于自动变量的作用域和生存期都局限于定义它的函数或复合语句内，因此不同的作用域中使用同名的变量而不会混淆，在函数内定义的自动变量可与在该函数内复合语句中定义的自动变量同名。定义自动变量时如果没有赋初值，则其值是一个随机数。因为所有的局部变量默认是 auto，所以 auto 很少用到。也就是说，定义变量时加不加 auto 都一样，所以一般把它省略，不必多此一举。例如“int n=1;”与“auto int n=1;”的效果完全一样。

static：static 声明的变量称为静态变量。静态变量属于静态存储方式，不管它是全局的还是局部的，都存储在静态数据区(全局变量本来就存储在静态数据区，即使不加 static)。自动变量属于动态存储方式，但是可以用 static 定义它为静态自动变量(静态局部变量)，从而属于静态存储方式。①静态局部变量在函数或复合语句内定义，在作用域结束时并不消失，它的生存期为整个源程序，其作用域仍与自动变量相同，即只能在定义该变量的函数或复合语句内使用该变量。退出该函数或复合语句后，尽管该变量还继续存在，但不能使用

它。②在全局变量(外部变量)的说明之前冠以 static 就构成了静态全局变量(静态外部变量),全局变量和非静态全局变量在存储方式上相同,都属于静态存储方式,区别在于作用域不同,一个源程序由多个源文件构成时,全局变量的作用域是整个源程序,而静态全局变量的作用域只在定义该变量的源文件内有效,在其他源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内,因此可以避免和其他源文件中的同名变量冲突。静态数据区的数据在程序启动时就会初始化,直到程序运行结束;对于函数或复合语句内的静态局部变量,即使代码块执行结束,也不会销毁。注意:静态数据区的变量只能初始化(定义)一次,以后它不能再被初始化,只能改变它的值,示例如表 3.7 所示。add 函数中定义了一个静态局部变量 sum,它存储在静态数据区(静态数据区的变量若不赋初值,则默认初值为 0),add 函数即使执行结束,也不会销毁,下次调用它继续有效。静态数据区的变量只能初始化一次,第一次调用 add 函数时已经对 sum 进行了初始化,所以再次调用时就不会初始化了,也就是说,再次调用 add 函数时"static int sum=0;"语句无效。静态局部变量虽然存储在静态数据区,但是它的作用域仅限于定义它的代码块,add 函数中的 sum 在函数外无效,与 main 函数中的 sum 不冲突,除了变量名一样,没有任何关系。

表 3.7 静态变量和 extern 示例


扫一扫

静态变量示例代码

```
#include<stdio.h>
void add(int n){
  static int sum=10;//静态自动变量
  sum+=n;
  printf("add.sum=%d\n",sum);
}
int main(){
  int sum=0, i;  //自动变量
  for(i=1;i<=3;i++) add(i);
  printf("main.sum=%d\n",sum);
  return 0;
}
```

运行结果

```
add.sum=11
add.sum=13
add.sum=16
main.sum=0
```

extern 示例:1.c

```
#include<stdio.h>
extern void fun();
extern int x;
extern int s;
int y;
int main(){
 printf("s=%d\n",s);
 printf("x=%d in 2.c\n",x);
 fun();
}
int s=3;
```

extern 示例:2.c

```
#include<stdio.h>
int x=5;  extern int s,y;
void fun(){
 printf("s=%d in 1.c\n",s);
 printf("y=%d in 1.c\n",y);
}
```

运行结果如下:

```
[root@ztg ~]# gcc 1.c 2.c
[root@ztg ~]# ./a.out
s=3
x=5 in 2.c
s=3 in 1.c
y=0 in 1.c
```

register:是寄存器变量的说明符,寄存器变量存放在 CPU 的寄存器中,使用时,不需要访问内存,直接从寄存器中读写,这样,可提高效率。一般情况下,变量的值是存储在内存中的,CPU 每次使用数据都要从内存中读取。如果有一些变量使用非常频繁,从内存中读取就会消耗很多时间,例如 for 循环语句"for(int i=0; i<1000; i++){}",CPU 为了获得 i,会读取 1000 次内存。为了解决这个问题,可以将使用频繁的变量放在 CPU 的通用寄存器中,这样,使用该变量时就不必访问内存了,直接从寄存器中读取,可大大提高程序的运行效率。不过,寄存器的数量是有限的,通常把使用最频繁的变量定义为 register。寄存器的长度一般和机器的字长一致,只有较短的类型(如 int、char、short 等)才适合定义为寄存器变量,如 double 等较大的类型,不推荐将其定义为寄存器类型。注意,为寄存器变量分配寄存器是动态完成的,属于动态存储方式,只有局部自动变量和形参才能定义为寄存器变量,局部静态变量不能定义为寄存器变量。另外,CPU 的寄存器数目有限,即使定义了寄存器变量,编译器也可能并不真正为其分配寄存器,而是将其当作普通的 auto 变量对待,为其分配栈内存。当然,有些优秀的编译器,能自动识别使用频繁的变量,如循环控制变量等,在有可用的寄存器时,即使没有使用 register 关键字,也自动为其分配寄存器,无须由程序员

指定。

extern：外部变量的类型说明符是 extern。当一个源程序由若干个源文件构成时，在一个源文件中定义的全局变量(外部变量)可以在所有源文件中访问，不过需要使用 extern 关键字对该变量进行声明，示例如表 3.7 所示。

变量说明的完整形式为："存储类别说明符 数据类型说明符 变量名，变量名，…；"，例如"static int a，b，c[5]；""auto double d1，d2；""extern float x，y；""char c1，c2；"。C 语言规定，函数内未加存储类别说明的变量均视为自动变量，也就是说，自动变量可省去存储类别说明符 auto。

3.1.9 内部函数和外部函数

函数一旦定义后，就可被其他函数调用。C 语言把函数分为内部函数和外部函数。

内部函数：当一个源程序由多个源文件构成时，如果在一个源文件 A 中定义的函数只能被源文件 A 中的函数调用，而不能被同一源程序其他文件中的函数调用，则这种函数称为内部函数。定义内部函数的一般形式是"static 类型说明符 函数名(形参列表){}"。内部函数也称为静态函数，此处 static 的含义不是指存储方式，而是指对函数的调用范围只局限于本文件。因此，在不同的源文件中定义同名的静态函数不会引起冲突。

外部函数：外部函数在整个源程序中都有效，其定义的一般形式是"extern 类型说明符 函数名(形参列表){}"。在函数定义中如果没有说明 extern 或 static，则隐含为 extern。在一个源文件的函数中调用其他源文件中定义的外部函数时，应该使用 extern 对被调函数进行声明，声明的一般形式是"extern 类型说明符 函数名(形参列表)；"，示例如表 3.7 所示。

3.2 预处理

3.2.1 预处理的步骤

编译预处理是指对 C 源程序正式编译之前所做的文本替换之类的工作，它由 C 预处理器负责完成。C 语言对一个源文件进行编译时，系统将自动调用预处理器对源程序中的预处理指令进行处理，预处理结束后自动调用编译器对源程序进行编译(词法扫描和语法解析)。

预处理的具体步骤如下。

1. 把三联符替换成相应的单字符

三联符(Trigraph)是以?? 开头的三个字符，预处理器会将这些三联符替换成相应的单字符，替换规则为：??=替换为#，??/替换为\，??'替换为^，??(替换为[，??)替换为]，??!替换为|，??<替换为{，??>替换为}，??-替换为~。

2. 将多个物理行替换成一个逻辑行

把用\字符续行的多行代码接成一行，这种续行的写法要求\后面紧跟换行符(回车)，中间不能有其他空白字符。

3. 将注释替换为空格

不管是单行注释还是多行注释，都替换成一个空格。三联符、换行符和注释的示例如表 3.8 所示。

表 3.8　三联符、换行符和注释的示例

示例代码，源代码文件 1.c	运 行 结 果

??=include<stdio.h>
int main()??<
  printf("??= ??/??/ ??' ??(??) ??! ??< ??> ??-\n");
  printf("hello w\
    orld!\n");
  printf("hello w\
orld!\n");
  int/*单行注释*/x=5; printf("x=%d\n",x);
  printf("%s %d %s\n", __FILE__, __LINE__, __func__);
??>
	[root@ztg ~]# gcc 1.c -trigraphs
	[root@ztg ~]# ./a.out
	# \ ^ [ ] | { } ~
	hello w orld!
	hello world!
	x=5
	1.c 9 main

4. 将逻辑行划分成预处理标记（Token）和空白字符

经过上面两步处理之后，去掉了一些换行，剩下的代码行称为逻辑代码行。预处理器把逻辑代码行划分成预处理 Token 和空白字符，这时的 Token 包括标识符、整数常量、浮点数常量、字符常量、字符串、运算符和其他符号。

5. include 预处理和宏展开

在 Token 中识别预处理标记，如果遇到 # include 标记，则把相应的源文件包含进来，并对源文件做以上 1～4 步的预处理。如果遇到宏定义，则做宏展开。

6. 将转义序列替换

找出字符常量或字符串中的转义序列，用相应字节替换它，比如把\n 替换为字节 0x0a。

7. 连接字符串

把相邻的字符串连接起来。

8. 丢弃空白字符

经过以上处理之后，把空白字符（包括空格、换行、水平 Tab、垂直 Tab、分页符）丢弃，把 Token 交给 C 编译器做语法解析，这时就不再是预处理 Token，而是 C Token。

3.2.2　宏定义和内联函数

用一个标识符表示一个字符串称为宏定义，标识符称为宏名。宏名遵循标识符的命名规则，习惯上全大写（也允许用小写字母），便于与变量区别。在编译预处理时，用宏定义中的字符串替换程序中出现的所有宏名，这个过程称为宏替换或宏展开。宏定义时要用宏定义指令 define，无参宏定义的语法为“# define 宏名 宏体字符串”，例如“# define PI 3.1415926”，PI 为宏名，3.1415926 为宏体字符串。宏名也叫符号常量，可以像变量一样在代码中使用。# 开头表示这是一条预处理指令，预处理器指令从 # 开始运行，到后面的第一个换行符为止，也就是说，指令的长度仅限于一行（逻辑行）。宏体字符串是宏名所要替换的一串字符，字符串不需要用双引号括起来，如果用双引号括起来，那么双引号也是宏体的一

部分，将被一起替换。宏体字符串可以是任意形式的单行连续字符序列，中间可以有空格和制表符 Tab。宏定义必须写在函数之外，其作用域，为从宏定义指令开始到源文件结束。如果要终止其作用域，可使用 undef 指令，语法为“#undef 宏名”，例如“#undef PI”。

C 标准规定了几个特殊的宏，它们在不同地方使用可以自动展开成不同的值，常用的有__FILE__和__LINE__。__FILE__展开为当前源文件的文件名，是一个字符串，__LINE__展开为当前代码行的行号，是一个整数。这两个宏不是用 define 指令定义的，它们是编译器内建的特殊宏。在打印调试信息时，除文件名和行号之外，还可以打印当前函数名，C99 引入一个特殊的标识符__func__支持这一功能。标识符__func__不属于预处理的范畴，但是它的作用和__FILE__、__LINE__类似，示例如表 3.8 所示。

宏定义包含无参宏定义和带参宏定义，无参宏定义也称为宏变量定义或变量式宏定义，带参宏定义也称为宏函数定义或函数式宏定义。带参宏定义的语法为“#define 宏名(形参列表) 宏体字符串”，宏名和形参列表之间不能有空格，例如“#define MAX(x,y) x>y?x:y”。带参宏调用的语法为“宏名(实参列表)”，例如“max=MAX(a,b);”，在宏调用时，用实参 a 和 b 分别替换形参 x 和 y，经预处理后宏展开为“max=a>b?a:b;”。

函数式宏定义和真正的函数调用的不同之处有：①宏函数定义中的形参是标识符，没有类型，宏函数调用中的实参可以是表达式，在宏函数调用时只是进行符号替换，不存在值传递的问题，不为形参分配内存单元，不做参数类型检查，所以宏函数调用传参时要格外小心。真正函数的形参和实参是两个不同的量，各有各的作用域，调用时要把实参值赋给形参，并且进行参数类型检查。②调用真正函数的代码和调用函数式宏定义的代码编译生成的指令不同。真正函数的函数体要编译生成机器指令，而函数式宏定义本身不必编译生成机器指令，宏函数调用替换为宏函数体。③定义宏函数时最好将宏函数体中的每个形参都用括号括起来，比如#define MAX(a, b) ((a)>(b)? (a):(b))。如果省去内层括号，则为#define MAX(a, b) (a>b? a:b)，对 MAX(x||y, i&&j)宏展开为(x||y>i&&j? x||y:i&&j)，可见运算的优先级发生了错乱。

尽管函数式宏定义和真正的函数相比有很多缺点，但只要小心使用，还是会显著提高代码的执行效率，毕竟省去了分配和释放栈帧、传参等一系列工作，因此那些简短并且被频繁调用的函数经常用函数式宏定义来代替实现。为了避免函数调用带来的开销，C99 还提供了另外一种方法，即内联函数(inline function)。

C99 引入了一个新关键字 inline，用于定义内联函数。内联函数会在它被调用的位置上展开，也就是说，内联函数的函数体代码会替换内联函数调用表达式，因此系统在调用内联函数时，无须再为被调函数分配和释放栈帧，少了普通函数的调用开销，程序执行效率会得到一定提升，所以内联函数的调用不同于普通函数的调用。C 语言标准规定内联函数的定义与调用该函数的代码必须在同一文件中。因此，通常同时使用函数说明符 inline 和存储类别说明符 static 定义内联函数，例如 static inline int max(int a, int b){return a>b?a:b;}，这种用法在 Linux 内核源代码中很常见。虽然内联函数可以避免函数调用带来的开销，但是要付出一定的代价。普通函数只需要编译出一份二进制代码就可以被其他函数调用，而内联函数只是将自身代码展开到被调用处，会使整个程序代码变长，占用更多的内存空间。采用内联函数实质是以空间换时间。因此，建议把那些对时间要求比较高、代码长度比较短的函数定义为内联函数。

3.2.3 条件编译

按不同的条件编译程序的不同部分，从而产生不同的目标代码文件，这称为条件编译。条件编译有三种形式，如表 3.9 所示。# if 后面跟的是整型常量表达式，而 # ifdef 和 # ifndef 后面跟的只能是一个宏名。

表 3.9 条件编译的三种形式

第一种形式	第二种形式	第三种形式
1 #ifdef 标识符1 2 程序段1 3 #elif 标识符2 4 程序段2 5 #else 6 程序段3 7 #endif	1 #ifndef 标识符 2 程序段1 3 #else 4 程序段2 5 #endif	1 #if 条件表达式1 2 程序段1 3 #elif 条件表达式2 4 程序段2 5 #else 6 程序段3 7 #endif
功能：若标识符 1 被 define 指令定义过，则对程序段 1 进行编译，若标识符 2 被 define 指令定义过，则对程序段 2 进行编译，否则对程序段 3 进行编译	功能：若标识符未被 define 指令定义过，则对程序段 1 进行编译，否则对程序段 2 进行编译	功能：若条件表达式 1 的值为真（非 0），则对程序段 1 进行编译，若条件表达式 2 的值为真，则对程序段 2 进行编译，否则对程序段 3 进行编译

条件编译主要应用于程序的移植和调试。条件预处理指令用于源代码配置管理的示例代码，如表 3.10 所示。假设这段程序是为多平台编写的，在 x86 平台上需要定义 x 为 short 型，在 x64 平台上需要定义 x 为 long 型，在 51 单片机（x51）上需要定义 x 为 char 型，对其他平台暂不提供支持，就可以用条件预处理指示写。在预处理这段代码时，若满足不同的编译条件，则包含不同的代码段。

表 3.10 条件编译的两种形式

第 1 种形式	运行结果	第 2 种形式	运行结果
1 #include<stdio.h> 2 #ifdef x86 3 short x; 4 #elif x64 5 long x; 6 #elif x51 7 char x; 8 #else 9 #error ERROR 10 #endif 11 int main(){ 12 int s=sizeof(x); 13 printf("%d\n",s); 14 }	1 [root@ztg ~]# gcc -Dx86 1.c 2 [root@ztg ~]# ./a.out 3 2 4 [root@ztg ~]# gcc -Dx64 1.c 5 [root@ztg ~]# ./a.out 6 8 7 [root@ztg ~]# gcc -Dx51 1.c 8 [root@ztg ~]# ./a.out 9 1 10 [root@ztg ~]#	1 #include<stdio.h> 2 #if ARCH==8086 3 short x; 4 #elif ARCH==8664 5 long x; 6 #elif ARCH==8051 7 char x; 8 #else 9 #error ERROR 10 #endif 11 int main(){ 12 int s=sizeof(x); 13 printf("%d\n",s); 14 }	1 [root@ztg ~]# gcc -DARCH=8086 1.c 2 [root@ztg ~]# ./a.out 3 2 4 [root@ztg ~]# gcc -DARCH=8664 1.c 5 [root@ztg ~]# ./a.out 6 8 7 [root@ztg ~]# gcc -DARCH=8051 1.c 8 [root@ztg ~]# ./a.out 9 1

3.2.4 文件包含

头文件是扩展名为.h 的文件，建议把所有的常量、宏定义、全局变量和函数原型声明写在头文件中，使它们可被多个源文件（.c 文件）引用。在源文件中使用头文件，需要用 C 预处理指令 include 引用它。一条 include 指令只能包含一个头文件，若要包含多个头文件，则

需用多条 include 指令。引用头文件相当于复制头文件的内容，也就是用头文件的内容替换 include 指令行，从而把头文件和当前源文件连接成一个源文件。由于头文件有两种（程序员编写的头文件、编译器自带的头文件），因此文件包含指令有两种形式：“#include <头文件名>”、“#include "头文件名"”。使用尖括号和双引号的区别在于头文件的搜索路径不同，一般情况下，使用尖括号主要包含的是编译器自带的头文件，在文件包含目录中查找，而不在源文件目录查找；使用双引号主要包含的是程序员编写的头文件，首先在当前的源文件目录中查找，如果没有找到，再到文件包含目录中查找。文件包含目录是由程序员在设置编译器环境时设置的。也就是说，使用双引号比使用尖括号多了一个查找路径，它的功能更为强大。前面示例中一直使用尖括号引用标准头文件，其实也可以使用双引号，例如#include "stdio.h"。stdio.h 是标准头文件，存放于系统路径（文件包含目录）下，所以使用尖括号和双引号都能成功引用。自己编写的头文件一般存放于当前项目路径中，因此不能使用尖括号，只能使用双引号。不过，如果把当前项目所在路径添加到系统路径（可以使用 gcc 的-I 选项添加），这样也可以使用尖括号，但是不建议这么做。建议读者使用尖括号引用标准头文件，使用双引号引用自定义头文件，这样可以提高代码的可读性，很容易看是哪类头文件。

如果一个头文件被引用多次，编译器会对该头文件处理多次，这将产生错误。为了防止这种情况，标准的做法是把文件的整个内容放在条件编译语句中，使用语法及示例如表 3.11 所示。ifndef 指令后面的宏名的命名方法是将头文件名中的字母都改为大写，“.”修改为“_”，这种命名方法可以尽可能避免一个项目中的宏名冲突问题。

表 3.11　防止头文件被引用多次

条件编译语句	示例，头文件 headerfile.h
1 #ifndef HEADER_FILE 2 #define HEADER_FILE 3 the entire header file 4 #endif	1 #ifndef HEADERFILE_H 2 #define HEADERFILE_H 3 int fun(void); 4 #endif

3.3 指针

指针是 C 语言中最重要的组成部分，使用指针编程便于表示各种数据结构，可提高程序的编写质量、编译效率和执行速度。通过指针可使主调函数和被调函数之间共享变量或数据结构，并且可以实现动态内存分配。

3.3.1 指针的基本运算

1. 指针和指针变量

在计算机中，所有要被 CPU 处理的数据都需要存放在内部存储器（内存）中。一般把内存中的一个字节称为一个内存单元，为了正确访问这些内存单元，必须为每个内存单元进行编号。内存单元的编号也叫地址。根据内存单元的编号或地址就可以准确找到所需的内

存单元。通常把这个地址称为指针。通俗地讲，指针就是地址，地址就是指针。对于一个内存单元来说，单元的地址，即为指针，其中存放的数据才是该单元的内容。

存放指针的变量称为指针变量。一个指针变量的值就是某个内存单元的地址，或称为某个内存单元的指针。严格地说，一个地址值是一个常量。一个指针变量却可以被赋予不同的地址值，是变量。指针的值是一个地址。

2. 指针变量的定义

在C语言中，指针是有类型的，在定义指针变量时指定指针类型。例如“int *pa, *pb;”，在同一语句中定义多个指针变量，每个变量都要有*号，定义指针的*号前后的空格都可以省略，写成“int *p, *q;”也算对，但*号通常和类型int之间留空格，而和变量名写在一起。

一种数据类型往往都占有一组连续的内存单元，例如一个double型的指针，它的值是一个内存单元的地址，这个地址值其实是一个double型数据的首地址，这个数据实际占据8字节的内存单元。也就是说，根据指针的值可以找到它所指向数据的首地址，根据指针类型，可以知道它所指向数据占用多少个内存单元。指针变量的定义：“类型说明符 *变量名;”。例如，“int *p;”表示p是一个指针变量，它的值是某个整型变量的地址。或者说p指向一个整型变量。至于p究竟指向哪一个整型变量，应由向p赋予的地址决定。“float *f;”表示f是指向单精度浮点变量的指针变量。“char *c;”表示c是指向字符变量的指针变量。一个指针应尽量始终指向同一类型的变量。

在栈上分配的变量的初始值是不确定的，也就是说，指针p所指向的内存地址是不确定的，后面用*p访问不确定的地址就会导致不确定的后果，如果导致段错误还比较容易改正，如果意外改写了数据而导致在随后的运行中出错，就很难找到错误原因了。像这种指向不确定地址的指针称为野指针，为避免出现野指针，在定义指针变量时应该给它赋予明确的初值，或者把它初始化为NULL，例如“int *p = NULL;”，此时如果“*p = 0;”，程序运行时会段错误。NULL在C标准库的头文件stddef.h中定义“#define NULL ((void *)0)”，就是把地址0转换成指针类型，称为空指针，它的特殊之处在于，操作系统不会把任何数据保存在地址0及其附近，也不会把地址0～0xfff的页面映射到物理内存，所以任何对地址0的访问都会立刻导致段错误。“*p = 0;”会导致段错误，相比之下，野指针的错误很难发现和排除。

3. 指针变量的赋值

C语言提供了两个指针运算符：取地址运算符&和取内容运算符*。取地址运算符&用来求变量的地址，取内容运算符*用来取出指针变量所指向的变量(存储单元)的值。

指针变量的赋值包括：①把变量地址赋予指针变量，例如“int a, *pi, *p=&a; pi=&a;”，定义指针变量p时赋初值，使用赋值语句为指针变量pi单独赋值；②同类型指针变量相互赋值，例如“int a, *pi, *p=&a; pi=p;”；③把数组、字符串的首地址赋予指针变量，例如“int a[5], *pa=a; char *pc="abc";”；④把函数入口地址赋予指针变量。

把一个数值赋予指针变量是危险的，如“int *p; p=1000;”。

取内容运算符*之后跟的变量必须是指针类型，例如“int a, b=5, *p=&b; a=*p;”。

正确的指针变量赋值示例："char c, *pc = &c; int i, *pi = &i;"，内存布局如图 3.3 左侧所示。这里的 & 是取地址运算符，&i 表示取变量 i 的地址，int *pi = &i;表示定义一个指向 int 型的指针变量 pi，并用 i 的地址初始化 pi。还定义了一个字符型变量 c 和一个指向 c 的字符型指针 pc，注意 pi 和 pc 虽然是不同类型的指针变量，但它们的内存单元都占 8 字节，因为是在 64 位平台，所以要保存 64 位的虚拟地址。

图 3.3 变量的内存布局以及它们之间的关系

指针之间可以相互赋值，也可以用一个指针初始化另一个指针，例如"int *pti = pi;"或"int *pti ; pti = pi;"表示 pi 和 pti 指向同一个变量。用一个指针给另一个指针赋值时要注意，两个指针必须是同一类型的。pi 是 int *型的指针，pc 是 char *型的指针，pi = pc;这样赋值就是错误的。但是可以先强制类型转换，然后赋值：pi = (int *)pc;，把 char *指针的值赋给 int *指针。

错误的指针变量赋值示例："char c, *pc = &c; int i, *pi = (int *)&c;"，内存布局如图 3.3 右侧所示，此时 pi 指向的地址和 pc 一样，都是 0x7fff0e9ff51f。通过 *pc 只能访问一字节，而通过 *pi 可以访问 4 个字节，后 3 字节已经不属于变量 c 了。pi 指向的从 0x7fff0e9ff51f 开始的 4 字节的数如果不是一个有意义的数，则肯定是一个错误的数。pi 指向的是从 0x7fff0e9ff51f 开始的 4 字节的整数，肯定是一个错误的数。

因此，使用指针要特别小心，很容易将指针指向错误的地址，访问这样的地址可能导致段错误，可能读到无意义的值，也可能意外改写了某些数据，使得程序在随后的运行中出错。

注意 为了描述方便，图 3.3 没有考虑字节对齐的情况。

如果要改变 pi 所指向的整型变量的值，比如把变量 i 的值增加 5，可以写成"*pi = *pi + 5;"，这里的 * 号是指针间接寻址运算符，*pi 表示取指针 pi 所指向变量的值，指针有时称为变量的引用。

& 运算符的操作数必须是左值，因为只有左值才表示一个内存单元，才会有地址，运算结果是指针类型。 * 运算符的操作数必须是指针类型，运算结果可以做左值。所以，如果表达式 E 可以做左值，则 *&E 和 E 等价，如果表达式 E 是指针类型，则 &*E 和 E 等价。

4. 指针变量的加减运算

对指向数组、字符串的指针变量可以进行加减运算。指向同一数组的两个指针变量可以相减。对指向其他类型的指针变量作加减运算是无意义的。

若 pa 是指针变量，n 是整数，则指针变量与整数的加减运算 pa+n、pa-n、pa++、

++pa、pa--、--pa 等都是合法的。指针变量加或减一个整数 n 的意义是把指针指向的当前位置(通常指向某数组元素)向前或向后移动 n 个位置(以指针指向的数据类型大小为一个单位)。注意,数组指针变量向前或向后移动一个位置和地址值加 1 或减 1 在概念上是不同的,因为数组可以有不同的类型,各种类型的数组元素所占的字节长度是不同的,示例如下。

```
int a[5], *pa;
pa=a;           // pa为数组a的首地址，也就是指向a[0]
pa=pa+2;        // pa指向a[2]，即pa的值为&a[2]
pa--;           // pa指向a[1]，即pa=&a[1]
```

两个指针相减所得之差是两个指针所指向的数组元素下标的差,也就是两个元素相差的元素个数,只有指向同一数组中元素的指针之间相减才有意义。两个指针变量的加法运算没有意义。

5. 指针变量的关系运算

指针之间的比较运算比的是地址值,两个指针变量只有指向同一数组的元素时才能进行比较或相减运算,否则运算无意义。指向同一数组的两个指针变量可以进行关系运算(>、>=、<、<=、==、!=)。假设 pa1 和 pa2 两个指针变量指向同一数组中的元素,pa1==pa2 表示 pa1 和 pa2 指向同一数组元素;pa1>pa2 表示 pa1 变量的值(地址值)大于 pa2 变量的值,也就是 pa1 指向数组元素的下标值大于 pa2 指向数组元素的下标值;pa1<pa2 的含义和 pa1>pa2 相反。

指针可与 0 比较,p==0 表示 p 为空指针,不指向任何变量。

6. 通用指针

void * 类型指针称为通用指针或万能指针。可以将通用指针转换为任意其他类型的指针,任意其他类型的指针也可以转换为通用指针,void * 指针与其他类型的指针之间可以隐式转换,而不必用强制类型转换运算符。注意,只能定义 void * 指针,不能定义 void 型变量,因为 void * 指针和别的指针一样都占 8 字节(64 位平台是 8 字节,32 位平台是 4 字节),如果定义 void 型变量(也就是类型暂时不确定的变量),编译器不知道该分配几字节给变量。void * 指针常用于函数接口。

7. 指针类型的参数

再看表 3.3 中采用地址传递的 swap 函数。调用函数的传参过程相当于定义并用实参初始化形参,swap(&a,&b)这个调用相当于“int *pa = &a; int *pb = &b;”,所以 pa 和 pb 分别指向 main 函数的局部变量 a 和 b,在 swap 函数中读写 *pa 和 *pb 就是读写 main 函数的 a 和 b。虽然在 swap 函数的作用域中访问不到 a 和 b 这两个变量名,但是可以通过地址访问它们,最终 swap 函数交换了 a 和 b 的值。

3.3.2 指针与数组

数组是内存中的一组连续地址空间,其首地址是数组名。数组名作为数组的首地址,它是一个常量指针。指向数组的指针变量称为数组指针变量。指向一维数组的指针变量称为一维数组指针变量。指向二维数组的指针变量称为二维数组指针变量。

1. 一维数组

一维数组指针变量加1表示让指针指向后一个数组元素，对该指针变量不断加1就能访问到该数组的所有元素。例如“int a[6]={0,1,2,3,4,5}, *p=a;”，此时指针p指向a[0]。一维数组名和一维数组指针的用法如图3.4所示。图3.4中说明了地址之间的关系(a[1]位于a[0]之后4字节处)以及指针与变量之间的关系(指针保存的是变量的地址)。可以让指针p直接指向某个数组元素，例如“p=&a[3];”，或可以写成“p=a+3;”。由于数组名a是常量指针，因此改变其值(如a++、++a)是错误的。在取数组元素时用数组名和用指针的语法一样，但如果把数组名做左值使用，和指针就有区别了。例如pa++是合法的，但a++就不合法，pa=a+1是合法的，但a=pa+1就不合法。一维数组元素及其地址

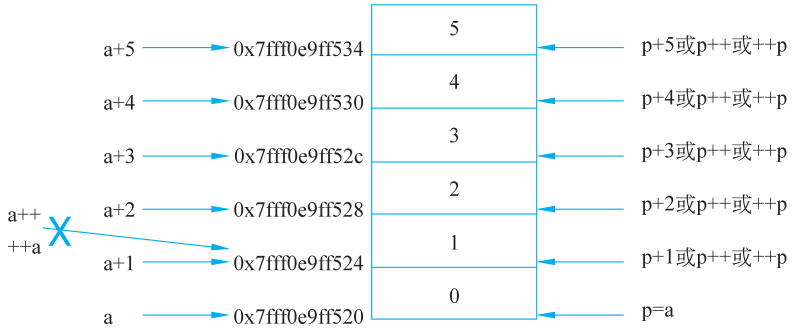

图3.4 一维数组名和一维数组指针的用法

的表示方法可以有多种方式，如表3.12所示。

表3.12 一维数组元素及其地址的表示方法

值	各种等价的表示方法					
a[0]的地址	a	a+0	p	p+0	&a[0]	&p[0]
a[0]的值	*a	*(a+0)	*p	*(p+0)	a[0]	p[0]
a[i]的地址		a+i		p+i	&a[i]	&p[i]
a[i]的值		*(a+i)		*(p+i)	a[i]	p[i]

例如“int a[6], *p=&a[0]; p++;”，首先指针p指向a[0]的地址，由于后缀运算符[ ]的优先级高于取地址运算符&，所以是取a[0]的地址，而不是取a的地址。然后p++让p指向下一个元素(也就是a[1])，由于p是int *指针，一个int型元素占4字节，所以p++使p所指向的地址加4，不是加1。既然指针可以用++运算符，当然也可以用+、-运算符，p+2这个表达式也是有意义的，如图3.4所示，p指向a[0]，则p+2指向a[2]。*(p+2)也可写成p[2]，p就像数组名一样，a[2]之所以能取数组的第2个元素，是因为它等价于*(a+2)，a[2]和p[2]本质上是一样的，都是通过指针间接寻址访问数组元素。由于a做右值使用时和&a[0]是一个意思，所以“int *p=&a[0];”通常写成更简洁的形式“int *p=a;”。

在函数原型中，如果参数是数组，则等价于参数是指针的形式，例如void func(int a[10]){}等价于void func(int *a){}。第一种形式方括号中的数字可以不写，如void func

(int a[])｛｝。参数写成指针形式还是数组形式对编译器来说没区别，都表示这个参数是指针，之所以规定两种形式，是为了给读代码的人提供有用的信息，如果这个参数指向一个元素，通常写成指针的形式，如果这个参数指向一串元素中的首元素，则写成数组的形式。

2. 二维数组

二维数组可以看成由若干个一维数组组成的数组，例如“int a[3][3]＝｛｛0,1,2｝,｛3,4,5｝,｛6,7,8｝｝；int (＊pp)[3]＝a;”。该数组由三个一维数组构成，它们是 a[0]、a[1]和 a[2]。把 a[0]看成一个整体，它是一个一维数组的名字，也就是这个一维数组的首地址，这个一维数组有 3 个元素，它们是 a[0][0]、a[0][1]、a[0][2]。可以把二维数组看成一个一维数组，其中每个元素又是一个一维数组。既然数组 a 可以看成一个一维数组，那么 a[0]就是其第 0 个元素(也就是第 0 行的首地址)、a[1]就是其第 1 个元素(也就是第 1 行的首地址)、a[2]就是其第 2 个元素(也就是第 2 行的首地址)。

可以定义指向“一个变量”的指针，例如“int ＊p＝＊a;”。也可以定义一个指向“一行变量(一维数组)”的指针变量，其语法为“类型说明符 (＊指针变量名)[一行元素的长度];”，类型说明符为所指数组的基本数据类型。＊表示其后的变量是指针类型。例如“int (＊pp)[3]＝a;”，定义了一个指针变量 pp，其指向固定有 3 个元素的一维数组。“int (＊pp)[3]＝&a[0];”等价于“int (＊pp)[3]＝a;”。此时执行 pp＋＋操作后，pp 指向二维数组 a 的下一行的首地址，即 pp 的实际地址值增加 12 字节。

二维数组名和二维数组指针的用法如图 3.5 所示。图 3.5 中说明了地址之间的关系以及指针与变量之间的关系。

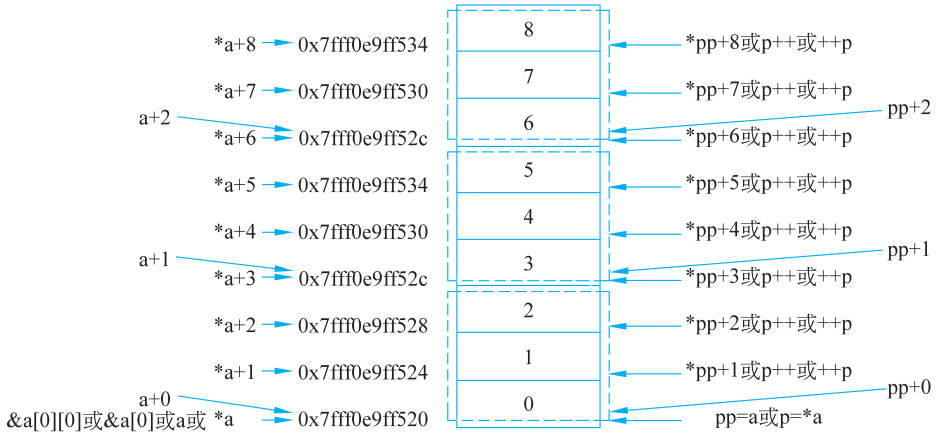

图 3.5　二维数组名和二维数组指针的用法

把二维数组 a 看成一维数组后，其元素 a[0]的大小就是原二维数组一行所包含元素的个数。所以，第 0 行的首地址是 &a[0]，也可以写成 a＋0(也就是 a[0][0]的地址)。同理，第一行的首地址是 &a[1]，也可以写成 a＋1(也就是 a[1][0]的地址)。既然数组 a[0]是一个一维数组，那么 a[0][0]就是其第 0 个元素(也就是二维数组的第 0 行第 0 列)、a[0][1]就是其第 1 个元素(也就是二维数组的第 0 行第 1 列)、a[0][2]就是其第 2 个元素(也就是二维数组的第 0 行第 2 列)。

a[0]是一个一维数组，那么 a[0]本身就是这个数组的首地址。根据一维数组元素地址