第 1 章
二进制安全概述

在网络安全中，二进制安全占有举足轻重的地位，其主要研究方向有软件漏洞挖掘、软件逆向工程、病毒木马分析等，涉及操作系统内核分析、调试与反调试、算法分析、缓冲区溢出等技术。由于经常需要处理二进制数据，因此将该方向称为二进制安全。在网络安全CTF竞赛中，Reverse和PWN（二进制漏洞挖掘）是专门用来考核选手二进制安全能力的竞赛题型。本章主要介绍与二进制安全相关的汇编指令、编译环境和调试工具。

1.1 汇 编 指 令

1.1.1 寄存器

1. x86 寄存器

x86寄存器主要包括通用寄存器、段寄存器、指令指针寄存器和标志寄存器。通用寄存器主要用于各种运算和数据传输，分为数据寄存器（32位为EAX、EBX、ECX和EDX，64位为RAX、RBX、RCX和RDX）和指针变址寄存器（32位为EBP、ESP、ESI和EDI，64位为RBP、RSP、RSI和RDI）。以32位为例，各寄存器功能如下：

- EAX为“累加器”，是加法和乘法指令的默认寄存器。
- EBX为“基地址”寄存器，在内存寻址时存放基地址。
- ECX为计数器，是REP前缀指令和LOOP指令的内定计数器。
- EDX用于存放整数除法产生的余数。
- EBP用于存放栈底指针。
- ESP用于存放栈顶指针。
- ESI/EDI分别叫作“源/目标索引寄存器”。

指令指针寄存器EIP用于存放下一条CPU指令的内存地址，当CPU执行完当前指令后，将从EIP寄存器中读取下一条指令的内存地址，继续执行。如果当前指令为一条跳转指令，如JMP、JE、JNE等，则会改变EIP的值，使得CPU执行指令产生跳跃，从而构成分支和循环程序结构。另外，中断和异常也会影响EIP的值。

段寄存器用于存放段的基地址，段是一块预分配的内存区域，用于存放程序的指令、程序的变量、函数变量和参数等。16位CPU有4个段寄存器：CS（代码段）、DS（数据段）、SS（堆栈段）和ES（附加数据段）。32位CPU增加两个寄存器：FS和GS，它们均为附加寄存器。

标志寄存器称为FLAGS，占2字节，寄存器中的每个标志只占1位，如表1-1所示。

表1-1　标志位

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

各标志位说明如下：

- CF（Carry Flag）进位标志位：运算结果的最高位产生进位或借位，其值为1，否则为0。
- PF（Parity Flag）奇偶标志位：运算结果的低8位中，1的个数为偶数，其值为1，否则为0。
- AF（Auxiliary Carry Flag）辅助进位标志位：在发生下列情况时，AF的值为1，否则为0:
  - 字节操作时，发生低4位向高4位进位或借位。
  - 字操作时，发生低字节向高字节进位或借位。
  - 双字操作时，发生低字向高字进位或借位。
- ZF（Zero Flag）零标志位：运算结果为0，其值为1，否则为0。
- SF（Sign Flag）符号标志位：与运算结果的最高位相同，最高位为1，其值为1，否则为0。
- OF（Overflow Flag）溢出标志位：运算结果超过当前运算位数所能表示的范围，称为溢出，其值为1，否则为0。
- DF（Direction Flags）方向标志位：在执行串处理指令时，如果DF = 0，则SI、DI递增；如果DF = 1，则SI、DI递减。
- TF（Trace Flag）调试标志位：如果TF=1，则处理器每次只执行一条指令，即单步执行；如果TF = 0，则处理器继续执行程序。
- IF（Interrupt Flag）中断允许标志位：用于控制CPU是否允许接收外部中断请求，若IF= 1，则CPU能响应外部中断，否则屏蔽外部中断。

2. ARM 寄存器

ARM处理器共有7种模式，37个寄存器。每种模式都有一组寄存器：一部分是所有模式共用的寄存器，另一部分是模式独自拥有的寄存器。寄存器又分为通用寄存器和状态寄存器：通用寄存器31个，状态寄存器6个（1个CPSR和5个SPSR），如表1-2所示。

表1-2　ARM寄存器

用户模式	系统模式	特权模式	中止模式	未定义指令模式	外部中断模式	快速中断模式
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

通用寄存器通常又分为3类：未分组寄存器（包括R0～R7）、分组寄存器（包括R8～R14）和程序计数器（PC，即R15）。

1）未分组寄存器

所有的处理器模式共用同一个物理寄存器，未被系统用于特殊的用途，可用于所有应用场合。

2）分组寄存器

分组寄存器中的R8～R12，在所有的处理器模式下，每个寄存器共用两个不同的物理寄存器。例如，当处于快速中断模式时，寄存器R8记作R8_fiq，使用的是一个物理寄存器；当处于其他6种模式时，寄存器R8记作R8，使用的是另一个物理寄存器。

分组寄存器R13和R14，每个寄存器共用6个不同的物理寄存器，用户模式和系统模式共用1个，其他5种处理器模式各自对应1个。R13通常用做堆栈指针，R14被称为连接寄存器。

3）程序计数器

程序计数器R15又被记作PC，用于存放当前准备执行的指令的地址，也可作为通用寄存器使用。

3. CPSR 寄存器

CPSR寄存器用于保存当前程序状态，可以在所有处理器模式下被访问，每种模式都有一个专用的程序状态备份寄存器SPSR。当特定的异常中断发生时，SPSR用于存放当前程序状态

寄存器的数据；当异常退出时，用SPSR保存的数据恢复CPSR。CPSR的具体格式如表1-3所示。

表1-3　CPSR寄存器

31	30	29	28	27	26	7	6	5	4	3	21	0
N	Z	C	V	Q	DNMLRAZ	I	F	I	M4	M3	M	M0

1）条件标志位

N（Negative）、Z（Zero）、C（Carry）和V（Overflow）统称为条件标志位。大部分的ARM指令可以依据CPSR中的条件标志位来选择性地执行指令。条件标志位的具体含义如表1-4所示。

表1-4　条件标志位

标　志　位	含　　义
N	该位被设置为当前指令运算结果的bit[31]的值。当两个补码表示的有符号整数运算时，N＝1表示运算的结果为负数，N＝0表示运算的结果为正数或0
Z	Z＝1表示运算结果为0，Z＝0表示运算结果不为0。对于cmp指令（比较指令），Z＝1表示进行比较的两个数大小相等
C	• 加法指令（包括比较指令cmn）：结果产生进位，则C ＝ 1，表示无符号数运算发生上溢出，其他情况下C＝0。 • 减法指令（包括比较指令cmp）：结果产生借位，则C ＝ 0，表示无符号数运算发生下溢出，其他情况下C＝1。 • 移位操作指令：C存储最后一次溢出位的数值，其他非加、减法指令，C的值通常不受影响
V	加、减法运算指令：当操作数和运算结果为二进制的补码表示的带符号数时，V＝1表示符号位溢出，其他的指令通常不影响V位

2）Q 标志位

在ARM v5的E系列处理器中，CPSR的bit[27]称为Q标志位，主要用于表示增强的DSP指令是否发生溢出。同样地，SPSR的bit[27]也称为Q标志位，用于在异常中断发生时保存和恢复CPSR中的Q标志位。

3）CPSR 中的控制位

CPSR的低8位，包括I、F、T及M[4：0]，统称为控制位。当异常中断发生时，控制位将发生变化，在特权级的处理器模式下，软件可以修改这些控制位。

① I中断禁止位：

- 当I＝1时，禁止IRQ中断。
- 当F＝1时，禁止FIQ中断。

通常一旦进入中断服务程序，可以通过置位I和F来禁止中断，但是在本中断服务程序退出前必须恢复I、F位的值。

② T控制位，用来控制指令执行的状态，即指明本指令是ARM指令还是Thumb指令。不同版本的ARM处理器，T控制位的含义也不相同。

ARM v3及更低的版本和ARM v4的非T系列版本的处理器，均没有ARM和Thumb指令的切换，T始终为0。

ARM v4及更高版本的T系列处理器，T控制位含义如下：

- 当T = 0时，表示执行的是ARM指令。
- 当T = 1时，表示执行的是Thumb指令。

ARM v5及更高的版本的非T系列处理器，T控制位的含义如下：

- 当T = 0时，表示执行的是ARM指令。
- 当T = 1时，表示强制下一条执行的指令产生未定义指令中断。

③ M控制位：

控制位M[4:0]称为处理器模式标识位，具体说明如表1-5所示。

表1-5　模式标识位

M[4:0]	处理器模式	可访问的寄存器
0b10000	User	PC，R14～R0，CPSR
0b10001	FIQ	PC，R14_fiq～R8_fiq，R7～R0，CPSR，SPSR_fiq
0b10010	IRQ	PC，R14_irq～R13_irq，R12～R0，CPSR，SPSR_irq
0b10011	Supervisor	PC，R14_svc～R13_svc，R12～R0，CPSR，SPSR_svc
0b10111	Abort	PC，R14_abt～R13_abt，R12～R0，CPSR，SPSR_abt
0b11011	Undefined	PC，R14_und～R13_und，R12～R0，CPSR，SPSR_und
0b11111	System	PC，R14～R0，CPSR（ARM v4及更高版本）

④ CPSR的其他位用于将来ARM版本的扩展。

4. MIPS32 寄存器

MIPS32寄存器分为两类：通用寄存器（GPR）和特殊寄存器。

MIPS32架构中有32个通用寄存器，用编号$0～$31表示，也可以用寄存器的名字表示，如$sp、$gp、fp、$t1、$ta等，如表1-6所示。

表1-6　MIPS 32寄存器

编　　号	寄存器名称	寄存器描述
$0	zero	第0号寄存器，其值始终为0
$1	$at	保留寄存器
$2～$3	$v0～$v1	保存表达式或函数的返回值
$4～$7	$a0～$a3	函数的前4个参数
$8～$15	$t0～$t7	供汇编程序使用的临时寄存器
$16～$23	$s0～$s7	调用子函数时，保存原寄存器的值
$24～$25	$t8～$t9	供汇编程序使用的临时寄存器，补充$t0～$t7

（续表）

编　　号	寄存器名称	寄存器描述
$26～$27	$k0～$k1	中断处理函数用于保存系统参数
$28	$gp	全局指针
$29	$sp	堆栈指针，指向堆栈的栈顶
$30	$fp	指向当前堆栈帧的开头
$31	$ra	保存子函数的返回地址

MIPS32架构中有3个特殊寄存器：PC（程序计数器）、HI（乘除结果高位寄存器）和LO（乘除结果低位寄存器）。在乘法运算中，HI和LO用于保存乘法运算的结果，HI用于保存高32位，LO保存低32位；在除法运算中，HI用于保存余数，LO用于保存商。

1.1.2　指令集

汇编语言是一种符号语言，与机器语言一一对应，使用助记符表示相应的操作，并遵循一定的语法规则。不同的CPU架构采用不同的汇编指令系统，CPU架构主要有x86、ARM、MIPS等类型。x86为当前CPU主流架构，主要应用于个人计算机、服务器、工作站等领域，国外的代表性厂商有Intel和AMD，国内主要有海光、兆芯、申威等。ARM主要应用于智能手机、平板电脑、物联网设备、嵌入式系统等领域，国外的代表性厂商为高通，国内主要有海思、飞腾等。MIPS主要应用于路由器、交换机、数字电视、游戏机等领域，代表性厂商主要有国内的龙芯。

1. x86 指令集

x86指令集采用CISC（Complex Instruction Set Compute）复杂指令系统，各opcode（汇编对应的机器码）的长度不尽相同。出于兼容性考虑，64位CPU指令集没有摒弃原有的指令集，早期16位8086 CPU指令不仅被x86指令集继承，也被最新的CPU指令集继续沿用。常用的x86汇编指令如表1-7所示。

表1-7 常用的x86汇编指令

指　　令	示　　例	含　　义
mov	mov dst, src	将src赋给dst
xchg	xchg dst1, dst2	互换dst1和dst2
push	push src	将src压栈，esp减1
pop	pop dst	栈顶数据出栈，并赋给dst，esp加1
add	add dst, src	dst +=src
sub	sub dst, src	dst -= src
inc	inc dst	dst += 1
dec	dec dst	dst -= 1
neg	neg dst	dst =- dst
cmp	cmp src1, src2	根据src1 - src2的值，设置状态标志位
and	and dst, src	dst &= src
or	or dst, src	dst |= src

（续表）

指　令	示　例	含　义
xor	xor dst, src	dst ^= src
not	not dst	dst = ~dst
test	test src1, src2	根据src1 & src2的值，设置状态标志位
jmp	jmp addr	跳转到地址addr
call	call addr	将函数返回地址压栈，然后调用函数
ret	ret	函数返回地址出栈，跳转到该地址
syscall	syscall	进入内核，执行系统调用
lea	lea dst, src	将内存地址src赋给dst
nop	nop	空指令

2. ARM 指令集

ARM指令集采用RICS（Reduced Instruction Set Computer）精简指令系统，各opcode的长度保持一致。早期的ARM指令的opcode长度为4字节，由于大部分指令未占满4字节，因此出现了opcode长度为2字节的Thumb指令集，以及部分opcode长度为2字节、部分opcode长度为4字节的Thumb-2指令集。目前，64位的ARM指令集所有指令的opcode长度均为4字节。常用的ARM汇编指令如表1-8所示。

表1-8　常用的ARM汇编指令

指　令	示　例	含　义
add	add r0, r0, #1	r0 = r0 + 1
mov	mov r0, #0x00ff	r0 = 0x00ff
movs	movs r0, r1, lsl #3	将寄存器r1的值左移3位后传递给r0，并影响标志位
mvn	mvn r0, r1	将寄存器r1的值按位求反后传递给r0
sub	sub r0, r0, #1	r0 = r0 - 1
sub	sub r0, r1, r2	r0 = r1 - r2
sub	sub r0, r1, r2, lsl #3	r0 = r1 - (r2 << 3)
ldr	ldr r0, [r1]	r0 = [r1]，将r1存储的内存地址传递给r0
ldr	ldr r0, [r1, #2]	r0 = [r1 + 2]，将r1中的值+2作为内存地址，然后将内存地址存储的值赋给r0
str	str r1, [r0, #0x12]	将r1赋给[r0+0x12]地址
rsb	rsb r0, r0, #0xffff	r0 = 0xffff - r0
rsb	rsb r0, r1, r2	r0 = r2 - r1
and	and r0, r1, r2	r0 = r1 & r2
and	and r0, r0, #3	r0 = r0 & 3
orr	orr r0, r0, #3	r0 = r0 | 3
eor	eor r0, r0, #0f	r0 = r0 ^ 0f
cmp	cmp r1, r0	计算r1 - r0，并改变cpsr标志位
cmn	cmn r1, r0	计算r1 + r0，并改变cpsr标志位
tst	tst r1, #0xf	检测r1的低4位是否为0
teq	teq r1, r2	将r1存储的值和r2存储的值进行异或运算，并修改 cpsr标志位

（续表）

指　令	示　例	含　义
b	b task1	无条件跳转到task1
b	b 0x1234	无条件跳转到绝对地址0x1234
bl	bl task1	自动将下一条指令的地址保存到r1寄存器，再跳转到task1标号执行，执行结束后，需将r1寄存器的值赋给PC寄存器才能跳转回来
bx	bx r0	跳转到r0处执行

3. MIPS 指令集

MIPS指令集采用RISC指令系统，所有指令的opcode长度均为4字节，操作码占用高6位，低26位按格式划分为R型、I型和J型。常用的MIPS汇编指令如表1-9所示。

表1-9　常用的MIPS汇编指令

指　令	示　例	含　义
add	add $s1, $s2, $s3	$s1 = $s2 + $s3
sub	sub $s1, $s2, $s3	$s1 = $s2 - $s3
addi（立即数加法）	addi $s1, $s2, 20	$s1 = $s2 + 20
lw（取字）	lw $s1, 20 ($s2)	$s1 = Memory[$2+20]
sw（存字）	sw $s1, 20 ($s2)	Memory[$s2+20] = $1
and	and $s1, $s2, $s3	$s1 = $s2 & $s3
or	or $s1, $s2, $s3	$s1 = $s2 | $s3
nor（或非）	nor $s1, $s2, $s3	$s1 = ~($s2 | $s3)
sll（逻辑左移）	sll $s1, $s2, 10	$s1 = $s2 << 10
srl（逻辑右移）	srl $s1, $s2, 10	$s1 = $s2 >> 10
beq（等于时跳转）	beq $s1, $s2, 25	if($s1 == $s2) go to PC + 4 +100
bne（不等于时跳转）	bne $s1, $s2, 25	if($s1 != $s2) go to PC + 4 +100
slt（小于时置位）	slt $s1, $s2, $s3	if($s2 < $s3) $s1 = 1; else $s1 = 0
sltu（无符号数比较，小于时置位）	sltu $s1, $s2, $s3	if($s2 < $s3) $s1 = 1; else $s1 = 0
j（跳转）	j 2500	go to 10000
jr（跳转至寄存器所指位置）	jr $ra	go to $ra

1.2　编 译 环 境

1.2.1　x86环境

在Linux中，主要使用gcc编译C代码，使用g++编译C++代码，使用gdb调试程序，使用pwndbg和pwngdb增强调试功能。gcc、g++和gdb的安装比较简单，下面主要演示pwndbg和pwngdb的安装过程。

步骤01　下载pwndbg和pwngdb安装包，并解压到指定的目录，如图1-1所示。

图 1-1

步骤 02　执行“vim ~/.gdbinit”命令，编辑“.gdbinit”文件，依次添加“source /home/ubuntu/ Pwndbg/ gdbinit.py”和“source/home/ubuntu/Pwngdb/pwngdb.py”，如图1-2所示。

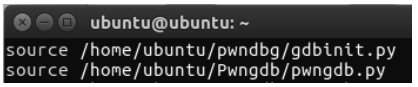

图 1-2

步骤 03　执行gdb命令，结果如图1-3所示。由图可知，pwndbg和pwngdb已经成功安装。

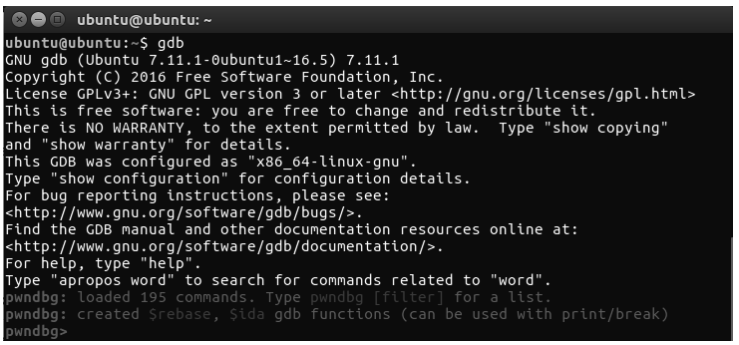

图 1-3

在Windows环境中，主要使用Visual Studio（简称VS）、Dev C++开发工具编译C和C++代码，本书采用Visual Studio 2019，读者可自行安装开发环境。

1.2.2　ARM环境

在Linux环境中，主要使用交叉编译工具编译ARM程序，使用gdb-multiarch调试程序。下面通过案例演示交叉编译工具和gdb-multiarch的安装过程。

步骤 01　访问“https://snapshots.linaro.org/gnu-toolchain/13.0-2022.11-1/aarch64-linux-gnu/”，结果如图1-4所示。

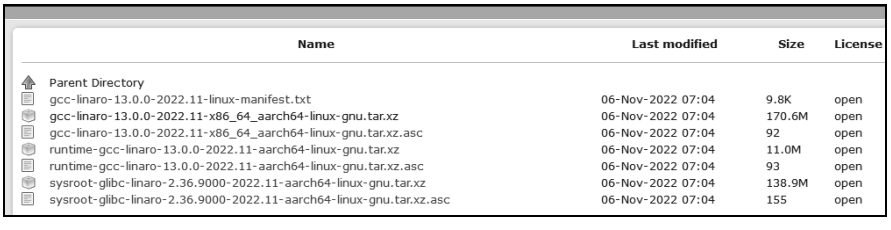

图 1-4

步骤02 下载gcc-linaro-13.0.0-2022.11-x86_64_aarch64-linux-gnu.tar.xz文件并解压，如图1-5所示。

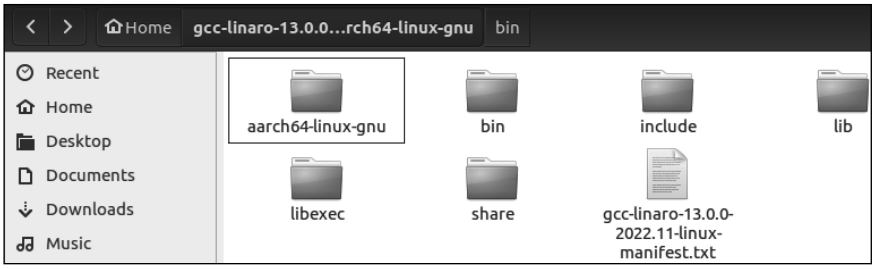

图 1-5

步骤03 执行“vim ~/.bashrc”命令，编辑“.bashrc”文件，在文件尾部添加“Export PATH = $PATH:/home/ubuntu/gcc-linaro-13.0.0-2022.11-x86_64_aarch64-linux-gnu/bin”，如图1-6所示。

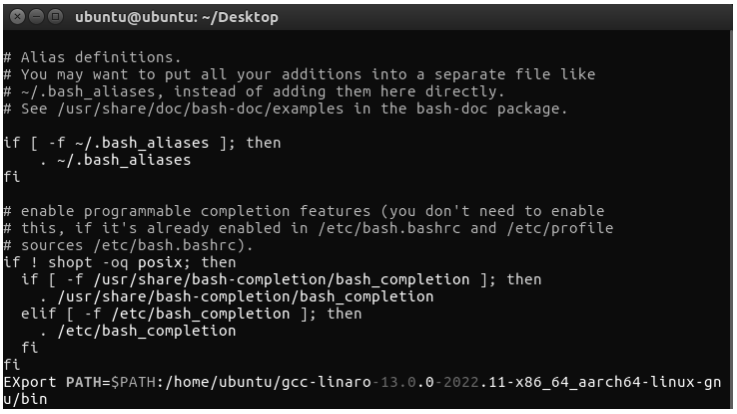

图 1-6

步骤04 执行“source ~/.bashrc”命令使配置文件生效。再执行“aarch64-linux-gnu-gcc -v”命令，结果如图1-7所示。由图可知，交叉编译工具安装成功。

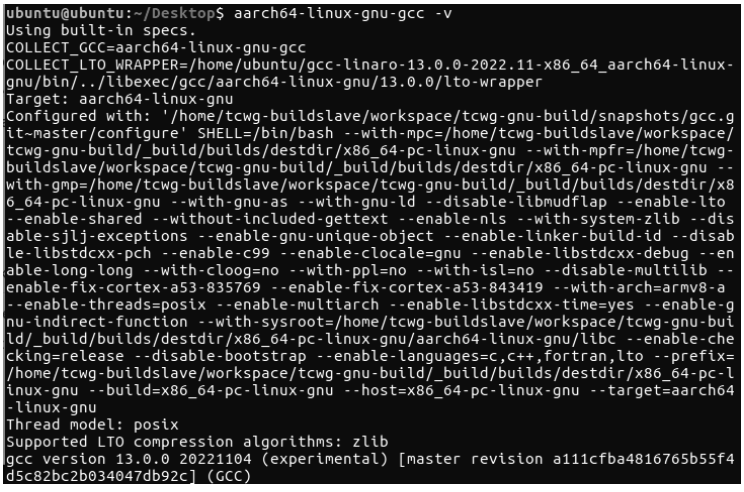

图 1-7

步骤 05　执行“apt install gdb-multiarch”命令，安装gdb-multiarch调试工具，再编写如下代码：

```
#include<stdio.h>
int main(int argc, char* argv[])
{
    printf("hello");
    return 0;
}
```

步骤 06　执行“aarch64-linux-gnu-gcc test.c -o test”命令编译程序，执行“gdb-multiarch test”命令调试程序，执行“disassemble main”命令查看main函数的汇编代码，结果如图1-8所示。由图可知，main函数的汇编代码采用ARM指令集，说明使用交叉编译工具和gdb-multiarch可以编译、调试ARM程序。

```
pwndbg> disassemble main
Dump of assembler code for function main:
   0x000000000040055c <+0>:     stp     x29, x30, [sp,#-32]!
   0x0000000000400560 <+4>:     mov     x29, sp
   0x0000000000400564 <+8>:     str     w0, [x29,#28]
   0x0000000000400568 <+12>:    str     x1, [x29,#16]
   0x000000000040056c <+16>:    adrp    x0, 0x400000
   0x0000000000400570 <+20>:    add     x0, x0, #0x638
   0x0000000000400574 <+24>:    bl      0x400450 <printf@plt>
   0x0000000000400578 <+28>:    mov     w0, #0x0
   0x000000000040057c <+32>:    ldp     x29, x30, [sp],#32
   0x0000000000400580 <+36>:    ret
End of assembler dump.
```

图 1-8

1.2.3　MIPS环境

在Linux中，主要使用交叉编译工具编译MIPS程序，使用gdb-multiarch调试程序。下面通过案例演示交叉编译工具和gdb-multiarch的安装过程。

步骤 01　执行“sudo apt install gcc-mips-linux-gnu”命令，安装交叉编译工具，安装完成后执行“mips-linux-gnu-gcc -v”命令，结果如图1-9所示。由图可知，gdb-multiarch安装成功。

```
ubuntu@ubuntu:~/Desktop$ mips-linux-gnu-gcc -v
Using built-in specs.
COLLECT_GCC=mips-linux-gnu-gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc-cross/mips-linux-gnu/5/lto-wrapper
Target: mips-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 5.4.0-6ubuntu
1~16.04.9' --with-bugurl=file:///usr/share/doc/gcc-5/README.Bugs --enable-la
nguages=c,ada,c++,java,go,d,fortran,objc,obj-c++ --prefix=/usr --program-suf
fix=-5 --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --with
out-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls -
-with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-libst
dcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --
disable-libitm --disable-libsanitizer --disable-libquadmath --enable-plugin
--with-system-zlib --disable-browser-plugin --enable-java-awt=gtk --enable-g
tk-cairo --with-java-home=/usr/lib/jvm/java-1.5.0-gcj-5-mips-cross/jre --ena
ble-java-home --with-jvm-root-dir=/usr/lib/jvm/java-1.5.0-gcj-5-mips-cross -
-with-jvm-jar-dir=/usr/lib/jvm-exports/java-1.5.0-gcj-5-mips-cross --with-ar
ch-directory=mips --with-ecj-jar=/usr/share/java/eclipse-ecj.jar --disable-l
ibgcj --enable-multiarch --disable-werror --enable-multilib --with-arch-32=m
ips32r2 --with-fp-32=xx --enable-targets=all --with-arch-64=mips64r2 --enabl
e-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target
=mips-linux-gnu --program-prefix=mips-linux-gnu- --includedir=/usr/mips-linu
x-gnu/include
Thread model: posix
gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.9)
```

图 1-9

步骤02 编写如下代码：

```
#include<stdio.h>
int main(int argc, char* argv[])
{
    printf("hello");
    return 0;
}
```

步骤03 执行“mips-linux-gnu-gcc test.c -o test”命令编译程序，执行“gdb-multiarch test”命令调试程序，执行“disassemble main”命令查看main函数的汇编代码，结果如图1-10所示。由图可知，main函数的汇编代码采用MIPS指令集，说明使用交叉编译工具和gdb-multiarch可以编译、调试MIPS程序。

```
pwndbg> disassemble main
Dump of assembler code for function main:
   0x004007b0 <+0>:     addiu   sp,sp,-32
   0x004007b4 <+4>:     sw      ra,28(sp)
   0x004007b8 <+8>:     sw      s8,24(sp)
   0x004007bc <+12>:    move    s8,sp
   0x004007c0 <+16>:    lui     gp,0x42
   0x004007c4 <+20>:    addiu   gp,gp,-28656
   0x004007c8 <+24>:    sw      gp,16(sp)
   0x004007cc <+28>:    sw      a0,32(s8)
   0x004007d0 <+32>:    sw      a1,36(s8)
   0x004007d4 <+36>:    lui     v0,0x40
   0x004007d8 <+40>:    addiu   a0,v0,2480
   0x004007dc <+44>:    lw      v0,-32692(gp)
   0x004007e0 <+48>:    move    t9,v0
   0x004007e4 <+52>:    jalr    t9
   0x004007e8 <+56>:    nop
   0x004007ec <+60>:    lw      gp,16(s8)
   0x004007f0 <+64>:    move    v0,zero
   0x004007f4 <+68>:    move    sp,s8
   0x004007f8 <+72>:    lw      ra,28(sp)
   0x004007fc <+76>:    lw      s8,24(sp)
   0x00400800 <+80>:    addiu   sp,sp,32
   0x00400804 <+84>:    jr      ra
   0x00400808 <+88>:    nop
End of assembler dump.
```

图 1-10

1.3 常 用 工 具

1.3.1 PE工具

PE（Portable Executable File Format，可移植的执行体文件格式）是Windows系统下的文件格式，常见的文件扩展名有exe、dll、sys等。PE工具主要用来查看、修改PE文件结构，也可以用来识别壳、编辑资源、导入表修复等。常见的PE工具有PEiD、LordPE、Exeinfo PE等。

1. PEiD

PEiD是一款PE文件识别、查壳工具，包含3种扫描模式：

- 正常扫描模式：在PE文档的入口点扫描所有记录的签名。
- 深度扫描模式：深入扫描所有记录的签名，扫描范围更广、更深入。
- 核心扫描模式：完整地扫描整个PE文档。

PEiD包含多个模块：

- 任务查看模块：扫描并查看当前正在运行的所有任务和模块，并可终止其运行。
- 多文件扫描模块：同时扫描多个文档。
- Hex十六进制查看模块：以十六进制方式快速查看文档。

PEiD的主界面如图1-11所示。

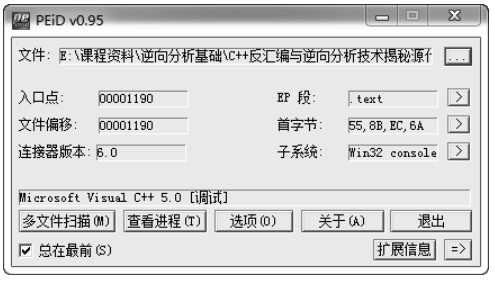

图 1-11

2. LordPE

LordPE是一款功能强大的PE文件分析、修改、脱壳工具，集合了进程转存、PE文件重建、PE文件编辑等功能。LordPE的主界面如图1-12所示。

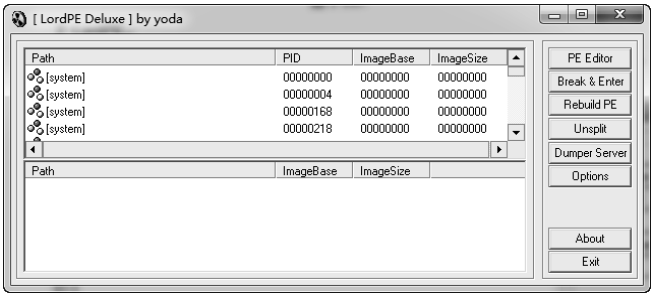

图 1-12

3. Exeinfo PE

Exeinfo PE是一款程序查壳工具，既可以查看加密程序的PE信息和编译信息，也可以查看是否加壳、输入输出表、入口地址等信息。Exeinfo PE的主界面如图1-13所示。

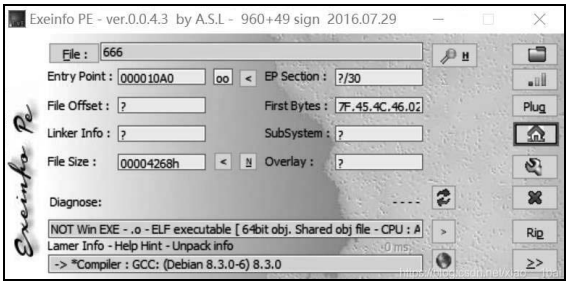

图 1-13

1.3.2　OllyDbg工具

OllyDbg简称OD，是Windows平台下Ring3级的、具有可视化界面的32位反汇编工具，可以反汇编并动态调试x86二进制文件，支持插件扩展，通常用于软件逆向分析和漏洞挖掘。下面介绍OD常用的窗口和快捷键。

1. 窗口

OD的默认主界面如图1-14所示，窗口由5个子窗口组成：反汇编窗口、信息窗口、寄存器窗口、数据窗口和堆栈窗口。

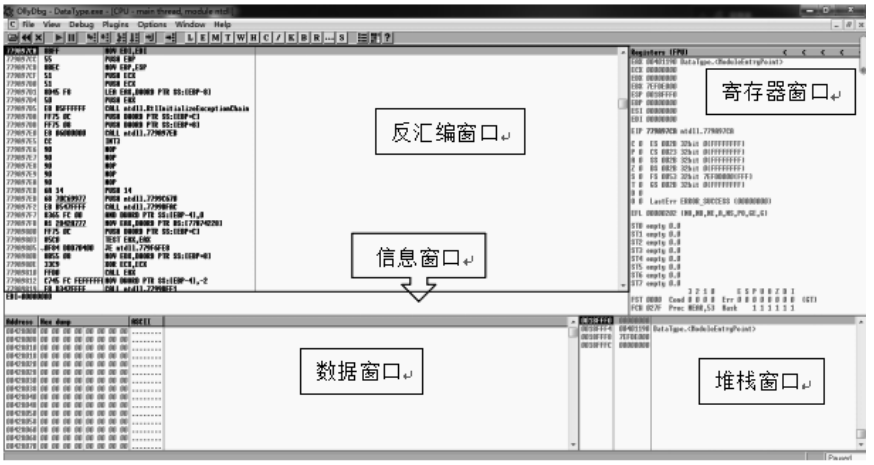

图 1-14

- 反汇编窗口：显示被调试程序的反汇编代码，标题栏共4列，分别是地址、HEX数据、反汇编和注释，具有搜索、分析、查找、修改等与反汇编操作相关的功能。
- 信息窗口：解释反汇编窗口中选中的命令，包括跳转目标地址、字符串、当前寄存器的值等。
- 寄存器窗口：显示当前所选线程的CPU寄存器数据信息，单击寄存器（FPU）标签可以切换寄存器的显示方式。
- 数据窗口：显示内存或文件的数据，可以通过CTRL+G快捷键输入内存地址，查看指定地址的数据。
- 堆栈窗口：显示当前线程的堆栈，窗口分为3列，分别是地址、数值和注释。

OD的其他窗口包括：

- L：显示日志。
- E：显示运行程序所使用的模块。
- M：显示程序映射到内存的信息。
- T：显示程序的线程。
- W：Windows显示程序。
- H：句柄窗口。

- C：回到CPU窗口。
- P：显示程序修改的信息。
- K：显示调用堆栈的信息。
- B：显示程序断点的列表。
- R：显示在软件中的搜索结果。
- …：显示RUN TRACE命令的执行结果。

2. 常用快捷键

OD的快捷键相当丰富，常用的快捷键如下：

（1）F2：在选中的汇编指令上设置或取消断点。
（2）F7：在调试程序时，跟进到子函数内部。
（3）F8：在调试程序时，直接运行完子函数。
（4）F9：在遇到断点时，单击F9键继续运行程序。
（5）Shift+F7/F8/F9：忽略异常，继续运行程序。
（6）Ctrl+F2：重新载入调试程序。
（7）Alt+F2：关闭调试程序。
（8）空格键：修改汇编代码，双击可以实现同样的功能。
（9）Alt+B/C/E/K/L/M：显示断点窗口/CPU窗口/模块窗口/调用栈窗口/日志窗口/内存窗口。
（10）Ctrl+B：打开搜索窗口。
（11）Ctrl+E：编辑所选内容。

1.3.3　IDA Pro工具

IDA Pro是一款交互式、可编程、可扩展的多处理器反汇编工具，简称IDA，支持数十种CPU指令集，包括Intel x86、x64、MIPS、ARM等，它是目前使用广泛的静态反编译软件。

IDA的安装根目录下包含多个文件夹，分别存储不同的内容：

- cfg: 存放各种配置文件（ida.cfg为基本的IDA配置文件，idagui.cfg为GUI配置文件，idatui.cfg为文本模式用户界面配置文件）。
- dbgsrv: 存放用于调试的服务器端软件。
- idc: 存放IDA的内置脚本语言IDC所需要的核心文件。
- ids: 存放一些符号文件。
- loaders: 存放用于识别和解析PE或者ELF的文件。
- plugins: 存放附加的插件模块。
- procs: 存放处理器模块相关文件。

1. 窗口

IDA的主界面如图1-15所示，窗口主要由3个子窗口组成，分别是反汇编主窗口、函数窗口和输出窗口。

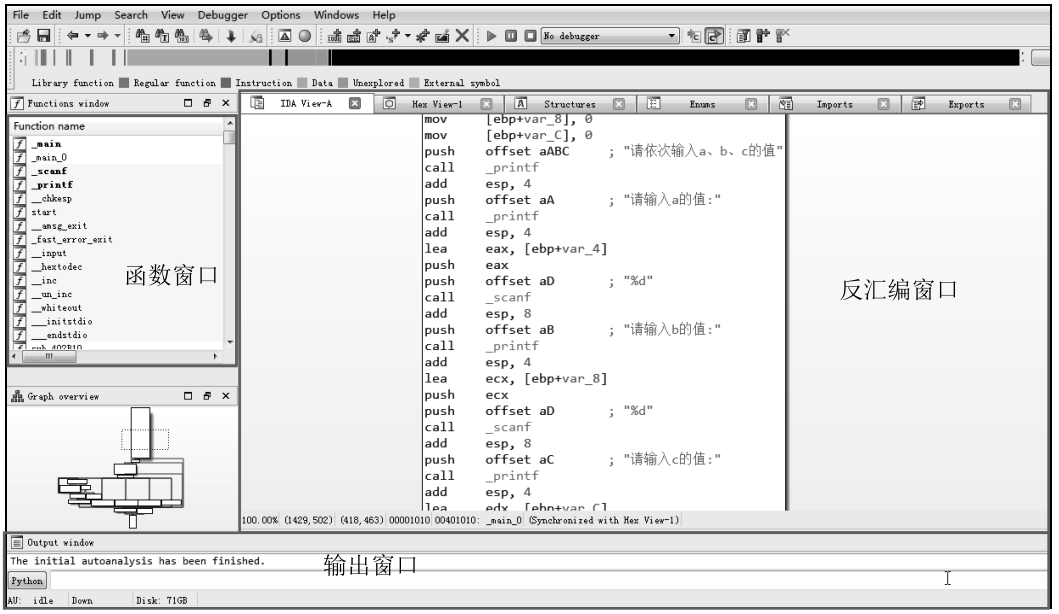

图 1-15

- 反汇编主窗口：也叫IDA-View窗口，显示反汇编的结果、控制流图等，是操作和分析二进制文件的主要窗口，其上还包括Enums（枚举窗口）、Structures（结构体窗口）、Imports（导入函数窗口）、Exports（导出函数窗口）等子窗口。

IDA图形视图将一个函数分解成多个不包含分支的最大指令序列块，以生动显示该函数的控制流程，并使用不同的彩色箭头区分函数块之间的控制流：Yes控制流的箭头为绿色，No控制流的箭头为红色（颜色参见下载资源中的相关文件）。窗口如图1-16所示。

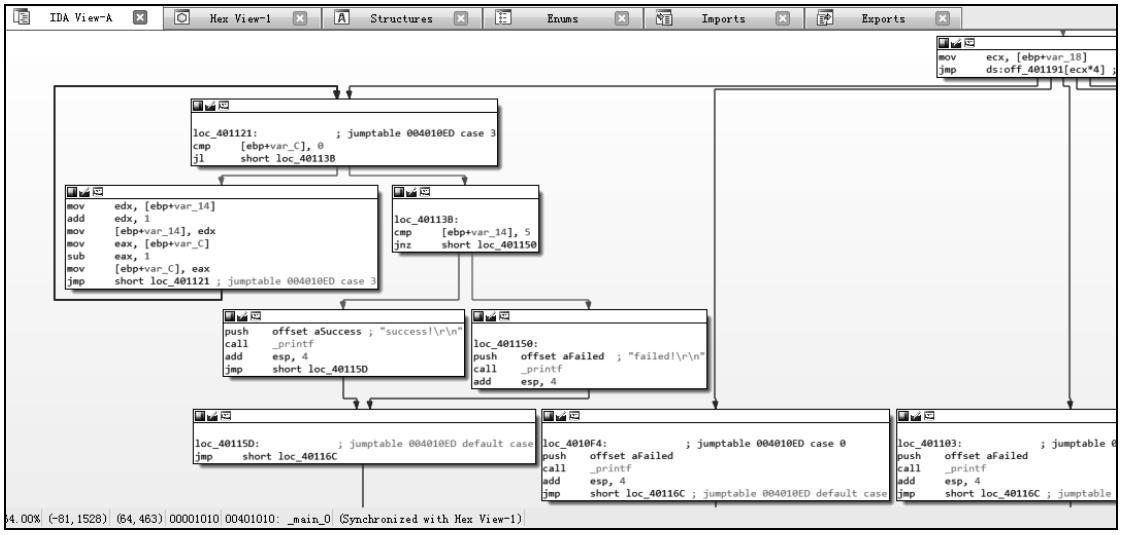

图 1-16

- 函数窗口：列举IDA利用内置数据库识别的函数，如图1-17所示。

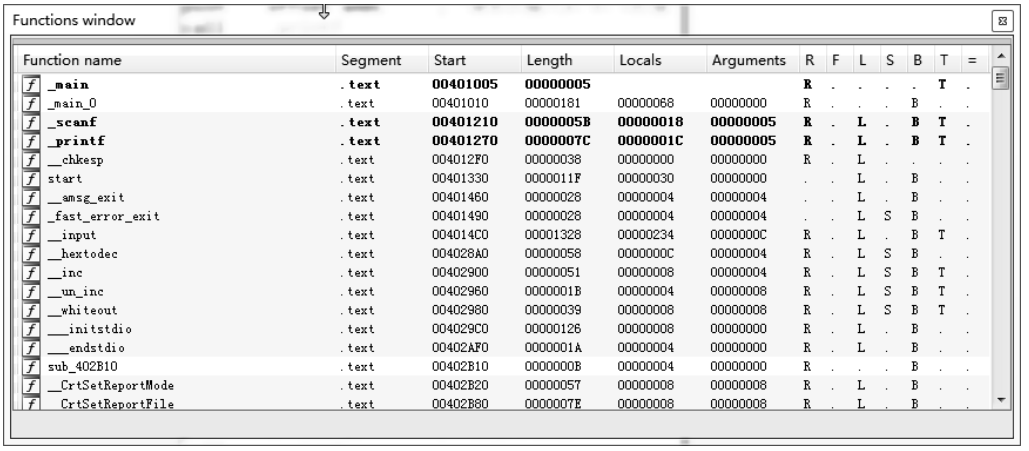

图 1-17

- 输出窗口：显示调试时的信息。

2. 常用快捷键

IDA中的快捷键相当丰富，常用的快捷键如下：

- A：将数据转换为字符串。
- F5：一键反汇编。
- Esc：回退键，倒回上一步操作的视图。
- Shift + F12：打开string窗口，找出所有的字符串。
- Ctrl + 鼠标滚轮：调节流程视图大小。
- X：查看交叉引用，选中某个函数或变量，按该快捷键即可查看。
- G：直接跳转到某个地址。
- N：更改变量的名称。
- Y：更改变量的类型。
- /：添加反汇编代码注释。
- \：隐藏或显示变量和函数的类型描述。
- 分号键：在反汇编界面中添加注释。

1.4　本 章 小 结

本章介绍了二进制安全的几个基本概念、常见编译环境及常用工具，主要内容包括x86、ARM和MIPS指令集及常见汇编指令；在Linux中，x86、ARM和MIPS编译环境的搭建；PE、OD、IDA工具的基本功能。通过本章的学习，读者能够了解二进制安全的基本概念，初步掌握几种不同指令集、编译环境和工具的安装。

1.5 习　题

1. 执行下列指令：

```
STR1  DW  'AB'
STR2  DB  16 DUP(?)
CNT  EQU  $ - STR1
MOV  CX, CNT
MOV  AX, STR1
HLT
```

寄存器CL的值是___（1）___，寄存器AX的值是___（2）___。

（1）　A. 10H　　B. 12H　　C. 0EH　　D. 0FH

（2）　A. 00ABH　　B. 00BAH　　C. 4142H　　D. 4241H

2. 执行下列指令：

```
MOV  AX, 1234H
MOV  CL, 4
ROL  AX, CL
DEC  AX
MOV  CX, 4
MUL  CX
HLT
```

寄存器AH的值是（　）。

A. 92H　　B. 8CH　　C. 8DH　　D. 00H

3. 下列指令中，不影响标志位的指令是（　）。

A. ROR　AL, 1　　B. JNC　Label　　C. INT　n　　D. SUB　AX, BX

4. x86寄存器有哪些？

5. ARM寄存器有哪些？

6. MIPS寄存器有哪些？

第 2 章
基本数据类型

2.1 整　　数

在C语言中，整数类型有3种：short、int和long。short在内存中占2字节，int和long占4字节。为了便于阅读，内存中的数据均采用十六进制表示。

整数类型分为两种：无符号型和有符号型。无符号型只能用来表示正数，有符号型用来表示正数和负数。

2.1.1 无符号整数

无符号整数的所有位都用来表示数值。以无符号int为例，其在内存中占4字节，取值范围为0x00000000～0xFFFFFFFF，当无符号整数不足32位时，用0来填充剩余的高位。例如，数值16对应的二进制为1000，只占4位，剩余28个高位用0填充，对应的十六进制数为0x00000010。在内存中，如果以“小尾方式”存放，小尾存放遵循低位数据存放在内存低端，高位数据存放在内存高端的方式，则数值16内存中存放为10 00 00 00；如果以“大尾方式”存放，大尾存放与小尾存放相反，则数值16在内存中存放为00 00 00 10。下面通过案例观察与无符号整数相关的汇编代码。

步骤01 编写C语言代码，将文件保存并命名为“uint.c”，代码如下：

```
int main(int argc, char* argv[])
{
    unsigned int uInt = 16;
    return 0;
}
```

步骤02 执行“gcc -m32 uint.c -o uint”命令，编译程序。

步骤03 执行“gdb uint”命令调试程序，结果如图2-1所示。

```
ubuntu@ubuntu:~/Desktop/jiaocai/DataType$ gdb uInt
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
pwndbg: loaded 195 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from uInt...(no debugging symbols found)...done.
pwndbg>
```

图 2-1

步骤04　执行start命令，程序自动暂停在main函数的第一行汇编代码处，如图2-2所示。

```
[ DISASM ]
► 0x80483e1  <main+6>                  mov    dword ptr [ebp - 4], 0x10
  0x80483e8  <main+13>                 mov    eax, 0
  0x80483ed  <main+18>                 leave
  0x80483ee  <main+19>                 ret
   ↓
  0xf7e1a647 <__libc_start_main+247>   add    esp, 0x10
  0xf7e1a64a <__libc_start_main+250>   sub    esp, 0xc
  0xf7e1a64d <__libc_start_main+253>   push   eax
  0xf7e1a64e <__libc_start_main+254>   call   exit                    <exit>

  0xf7e1a653 <__libc_start_main+259>   xor    ecx, ecx
  0xf7e1a655 <__libc_start_main+261>   jmp    __libc_start_main+50
n+50>

  0xf7e1a65a <__libc_start_main+266>   mov    esi, dword ptr [esp + 8]
```

图 2-2

步骤05　由图2-2可知，EIP指向的代码为“unsigned int uInt = 16;”对应的汇编代码，且0x10保存在[ebp-4]中。执行“n”命令，执行下一条指令，再执行“x $ebp-4”命令，查看[ebp-4]的真实地址值，结果如图2-3所示。由图可知，[ebp-4]的真实值为0xffffd034。

```
pwndbg> x $ebp-4
0xffffd034:     0x10
pwndbg>
```

图 2-3

步骤06　执行“x/32b 0xffffd034”命令，查看0xffffd034地址存储的数据，结果如图2-4所示。由图可知，数值16在内存中存储为10 00 00 00。

```
pwndbg> x/4b 0xffffd034
0xffffd034:     0x10    0x00    0x00    0x00
```

图 2-4

步骤07　在Visual Studio环境中，查看C代码对应的汇编代码，结果如图2-5所示。由图可知，Visual Studio和gcc在处理无符号整数时，方法是一致的。

```
    unsigned int uInt = 16;
00B21775  mov         dword ptr [uInt],10h
```

图 2-5

2.1.2　有符号整数

有符号整数用最高位表示符号，最高位为0表示正数，为1表示负数。以有符号int为例，由于最高位为符号位，因此有符号int类型正数取值范围为0x00000000～0x7FFFFFFF，负数取值范围为0x80000000～0xFFFFFFFF。正数在内存中以原码形式存放。负数在内存中以补码形式存放，补码的规则就是原码取反后加1。例如，数值−16的原码为0000 0000 0000 0000 0000 0000 0001 0000，反码为1111 1111 1111 1111 1111 1111 1110 1111，加1后为1111 1111 1111 1111 1111 1111 0000，即0xFFFFFFF0。下面通过案例观察与有符号整数相关的汇编代码。

步骤 01　编写C语言代码：

```
int i = -16;
```

步骤 02　编译程序，并用gdb调试程序，再查看汇编代码，结果如图2-6所示。由图可知，−16在内存中存储为0xFFFFFFF0。

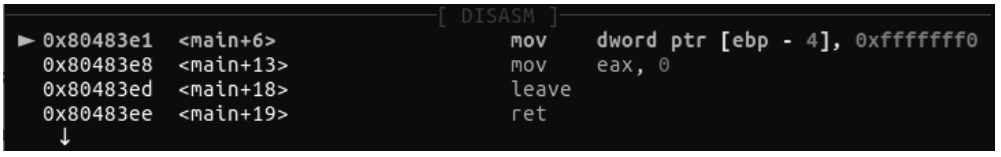

图 2-6

汇编指令操作的数据是有符号数还是无符号数，需要结合与上下文相关联的其他指令来综合判断。例如，数据作为参数被传递到某个函数，而该函数的参数确定为有符号数，则可以断定该数据为有符号数。

步骤 03　在Visual Studio环境中，查看C语言代码对应的汇编代码，结果如图2-7所示。由图可知，Visual Studio和gcc在处理有符号整数时，方法是一致的。

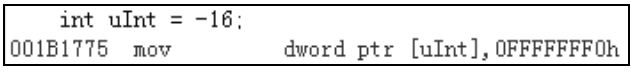

图 2-7

2.2　浮　点　数

用“定点数”存储小数，存在数值范围和精度范围有限的缺点，所以在计算机中，一般使用“浮点数”来存储小数。浮点数有两种类型：float（单精度）和double（双精度）。float类型在内存中占4字节，double类型在内存中占8字节。浮点数类型与整数类型一样，以十六进制的方式在内存中存储。整数类型是将十进制数直接转换为十六进制进行存储，而浮点数类型是先将十进制浮点小数转换为对应的二进制数，再进行编码、存储。

浮点数的操作不使用通用寄存器，而是使用专门用于浮点数计算的浮点寄存器，且在使用之前，需要对浮点寄存器进行初始化。

2.2.1　浮点指令

浮点寄存器共有8个，记作st(0)～st(7)，每个浮点寄存器占8字节。在使用浮点寄存器时，必须按从st(0)到st(7)的顺序依次使用，使用浮点寄存器的方法就是压栈和出栈。常用的浮点指令如表2-1所示。

表2-1　常用的浮点指令

指令名称	使用格式	指令功能
fld	fld src	将浮点数src压入栈中
fild	fild src	将整数src压入st(0)
fadd	fadd	将st(0)和st(1)出栈，并相加，再将它们的和入栈
	fadd src	st(0)与src相加，结果存放在st(0)
fst	fst dst	取浮点数st(0)到dst，不影响栈状态
fist	fist dst	取整数s(0)到dst，不影响栈状态
fstp	fstp dst	取浮点数st(0)到dst，执行出栈操作
fistp	fistp dst	取整数st(0)到dst，执行出栈操作
fcom	fcom src	st(0)与浮点数src比较，影响标志位
ficom	ficom src	st(0)与整数src比较，影响标志位

说明：f：float；i：integer；ld：load；st:store；p：pop；com：compare。

2.2.2　编码

浮点数的编码转换采用的是IEEE规定的编码标准。float和double类型的数据转换原理相同，但是由于它们的范围不一样，编码方式略有区别。

float类型占4字节，即32位，最高位用于表示符号，中间8位用于表示指数，最后23位用于表示尾数，即表示为“S E*08 M*23”，其中S表示符号位（0为正、1为负），E表示指数位，M表示尾数。

double类型占8字节，即64位，最高位用于表示符号，中间11位用于表示指数，最后52位用于表示尾数，即表示为“S E*11 M*52”。

在进行二进制转换时，需对浮点型数据进行科学记数法转换。例如，将10.5f进行IEEE编码，先将10.5f转换为对应的二进制数，结果为1010.1，其中，整数部分为1010，小数部分为1；再将1010.1转换为整数部分只有1位的小数，即1.0101，指数为3；然后进行编码，符号位为0；指数位为十进制的3 + 127，转换为二进制的1000 0010；尾数位为0101 0000 0000 0000 0000 000（不足23位，低位用0补齐）。

指数位加127的原因是指数可能为负数，十进制127可表示为二进制01111111。IEEE编码方式规定，当指数域小于0111 1111时为一个负数，反之为一个正数。

将10.5f的IEEE编码按二进制拼接，结果为0 1000 0010 0101 0000 0000 0000 0000 000，转换为十六进制数，结果为0x41280000，在内存中按小端方式排列，为00 00 28 41。

如果小数部分转换为二进制数时是一个无穷值，则会根据尾数长度舍弃多余的部分。例如

10.3，先转换为二进制小数，为1010.0100 1100 1100 1100 1101；再转换为整数部分只有1位的小数，为1.0100 1001 1001 1001 1001 101，尾数部分为23位，指数为3，编码后为0 1000 0010 0100 1001 1001 1001 1001 101；最后转换为十六进制数，为0x4124cccd，在内存中按小端方式排列，为cd cc 28 41。

double类型的IEEE编码转换过程与float类型一样，不同的是指数位加1023。下面通过案例观察与浮点型数据相关的汇编代码。

步骤01　编写C语言代码，将文件保存并命名为“f.c”，代码如下：

```
#include<stdio.h>
int main(int argc, char* argv[])
{
    float f1 = 10.5f;
    float f2 = 10.3f;
    f1 = f1 + 1.0f;
    return 0;
}
```

步骤02　先执行“gcc -m32 -g f.c -o f”命令编译程序，再使用gdb调试程序，执行“disassemble main”命令查看main函数的汇编代码，结果如图2-8所示。

```
0x080483e1 <+6>:     fld    DWORD PTR ds:0x8048490
0x080483e7 <+12>:    fstp   DWORD PTR [ebp-0x8]
0x080483ea <+15>:    fld    DWORD PTR ds:0x8048494
0x080483f0 <+21>:    fstp   DWORD PTR [ebp-0x4]
0x080483f3 <+24>:    fld    DWORD PTR [ebp-0x8]
0x080483f6 <+27>:    fld1
0x080483f8 <+29>:    faddp  st(1),st
0x080483fa <+31>:    fstp   DWORD PTR [ebp-0x8]
```

图 2-8

由图2-8可知，代码“float f1 = 10.5f;”对应的汇编代码为：

```
0x080483e1 <+6>:fld dword ptr ds:0x8048490
0x080483e7 <+12>:fstp    dword ptr [ebp-0x8]
```

执行“x/4ab 0x8048490”命令，查看0x8048490地址存储的数据，结果如图2-9所示。由图可知，10.5f在内存中存储为00 00 28 41，与前文分析一致。

```
pwndbg> x/4ab 0x8048490
0x8048490:      0x0     0x0     0x28    0x41
```

图 2-9

代码“f1 = f1 + 1.0f;”对应的汇编代码为：

```
0x080483f3 <+24>:flddword ptr [ebp-0x8]
0x080483f6 <+27>:fld1
0x080483f8 <+29>:faddp   st(1), st
0x080483fa <+31>:fstp    dword ptr [ebp-0x8]
```

步骤03　在Visual Studio环境中，查看C代码对应的汇编代码，结果如图2-10所示。由图可知，Visual Studio和gcc在处理浮点型数据时，所用指令和寄存器均不一致，Visual Studio编译器用于浮点运算的指令为movss、addss等，用于浮点运算的寄存器为xmm0～xmm7。

```
    float f1 = 10.5f;
00951775  movss       xmm0,dword ptr [__real@41280000 (0957B38h)]
0095177D  movss       dword ptr [f1],xmm0
    float f2 = 10.3f;
00951782  movss       xmm0,dword ptr [__real@4124cccd (0957B34h)]
0095178A  movss       dword ptr [f2],xmm0
    f1 = f1 + 1.0f;
0095178F  movss       xmm0,dword ptr [f1]
00951794  addss       xmm0,dword ptr [__real@3f800000 (0957B30h)]
0095179C  movss       dword ptr [f1],xmm0
```

图 2-10

2.3　字符和字符串

字符主要包括数字、字母、控制、通信等符号，编码方式分为两种：ASCII和Unicode。ASCII编码占用1字节，表示范围为0～128，使用7位二进制数（剩下的1位二进制为0）来表示所有的大小写字母、数字0～9、标点符号和特殊控制字符。Unicode是一个全球统一的字符编码标准，可以用1～4字节来表示世界上几乎所有的文字，其表示范围达到0x10FFFF。

在C语言中，使用char定义ASCII编码格式的字符，使用wchar_t定义Unicode编码格式的字符。

字符串是由数字、字母和下画线组成的一串字符，字符串在存储上类似字符数组，在存储方式上分为两种方法：一种是在首地址的4字节中保存字符串总长度；另一种是在字符串的结尾处使用结束符，在C语言中，使用“\0”为结束符。

源代码文件必须为UTF-8编码格式且不能有BOM标志头，gcc才能正确支持wchar_t字符和字符串。在Windows平台，宽字符是16位的UTF-16类型，在Linux平台，宽字符是32位的UTF-32类型。

gcc可以使用“-finput-charset = charset”或“-fwide-exec-charset = charset”来改变宽字符串的类型，常见的字符集如下：

- ANSI体系：ASCII字符集、GB2312字符集和GBK字符集。
- Unicode体系：Unicode字符集。

常见编码如下：

- ASCII字符集：ASCII编码。
- GB2312字符集：GB2312编码。
- GBK字符集：GBK编码。
- Unicode字符集：UTF-8编码、UTF-16编码和UTF-32编码。

下面通过案例来观察字符和字符串的存储方式。

步骤 01　编写C语言代码，将文件保存并命名为“char.c”，代码如下：

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

```
#include<wchar.h>
int main(int argc, char* argv[])
{
    char c = 'a';
    wchar_t wc = L'a';
    char* s = "abc";
    wchar_t* sw = L"abc";
    char* pc = "二进制安全";
    wchar_t* pw = L"二进制安全";
    return 0;
}
```

步骤02 先执行“gcc -m32 -g char.c -o char”命令编译程序，再使用gdb调试程序，执行“disassemble main”命令查看main函数的汇编代码，结果如图2-11所示。

```
0x080483e1 <+6>:     mov    BYTE PTR [ebp-0x15],0x61
0x080483e5 <+10>:    mov    DWORD PTR [ebp-0x14],0x61
0x080483ec <+17>:    mov    DWORD PTR [ebp-0x10],0x8048490
0x080483f3 <+24>:    mov    DWORD PTR [ebp-0xc],0x8048494
0x080483fa <+31>:    mov    DWORD PTR [ebp-0x8],0x80484a8
0x08048401 <+38>:    mov    DWORD PTR [ebp-0x4],0x80484b8
```

图 2-11

由图2-11可知，代码“char c = 'a'; wchar_t wc = L'a';”对应的汇编代码为：

```
0x80483e1 <main+6>:mov  byte ptr [ebp-0x15], 0x61
0x80483e5 <main+10>:mov  dword ptr [ebp-0x14], 0x61
```

由代码可知，在Linux中，char字符占1字节，wchar_t字符占4字节。

代码“char* s = "abc"; wchar_t* sw = L"abc";”对应的汇编代码为：

```
0x080483ec <+17>:movdword ptr [ebp-0x10], 0x8048490
0x080483f3 <+24>:movdword ptr [ebp-0xc], 0x8048494
```

执行“x/4ub 0x8048490”命令，查看0x8048490地址存储的数据，结果如图2-12所示。

```
pwndbg> x/4ub 0x8048490
0x8048490:      97      98      99      0
```

图 2-12

执行“x/16ub 0x8048494”命令，查看0x8048494地址存储的数据，结果如图2-13所示。

```
pwndbg> x/16ub 0x8048494
0x8048494:      97      0       0       0       98      0       0       0
0x804849c:      99      0       0       0       0       0       0       0
```

图 2-13

由此可知，在Linux中，char字符串中字符占1字节，wchar_t字符串中字符占4字节。

代码“char* pc = "二进制安全"; wchar_t* pw = L"二进制安全";”对应的汇编代码为：

```
0x080483fa <+31>:movdword ptr [ebp-0x8], 0x80484a4
0x08048401 <+38>:movdword ptr [ebp-0x4], 0x80484b4
```

执行“x/4ab 0x80484a4”和“x/4ab 0x80484b4”命令，查看0x80484a4和0x80484b4地址存储的数据，即char和wchar_t汉字字符串存储方式，结果如图2-14所示。

```
pwndbg> x/4ab 0x80484a4
0x80484a4:      0xfffffffe      0xfffffffba      0xffffff8c      0xfffffffe8
pwndbg> x/4ab 0x80484b4
0x80484b4:      0xffffff8c      0x4e    0x0     0x0
```

图 2-14

再查询汉字“二”对应的编码，查询结果如图2-15所示。

图 2-15

由此可知，在Linux中，char字符串中字符采用UTF-8编码方式，wchar_t字符串中字符采用UTF-32编码方式。

步骤03 在Visual Studio环境中，查看C代码对应的汇编代码，结果如图2-16所示。由图可知，Visual Studio和gcc在处理字符和字符串时，方法一致。

```
    char c = 'a';
00FE43B5  mov         byte ptr [c],61h
    wchar_t wc = L'a';
00FE43B9  mov         eax,61h
00FE43BE  mov         word ptr [wc],ax
    char* s = "abc";
00FE43C2  mov         dword ptr [s],offset string "abc" (0FE7BE0h)
    wchar_t* sw = L"abc";
00FE43C9  mov         dword ptr [sw],offset string "abc" (0FE7BD8h)
    char* pc = "二进制安全";
00FE43D0  mov         dword ptr [pc],offset string "%d" (0FE7B30h)
    wchar_t* pw = L"二进制安全";
00FE43D7  mov         dword ptr [pw],offset string "argc=3 \r\n" (0FE7BE4h)
```

图 2-16

查看0x0FE7BE0地址存储的数据，结果如图2-17所示。查看0x0FE7BD8地址存储的数据，结果如图2-18所示。

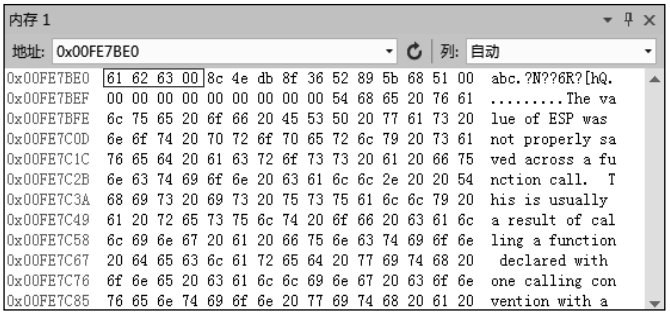

图 2-17

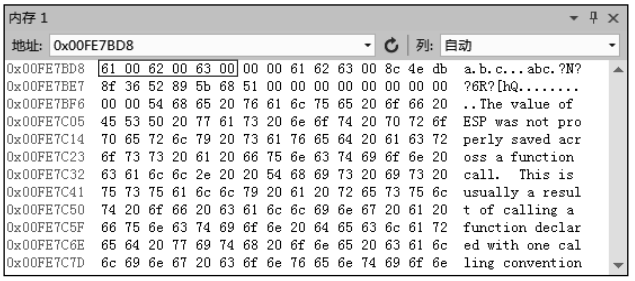

图 2-18

由图可知，在Visual Studio编译器下，char字符串中字符占1字节，wchar_t字符串中字符占2字节。

查看0x0FE7B30地址存储的数据，结果如图2-19所示。查看0x0FE7BE4地址存储的数据，结果如图2-20所示。

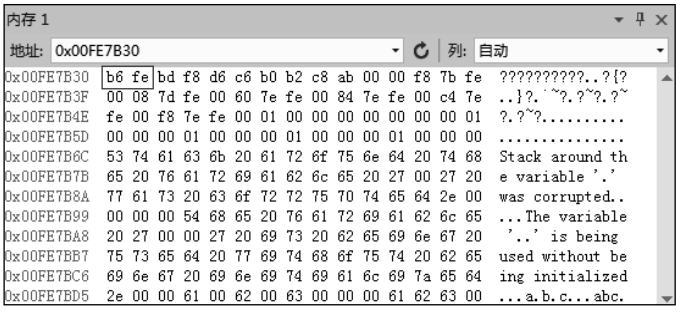

图 2-19

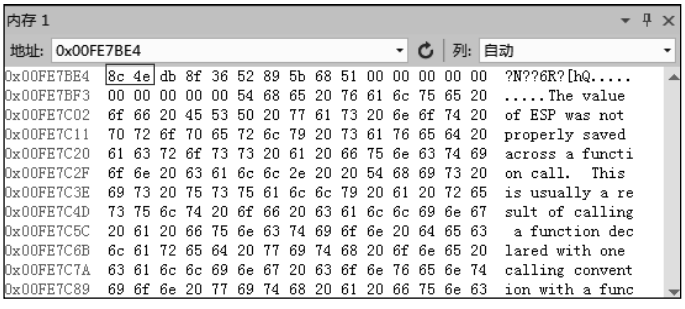

图 2-20

由图可知，在Visual Studio编译器下，char字符串中字符采用国标码，wchar_t字符串中字符采用UTF-16BE编码。

2.4　布　尔　型

C语言没有定义布尔类型，判断真假时以0为假，非0为真。布尔类型在内存中占1字节。下面通过案例来观察与布尔型数据相关的汇编代码。

步骤01 编写C语言代码，将文件保存并命名为“bool.c”，代码如下：

```
#include<stdio.h>
#include<stdbool.h>
int main(int argc, char* argv[])
{
    bool b1 = true;
    bool b2 = false;
return 0;
}
```

步骤02 先执行“gcc -m32 -g bool.c -o bool”命令编译程序，再使用gdb调试程序，执行“disassemble main”命令查看main函数的汇编代码，核心代码如图2-21所示。

```
0x080483e1 <+6>:     mov    BYTE PTR [ebp-0x2],0x1
0x080483e5 <+10>:    mov    BYTE PTR [ebp-0x1],0x0
```

图 2-21

由图2-21可知，布尔型变量b1对应0x1，b2对应0x0。

步骤03 在Visual Studio环境中，查看C代码对应的汇编代码，结果如图2-22所示。由图可知，Visual Studio和gcc在处理布尔型数据时，方法一致。

```
    bool b1 = true;
01171775  mov         byte ptr [b1],1
    bool b2 = false;
01171779  mov         byte ptr [b2],0
```

图 2-22

2.5 指　　针

在C语言中，使用“&”符号取变量的地址，使用“TYPE *”定义指针。TYPE为数据类型，任何数据类型都有对应的指针类型，指针只保存数据的首地址，需要根据对应的类型来解析数据。同一地址，使用不同类型的指针进行访问，取出的数据各不相同。下面通过案例来观察不同的指针类型取出不同的数据。

步骤01 编写C语言代码，将文件保存并命名为“p1.c”，代码如下：

```
#include<stdio.h>
int main(int argc, char* argv[])
{
    int n = 0x12345678;
    int *pn = (int*)&n;
    char *pc = (char*)&n;
    short *ps = (short*)&n;
    printf("%08x \n", *pn);
    printf("%08x \n", *pc);
    printf("%08x \n", *ps);
```

```
    return 0;
}
```

步骤02　先执行“gcc -m32 -g p1.c -o p1”命令编译程序，再执行“./p1”命令运行程序，结果如图2-23所示。

```
ubuntu@ubuntu:~/Desktop/textbook/ch2$ ./p1
12345678
00000078
00005678
```

图 2-23

变量n在内存中存储为78 56 34 12，指针pn为int类型，占4字节，因此取出结果为12345678；指针pc为char类型，占1字节，因此取出结果为78；指针ps为short类型，占2字节，因此取出结果为5678。

指针通过加法或减法运算来实现地址偏移，地址的偏移量由指针类型决定。例如，指针类型为int，指针加1，则地址偏移4。下面通过案例进行观察。

步骤01　编写C语言代码，将文件保存并命名为“p2.c”，代码如下：

```
#include<stdio.h>
int main(int argc, char* argv[])
{
    int n = 0x12345678;
    int *pn = (int*)&n;
    char *pc = (char*)&n;
    short *ps = (short*)&n;
    pn++;
    pc++;
    ps++;
    return 0;
}
```

步骤02　先执行“gcc -m32 -g p2.c -o p2”命令编译程序，再使用gdb调试程序，执行“disassemble main”命令查看main函数的汇编代码，结果如图2-24所示。

```
0x08048457 <+28>:    mov    DWORD PTR [ebp-0x1c],0x12345678
0x0804845e <+35>:    lea    eax,[ebp-0x1c]
0x08048461 <+38>:    mov    DWORD PTR [ebp-0x18],eax
0x08048464 <+41>:    lea    eax,[ebp-0x1c]
0x08048467 <+44>:    mov    DWORD PTR [ebp-0x14],eax
0x0804846a <+47>:    lea    eax,[ebp-0x1c]
0x0804846d <+50>:    mov    DWORD PTR [ebp-0x10],eax
0x08048470 <+53>:    add    DWORD PTR [ebp-0x18],0x4
0x08048474 <+57>:    add    DWORD PTR [ebp-0x14],0x1
0x08048478 <+61>:    add    DWORD PTR [ebp-0x10],0x2
```

图 2-24

由图2-24可知，int、char、short类型的指针执行加1操作对应的汇编代码分别为：

```
0x08048470 <+53>:adddword ptr [ebp-0x18], 0x4
0x08048474 <+57>:adddword ptr [ebp-0x14], 0x1
0x08048478 <+61>:adddword ptr [ebp-0x10], 0x2
```

由代码可知，3种不同类型指针执行加1操作，地址分别偏移 4、1和2。因此，地址的偏移量由指针类型决定。

步骤03 在Visual Studio环境中，查看C代码对应的汇编代码，结果如图2-25所示。由图可知，Visual Studio和gcc在处理int、char和short类型指针的加1操作时，方法一致。

```
    int n = 0x12345678;
00BE43BF  mov         dword ptr [n],12345678h
    int* pn = (int*)&n;
00BE43C6  lea         eax,[n]
00BE43C9  mov         dword ptr [pn],eax
    char* pc = (char*)&n;
00BE43CC  lea         eax,[n]
00BE43CF  mov         dword ptr [pc],eax
    short* ps = (short*)&n;
00BE43D2  lea         eax,[n]
00BE43D5  mov         dword ptr [ps],eax
    pn++;
00BE43D8  mov         eax,dword ptr [pn]
00BE43DB  add         eax,4
00BE43DE  mov         dword ptr [pn],eax
    pc++;
00BE43E1  mov         eax,dword ptr [pc]
00BE43E4  add         eax,1
00BE43E7  mov         dword ptr [pc],eax
    ps++;
00BE43EA  mov         eax,dword ptr [ps]
00BE43ED  add         eax,2
00BE43F0  mov         dword ptr [ps],eax
```

图 2-25

2.6 常　　量

常量数据在程序运行前就已经存在，存放在可执行文件中。常量数据在C语言中有两种定义方式：一是使用#define预处理器，二是使用const关键字。#define定义的是真常量；const定义的是假常量，其本质仍然是变量，只是在编译器内进行检查，禁止修改，如果修改则报错。因此，const修饰的变量可以先利用指针获取变量地址，再通过指针修改变量地址的值，从而实现修改const修饰的变量的值。下面通过案例来观察常量的特征及相关的汇编代码。

步骤01 编写C语言代码，将文件保存并命名为“cst.c”，代码如下：

```
#include<stdio.h>
#define N 10
int main(int argc, char* argv[])
{
    const int a = 1;
    int* pn = (int*)&a;
    *pn = 10;
    printf("a=%d \n", a);
    printf("N=%d \n", N);
    return 0;
}
```

步骤02 先执行“gcc -m32 -g cst.c -o cst”命令编译程序，再使用gdb调试程序，执行“disassemble main”命令查看main函数的汇编代码，结果如图2-26所示。

```
0x08048487 <+28>:    mov    DWORD PTR [ebp-0x14],0x1
0x0804848e <+35>:    lea    eax,[ebp-0x14]
0x08048491 <+38>:    mov    DWORD PTR [ebp-0x10],eax
0x08048494 <+41>:    mov    eax,DWORD PTR [ebp-0x10]
0x08048497 <+44>:    mov    DWORD PTR [eax],0xa
0x0804849d <+50>:    mov    eax,DWORD PTR [ebp-0x14]
0x080484a0 <+53>:    sub    esp,0x8
0x080484a3 <+56>:    push   eax
0x080484a4 <+57>:    push   0x8048570
0x080484a9 <+62>:    call   0x8048330 <printf@plt>
0x080484ae <+67>:    add    esp,0x10
0x080484b1 <+70>:    sub    esp,0x8
0x080484b4 <+73>:    push   0xa
0x080484b6 <+75>:    push   0x8048576
0x080484bb <+80>:    call   0x8048330 <printf@plt>
```

图 2-26

由图2-26可知，#define定义的常量并未生成汇编代码；有或无const修饰的变量的赋值操作对应的汇编代码是一致的，通过指针可以修改const修饰的变量值。

步骤03 执行“./cst”命令，运行程序，结果如图2-27所示。

```
ubuntu@ubuntu:~/Desktop/textbook/ch2$ ./cst
a=10
N=10
```

图 2-27

由图2-27可知，const修饰的变量a的初始值为1，通过修改指针变为了10。

步骤04 在Visual Studio环境中，查看C代码对应的汇编代码，结果如图2-28所示。由图可知，Visual Studio和gcc在处理const修饰的变量时，方法一致。

```
    const int a = 1;
00E1487F  mov         dword ptr [a],1
    int* pn = (int*)&a;
00E14886  lea         eax,[a]
00E14889  mov         dword ptr [pn],eax
    *pn = 10;
00E1488C  mov         eax,dword ptr [pn]
00E1488F  mov         dword ptr [eax],0Ah
    printf("a=%d\n", a);
00E14895  push        1
00E14897  push        offset string "a=%d\n" (0E17B30h)
00E1489C  call        _main (0E1139Dh)
00E148A1  add         esp,8
    printf("N=%d\n", N);
00E148A4  push        0Ah
00E148A6  push        offset string "N=%d\n" (0E17BD8h)
00E148AB  call        _main (0E1139Dh)
00E148B0  add         esp,8
```

图 2-28

2.7 案　例

根据所给附件（可在本书配套提供的下载资源中获取），分析程序功能，并给出相应的C代码。附件中的源代码如下：

```
#include<stdio.h>
#define N 10
int main(int argc, char* argv[])
{
    int a = 100;
    a = a + 1;
    printf("%d \n", a);
    float f1 = 3.5f;
    f1 = f1 + 2;
    printf("%f \n", f1);
    char ch = 'a';
    char* pCh = "abc";
    printf("%c, %s \n", ch, pCh);
    const int b = 100;
    printf("%d \n", b);
    printf("%d \n", N);
    return 0;
}
```

步骤01　运行附件，结果如图2-29所示，程序输出了一系列的数值。

步骤02　使用IDA打开附件，核心代码如图2-30～图2-34所示。

```
ubuntu@ubuntu:~/Desktop/textbook/ch2$ ./2-1
101
5.500000
a, abc
100
10
```

图 2-29

```
mov     [ebp+a], 64h ; 'd'
add     [ebp+a], 1
sub     esp, 8
push    [ebp+a]
push    offset format   ; "%d \n"
call    _printf
add     esp, 10h
```

图 2-30

由图2-30可知，程序首先将64h赋给[ebp+a]，然后将[ebp+a]地址存储的数据的值加1，最后将[ebp+a]地址存储的数据使用printf函数输出，其中第一个参数为“%d \n”。因此，程序对应的C代码应为：

```
int a = 0x64;
printf("%d \n", a);
```

由图2-31可知，程序首先将ds:flt_8048568的值压入浮点寄存器，然后赋给[ebp+f1]，再将ds:flt_804856C的值压入浮点寄存器，并将两个浮点寄存器的值相加，最后将[ebp+f1]地址存储的数据使用printf函数输出，其中第一个参数为“%f \n”。查看ds:flt_8048568和ds:flt_804856C的值，结果如图2-32所示。

```
fld     ds:flt_8048568
fstp    [ebp+f1]
fld     [ebp+f1]
fld     ds:flt_804856C
faddp   st(1), st
fstp    [ebp+f1]
fld     [ebp+f1]
sub     esp, 4
lea     esp, [esp-8]
fstp    qword ptr [esp]
push    offset asc_8048555 ; "%f \n"
call    _printf
add     esp, 10h
```

图 2-31

```
.rodata:08048568 flt_8048568     dd 3.5                  ; DATA XREF: main+2F↑r
.rodata:0804856C flt_804856C     dd 2.0                  ; DATA XREF: main+3B↑r
```

图 2-32

因此，程序对应的C代码应为：

```
int f1 = 3.5;
f1 = f1+2.0
printf("%f \n", f1);
```

```
mov     [ebp+ch_0], 61h ; 'a'
mov     [ebp+pCh], offset aAbc ; "abc"
movsx   eax, [ebp+ch_0]
sub     esp, 4
push    [ebp+pCh]
push    eax
push    offset aCS      ; "%c, %s \n"
call    _printf
add     esp, 10h
```

图 2-33

```
mov     [ebp+b], 64h ; 'd'
sub     esp, 8
push    [ebp+b]
push    offset format   ; "%d \n"
call    _printf
add     esp, 10h
sub     esp, 8
push    0Ah
push    offset format   ; "%d \n"
call    _printf
add     esp, 10h
```

图 2-34

由图2-33可知，程序首先将'a'赋给[ebp+ch_0]，再将"abc"赋给[ebp+pCh]，最后将[ebp+ch_0]和[ebp+pCh]地址存储的数据使用printf函数输出，其中第一个参数为“%c, %s \n”。

因此，程序对应的C代码应为：

```
char ch_0 = 'a';
char* pCh = "abc";
printf("%c, %s \n", ch_0, pCh);
```

由图2-34可知，程序首先将64h赋给[ebp+b]，并使用printf函数输出，其中第一个参数为“%d \n”，再将常量0Ah使用printf函数输出，其中第一个参数为“%d \n”。

因此，程序对应的C代码应为：

```
#define N 10
int b = 0x64;
printf("%d \n", b);
printf("%d \n", N);
```

2.8　本 章 小 结

本章介绍了C语言中几种常见的基本数据类型的汇编代码，主要包括无符号和有符号整数的汇编代码，浮点数的汇编代码，字符和字符串的汇编代码，指针的汇编代码等。通过本章的学习，读者能够掌握整数、浮点数、指针等基本数据类型相关的汇编代码。

2.9 习　题

1. 已知汇编指令如下：

```
0x080483e1 <+6>:mov  dword ptr [ebp-0xc], 0xa
0x080483e8 <+13>:movdword ptr [ebp-0x8], 0xfffffff6
0x080483ef <+20>:movBYTE ptr [ebp-0xd], 0x7a
0x080483f3 <+24>:movdword ptr [ebp-0x4], 0xa
0x080483fa <+31>:moveax, 0x0
```

请分析汇编代码，写出对应的C代码。

2. 已知汇编指令如下：

```
0x080483e1 <+6>:fld  dword ptr ds:0x8048480
0x080483e7 <+12>:fstp   dword ptr [ebp-0x4]
0x080483ea <+15>:flddword ptr [ebp-0x4]
0x080483ed <+18>:fld1
0x080483ef <+20>:faddp  st(1), st
0x080483f1 <+22>:fstp   dword ptr [ebp-0x4]
0x080483f4 <+25>:moveax, 0x0
```

0x8048480地址存储的数据为00 00 c8 40，请分析汇编代码，写出对应的C代码。