

例题 1

一个C语言程序及其在X86/Linux操作系统上的编译结果如下。根据所生成的汇编程序来解释程序中四个变量的存储分配、生存期、作用域和置初值方式等方面的区别

```
static long aa = 10;
short bb = 20;
func() {
    static long cc = 30;
    short dd = 40;
}
```

```
static long aa = 10;
short bb = 20;
func() {
    static long cc = 30;
    short dd = 40; }
```

.data	.align 4	.type aa,@object	.size aa,4
aa:	.long 10	.globl bb	.align 2
.type bb,@object	.size bb,2	bb:	.value 20

.align 4	.type cc.2,@object	.size cc.2,4
cc.2:	.long 30	.text
.align 4	.globl func	func:
...	movw \$40,-2(%ebp)	...

```
static long aa = 10;
short bb = 20;
func() {
    static long cc = 30;
    short dd = 40; }
```

.data	.align 4	.type aa,@object	.size aa,4
aa:	.long 10	.globl bb	.align 2
.type bb,@object	.size bb,2	bb:	.value 20

.align 4	.type cc.2,@object	.size cc.2,4
cc.2:	.long 30	.text
.align 4	.globl func	func:
...	movw \$40,-2(%ebp)	...

```
static long aa = 10;
short bb = 20;
func() {
    static long cc = 30;
    short dd = 40; }
```

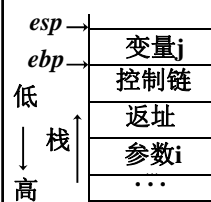
.data	.align 4	.type aa,@object	.size aa,4
aa:	.long 10	.globl bb	.align 2
.type bb,@object	.size bb,2	bb:	.value 20

.align 4	.type cc.2,@object	.size cc.2,4
cc.2:	.long 30	.text
.align 4	.globl func	func:
...	movw \$40,-2(%ebp)	...

例题 2

```
func(i)
long i;
{
    long j;
    j = i - 1;
    func(j);
}
```

```
func:
pushl %ebp 老的基地址指针压栈
movl %esp,%ebp 修改基地址指针
subl $4,%esp 为j分配空间
movl 8(%ebp),%edx 取i到寄存器
decl %edx i - 1
movl %edx,-4(%ebp) i - 1 => j
movl -4(%ebp),%eax
pushl %eax 把实参j的值压栈
call func 函数调用
addl $4,%esp 恢复栈顶指针
L1:
leave 即 mov ebp, esp; pop ebp
ret 即 pop eip(下条指令地址)
```



例 题 2

```
func(i)
long i;
{
    long j;
    j = i - 1;
    func(j);
}
```

esp →
 ebp →

```
func:
    pushl %ebp 老的基地址指针压栈
    movl %esp,%ebp 修改基地址指针
    subl $4,%esp 为j分配空间
    movl 8(%ebp),%edx 取i到寄存器
    decl %edx i - 1
    movl %edx,-4(%ebp) i - 1 ⇒ j
    movl -4(%ebp),%eax
    pushl %eax 把实参j的值压栈
    call func 函数调用
    addl $4,%esp 恢复栈顶指针
L1:
    leave 即 mov ebp, esp; pop ebp
    ret 即 pop eip(下条指令地址)
```

例 题 2

```
func(i)
long i;
{
    long j;
    j = i - 1;
    func(j);
}
```

esp →
 ebp →

```
func:
    pushl %ebp 老的基地址指针压栈
    movl %esp,%ebp 修改基地址指针
    subl $4,%esp 为j分配空间
    movl 8(%ebp),%edx 取i到寄存器
    decl %edx i - 1
    movl %edx,-4(%ebp) i - 1 ⇒ j
    movl -4(%ebp),%eax
    pushl %eax 把实参j的值压栈
    call func 函数调用
    addl $4,%esp 恢复栈顶指针
L1:
    leave 即 mov ebp, esp; pop ebp
    ret 即 pop eip(下条指令地址)
```

例 题 2

```
func(i)
long i;
{
    long j;
    j = i - 1;
    func(j);
}
```

esp →
 ebp →

```
func:
    pushl %ebp 老的基地址指针压栈
    movl %esp,%ebp 修改基地址指针
    subl $4,%esp 为j分配空间
    movl 8(%ebp),%edx 取i到寄存器
    decl %edx i - 1
    movl %edx,-4(%ebp) i - 1 ⇒ j
    movl -4(%ebp),%eax
    pushl %eax 把实参j的值压栈
    call func 函数调用
    addl $4,%esp 恢复栈顶指针
L1:
    leave 即 mov ebp, esp; pop ebp
    ret 即 pop eip(下条指令地址)
```

例 题 2

```
func(i)
long i;
{
    long j;
    j = i - 1;
    func(j);
}
```

esp →
 ebp →

```
func:
    pushl %ebp 老的基地址指针压栈
    movl %esp,%ebp 修改基地址指针
    subl $4,%esp 为j分配空间
    movl 8(%ebp),%edx 取i到寄存器
    decl %edx i - 1
    movl %edx,-4(%ebp) i - 1 ⇒ j
    movl -4(%ebp),%eax
    pushl %eax 把实参j的值压栈
    call func 函数调用
    addl $4,%esp 恢复栈顶指针
L1:
    leave 即 mov ebp, esp; pop ebp
    ret 即 pop eip(下条指令地址)
```

例 题 2

```
func(i)
long i;
{
    long j;
    j = i - 1;
    func(j);
}
```

esp →
 ebp →

```
func:
    pushl %ebp 老的基地址指针压栈
    movl %esp,%ebp 修改基地址指针
    subl $4,%esp 为j分配空间
    movl 8(%ebp),%edx 取i到寄存器
    decl %edx i - 1
    movl %edx,-4(%ebp) i - 1 ⇒ j
    movl -4(%ebp),%eax
    pushl %eax 把实参j的值压栈
    call func 函数调用
    addl $4,%esp 恢复栈顶指针
L1:
    leave 即 mov ebp, esp; pop ebp
    ret 即 pop eip(下条指令地址)
```

例 题 2

```
func(i)
long i;
{
    long j;
    j = i - 1;
    func(j);
}
```

esp →
 ebp →

```
func:
    pushl %ebp 老的基地址指针压栈
    movl %esp,%ebp 修改基地址指针
    subl $4,%esp 为j分配空间
    movl 8(%ebp),%edx 取i到寄存器
    decl %edx i - 1
    movl %edx,-4(%ebp) i - 1 ⇒ j
    movl -4(%ebp),%eax
    pushl %eax 把实参j的值压栈
    call func 函数调用
    addl $4,%esp 恢复栈顶指针
L1:
    leave 即 mov ebp, esp; pop ebp
    ret 即 pop eip(下条指令地址)
```

func(i)

long i;

{

long j;

j= i -1;

func(j);

}

esp →

ebp →

低

高

↓ 栈 ↑

变量j

控制链

返址

参数i

...

例 题 2 调用序列之一

调用序列之二

func:

pushl %ebp 老的基地址指针压栈

movl %esp,%ebp 修改基地址指针

subl \$4,%esp 为j分配空间

movl 8(%ebp),%edx 取i到寄存器

decl %edx i - 1

movl %edx,-4(%ebp) i - 1 ⇒ j

movl -4(%ebp),%eax

pushl %eax 把实参j的值压栈

call func 函数调用

addl \$4,%esp 恢复栈顶指针

L1:

leave 即 mov ebp, esp; pop ebp

ret 即 pop eip(下条指令地址)

func(i)

long i;

{

long j;

j= i -1;

func(j);

}

esp →

ebp →

低

高

↓ 栈 ↑

变量j

控制链

返址

参数i

...

例 题 2 返回序列之一

返回序列之二

func:

pushl %ebp 老的基地址指针压栈

movl %esp,%ebp 修改基地址指针

subl \$4,%esp 为j分配空间

movl 8(%ebp),%edx 取i到寄存器

decl %edx i - 1

movl %edx,-4(%ebp) i - 1 ⇒ j

movl -4(%ebp),%eax

pushl %eax 把实参j的值压栈

call func 函数调用

addl \$4,%esp 恢复栈顶指针

L1:

leave 即 mov ebp, esp; pop ebp

ret 即 pop eip(下条指令地址)

func(i)

long i;

{

long j;

j= i -1;

func(j);

}

esp →

ebp →

控制链

返址

参数i

...

例 题 2 返回序列之一

返回序列之二

func:

pushl %ebp 老的基地址指针压栈

movl %esp,%ebp 修改基地址指针

subl \$4,%esp 为j分配空间

movl 8(%ebp),%edx 取i到寄存器

decl %edx i - 1

movl %edx,-4(%ebp) i - 1 ⇒ j

movl -4(%ebp),%eax

pushl %eax 把实参j的值压栈

call func 函数调用

addl \$4,%esp 恢复栈顶指针

L1:

leave 即 mov ebp, esp; pop ebp

ret 即 pop eip(下条指令地址)

func(i)

long i;

{

long j;

j= i -1;

func(j);

}

esp →

ebp →

返址

参数i

...

例 题 2 返回序列之一

返回序列之二

func:

pushl %ebp 老的基地址指针压栈

movl %esp,%ebp 修改基地址指针

subl \$4,%esp 为j分配空间

movl 8(%ebp),%edx 取i到寄存器

decl %edx i - 1

movl %edx,-4(%ebp) i - 1 ⇒ j

movl -4(%ebp),%eax

pushl %eax 把实参j的值压栈

call func 函数调用

addl \$4,%esp 恢复栈顶指针

L1:

leave 即 mov ebp, esp; pop ebp

ret 即 pop eip(下条指令地址)

func(i)

long i;

{

long j;

j= i -1;

func(j);

}

esp →

ebp →

参数i

...

例 题 2 返回序列之一

返回序列之二

func:

pushl %ebp 老的基地址指针压栈

movl %esp,%ebp 修改基地址指针

subl \$4,%esp 为j分配空间

movl 8(%ebp),%edx 取i到寄存器

decl %edx i - 1

movl %edx,-4(%ebp) i - 1 ⇒ j

movl -4(%ebp),%eax

pushl %eax 把实参j的值压栈

call func 函数调用

addl \$4,%esp 恢复栈顶指针

L1:

leave 即 mov ebp, esp; pop ebp

ret 即 pop eip(下条指令地址)

func(i)

long i;

{

long j;

j= i -1;

func(j);

}

esp →

ebp →

...

例 题 2 返回序列之一

返回序列之二

func:

pushl %ebp 老的基地址指针压栈

movl %esp,%ebp 修改基地址指针

subl \$4,%esp 为j分配空间

movl 8(%ebp),%edx 取i到寄存器

decl %edx i - 1

movl %edx,-4(%ebp) i - 1 ⇒ j

movl -4(%ebp),%eax

pushl %eax 把实参j的值压栈

call func 函数调用

addl \$4,%esp 恢复栈顶指针

L1:

leave 即 mov ebp, esp; pop ebp

ret 即 pop eip(下条指令地址)

例 题 2 返回序列之一 返回序列之二

```
func(i)
long i;
{
    long j;
    j = i - 1;
    func(j);
}

func:
    pushl %ebp 老的基地址指针压栈
    movl %esp,%ebp 修改基地址指针
    subl $4,%esp 为j分配空间
    movl 8(%ebp),%edx 取i到寄存器
    decl %edx i - 1
    movl %edx,-4(%ebp) i - 1 ⇒ j
    movl -4(%ebp),%eax
    pushl %eax 把实参j的值压栈
    call func 函数调用
    addl $4,%esp 恢复栈顶指针
L1:
    leave 即 mov ebp, esp; pop ebp
    ret 即 pop eip(下条指令地址)
```

例 题 3

下面的程序运行时输出3个整数。试从运行环境和printf的实现来分析，为什么此程序会有3个整数输出？

```
main()
{
    printf(“%d, %d, %d\n”);
}
```

例 题 4

```
main()
{
    char *cp1, *cp2;

    cp1 = "12345";
    cp2 = "abcdefghij";
    strcpy(cp1, cp2);
    printf("cp1 = %s\ncp2 = %s\n", cp1, cp2);
}
```

在某些系统上的运行结果是：

cp1 = abcdefghij

cp2 = ghij

为什么cp2所指的串被修改了？

例 题 4

因为常量串“12345”和“abcdefghij”连续分配在常数区

执行前：

```
1 2 3 4 5 \0 a b c d e f g h i j \0
↑      ↑
cp1     cp2
```

例 题 4

因为常量串“12345”和“abcdefghij”连续分配在常数区

执行前：

```
1 2 3 4 5 \0 a b c d e f g h i j \0
↑      ↑
cp1     cp2
```

执行后：

```
a b c d e f g h i j \0 f g h i j \0
↑      ↑
cp1     cp2
```

例 题 4

因为常量串“12345”和“abcdefghij”连续分配在常数区

执行前：

```
1 2 3 4 5 \0 a b c d e f g h i j \0
↑      ↑
cp1     cp2
```

执行后：

```
a b c d e f g h i j \0 f g h i j \0
↑      ↑
cp1     cp2
```

现在的编译器大都把程序中的串常量单独存放在只读数据段中，因此运行时会报错

例 题 5

```
func(i,j,f,e)
short i,j; float f,e;
{
    short i1,j1; float f1,e1;
    printf(&i,&j,&f,&e);
    printf(&i1,&j1,&f1,&e1);
}
main()
{
    short i,j; float f,e;
    func(i,j,f,e);
}
```

Address of i,j,f,e = ...36, ...42, ...44, ...54 (八进制数)
Address of i1,j1,f1,e1 = ...26, ...24, ...20, ...14

例 题 5

```
func(i,j,f,e)
short i,j; float f,e;
{
    short i1,j1; float f1,e1;
    printf(&i,&j,&f,&e);
    printf(&i1,&j1,&f1,&e1);
}
main()
{
    short i,j; float f,e;
    func(i,j,f,e);
}
```

Sizes of short, int, long, float,
double = 2, 4, 4, 4, 8
(在SPARC/SUN工作站上)

Address of i,j,f,e = ...36, ...42, ...44, ...54 (八进制数)
Address of i1,j1,f1,e1 = ...26, ...24, ...20, ...14

例 题 5

```
func(i,j,f,e)
short i,j; float f,e;
{
    short i1,j1; float f1,e1;
    printf(&i,&j,&f,&e);
    printf(&i1,&j1,&f1,&e1);
}
main()
{
    short i,j; float f,e;
    func(i,j,f,e);
}
```

Sizes of short, int, long, float,
double = 2, 4, 4, 4, 8
(在SPARC/SUN工作站上)

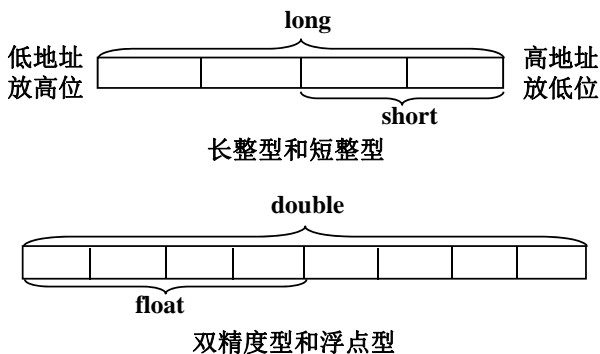
为什么4个形式参数i,j,f,e的地址
间隔和它们类型的大小不一致

Address of i,j,f,e = ...36, ...42, ...44, ...54 (八进制数)
Address of i1,j1,f1,e1 = ...26, ...24, ...20, ...14

例 题 5

- 当用传统的参数声明方式时，编译器不检查实参和形参的个数和类型是否一致，由程序员自己负责
- 但对形参和实参是不同的整型，或不同的实型
 - 编译器试图保证运行时能得到正确结果
 - 条件是：若需数据类型转换时，不出现溢出
- 编译器的做法
 - 把整型或实型数据分别提升到long和double类型的数据，再传递到被调用函数
 - 被调用函数根据形参所声明的类型，决定是否要将传来的实参向低级别类型转换

例 题 5



例 题 5

在main函数
中参数压栈
时的观点

在func函数中
存取形式参数
时的观点



例 题 6

下面程序为什么死循环（在SPARC/SUN工作站上）？

```
main() { addr(); loop(); }
long *p;
loop()
{
    long i,j;
    j=0;
    for(i=0;i<10;i++){ (*p)--; j++; }
}
addr() { long k; k=0; p=&k;}
```

例 题 6

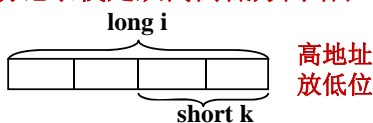
将long *p改成short *p，long k 改成short k 后，循环体执行一次便停止，为什么？

```
main() { addr(); loop(); }
short *p;
loop()
{
    long i,j;
    j=0;
    for(i=0;i<10;i++){ (*p)--; j++; }
}
addr() { short k; k=0; p=&k;}
```

例 题 6

将long *p改成short *p，long k 改成short k 后，循环体执行一次便停止，为什么？

```
main() { addr(); loop(); }
short *p;
loop()
{
    long i,j;
    j=0;
    for(i=0;i<10;i++){ (*p)--; j++; }
}
addr() { short k; k=0; p=&k;}
```



例 题 7

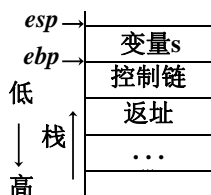
```
main()
{ func(); printf("Return from func\n"); }

func()
{ char s[4];
  strcpy(s,"12345678"); printf("%s\n",s); }
在X86/Linux操作系统上的运行结果如下：
12345678
Return from func
Segmentation fault (core dumped)
```

例 题 7

```
main()
{ func(); printf("Return from func\n"); }
```

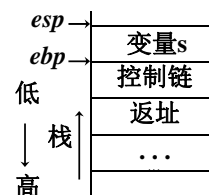
```
func()
{ char s[4];
  strcpy(s,"12345678");
  printf("%s\n",s);
}
```



例 题 7

```
main()
{ func(); printf("Return from func\n"); }
```

```
func()
{ char s[4];
  strcpy(s,"123456789");
  printf("%s\n",s);
}
```



123456789
Segmentation fault (core dumped)

例 题 8

```
int fact(i)      | main()
int i;          | {
{               |   printf("%d\n", fact(5));
  if(i==0)      |   printf("%d\n", fact(5,10,15));
    return 1;   |   printf("%d\n", fact(5.0));
  else         |   printf("%d\n", fact());
    return i*fact(i-1); | }
}
```

该程序在X86/Linux机器上的运行结果如下：

120

120

1

Segmentation fault (core dumped)

例 题 8

请解释下面问题：

- 第二个fact调用：结果为什么没有受参数过多的影响？
- 第三个fact调用：为什么用浮点数5.0作为参数时结果变成1？
- 第四个fact调用：为什么没有提供参数时会出现 Segmentation fault？

例 题 8

请解释下面问题：

- 第二个fact调用：结果为什么没有受参数过多的影响？

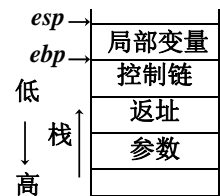
解答：参数表达式逆序计算并进栈，fact能够取到第一个参数

例 题 8

请解释下面问题：

- 第三个fact调用：为什么用浮点数5.0作为参数时结果变成1？

解答：参数5.0转换成双精度数进栈，占8个字节
它低地址的4个字节看成整数时正好是0



例 题 8

请解释下面问题：

- 第四个fact调用：为什么没有提供参数时会出现 Segmentation fault？

解答：由于没有提供参数，而main函数又无局部变量，fact把老ebp（控制链）（main的活动记录中保存的ebp）当成参数，它一定是一个很大的整数，使得活动记录栈溢出

