第十五章 函数

子程序

- 现代程序设计语言的灵魂
  - 提供一个基本程序块的集合
    - 扩充了运算和结构
  - 以模块化的方式写程序
- 提供了抽象的能力
  - 把功能与实现分开
  - 提高了构建复杂系统的能力

函数

- C程序本质上是函数的集合
  - 每条语句属于一个（并仅属于一个）函数
- 从main函数开始和结束执行

Hello World!

```
#include <stdio.h>

void SayHello();          /*函数声明*/

int main()
{
   SayHello ();           /*函数调用*/
}

void SayHello ()          /*函数定义*/
{
   printf("Hello World!\n");
}
```

- main函数:调用者
- SayHello:被调用者

计算圆面积

$$AreaOfCircle\ (r) = \pi \times r^2$$

```
1 #include <stdio.h>
2
3 double AreaOfCircle (double r);              /*函数声明*/
4
5 int main ()                                  /*main函数定义*/
6 {
7             double radius;                   /*来自用户的输入*/
8             double area;                     /*结果*/
9
10            printf ("Input a radius: ");
11
12            scanf ("%lf", &radius);
13
14            area = AreaOfCircle (radius);    /*调用AreaOfCircle */
15
16            printf ("The area of the circle is %f\n " , area);
17 }
```

```
18
19 /*计算一个给定半径的圆的面积*/
20 double AreaOfCircle (double r)              /*函数定义*/
21 {
22            double pi = 3.14159265;
23            double result;
24            result = pi * r * r;             /*计算圆面积*/
25            return result;                   /*返回调用者*/
26 }
```

函数声明

```
1 #include <stdio.h>
2
3 double AreaOfCircle (double r);          /*函数声明*/
4
5 int main ()                              /*main函数定义*/
6 {
7             double radius;               /*来自用户的输入*/
8             double area;                 /*结果*/
9
10            printf ("Input a radius: ");
11
12            scanf ("%lf", &radius);
13
14            area = AreaOfCircle (radius);   /*调用AreaOfCircle */
15
16            printf ("The area of the circle is %f\n " , area);
17 }
```

- 告诉编译器关于函数的一些相关属性
  - 在编译阶段对调用函数的合法性进行检查

函数声明

```
1 #include <stdio.h>
2
3 double AreaOfCircle (double r);          /*函数声明*/
```

- 函数的原型
  - 函数的名称，返回值的类型，参数的列表
  - 以分号结束

返回值类型 函数名(参数parameter的类型和顺序);

```
double AreaOfCircle (double);
void SayHello();
```

帕斯卡命名法

- Pascal-case，大驼峰式命名法
- 混合使用大小写字母来构成变量和函数的名字，首字母大写

函数调用

```
1 #include <stdio.h>
2
3 double AreaOfCircle (double r);          /*函数声明*/
4
5 int main ()                              /*main函数定义*/
6 {
7             double radius;               /*来自用户的输入*/
8             double area;                 /*结果*/
9
10            printf ("Input a radius: ");
11
12            scanf ("%lf", &radius);
13
14            area = AreaOfCircle (radius);   /*调用AreaOfCircle */
15
16            printf ("The area of the circle is %f\n " , area);
17 }
```

- 传给被调用者的值，变元（argument）/实际参数/实参
  - 任意的合法表达式
  - 与被调用者期望的类型相匹配

函数定义

```
18
19 /*计算一个给定半径的圆的面积*/
20 double AreaOfCircle (double r)        /*函数定义*/
21 {
22             double pi = 3.14159265;
23             double result;
24             result = pi * r * r;      /*计算圆面积*/
25             return result;            /*返回调用者*/
26 }
```

- 与函数声明相匹配

返回值类型 函数名(*形式参数列表*)；

- 形参列表
  - 变量声明的列表
  - 每一个变量都被初始化为调用者提供的变元/实参
- 实际参数应与形参列表的类型和顺序相匹配

函数体

```
18
19 /*计算一个给定半径的圆的面积*/
20 double AreaOfCircle (double r)        /*函数定义*/
21 {
22             double pi = 3.14159265;
23             double result;
24             result = pi * r * r;      /*计算圆面积*/
25             return result;            /*返回调用者*/
26 }
```

- 函数体
  - 函数执行计算的语句和声明
  - 在大括号中声明的变量
    - 该函数的局部变量

重要概念

```
1 #include <stdio.h>
2
3 double AreaOfCircle (double r);          /*函数声明*/
4
5 int main ()                              /*main函数定义*/
6 {
7             double radius;               /*来自用户的输入*/
8             double area;                 /*结果*/
9
10            printf ("Input a radius: ");
11
12            scanf ("%lf", &radius);
13
14            area = AreaOfCircle (radius);   /*调用AreaOfCircle */
15
16            printf ("The area of the circle is %f\n " , area);
17 }
```

```
18
19 /*计算一个给定半径的圆的面积*/
20 double AreaOfCircle (double r)          /*函数定义*/
21 {
22            double pi = 3.14159265;
23            double result;
24            result = pi * r * r;        /*计算圆面积*/
25            return result;              /*返回调用者*/
26 }
```

- 任何调用者的局部变量对于被调用函数都是不可见的
  - AreaOfCircle
    - 可以“看到”并且修改其局部变量pi,result和参数r
    - 不能修改main函数中的变量radius
    - 可以通过返回值修改main函数中的变量area
- 调用者的变元作为值被传给被调用者

函数返回值

```
18
19 /*计算一个给定半径的圆的面积*/
20 double AreaOfCircle (double r)        /*函数定义*/
21 {
22             double pi = 3.14159265;
23             double result;
24             result = pi * r * r;      /*计算圆面积*/
25             return result;            /*返回调用者*/
26 }
```

return 表达式;

- 控制传回到调用者
- 表达式的类型应与函数声明的返回值类型相匹配

- 不返回数值的函数
  - void类型
  - 最后一条语句执行后，控制传回调用者

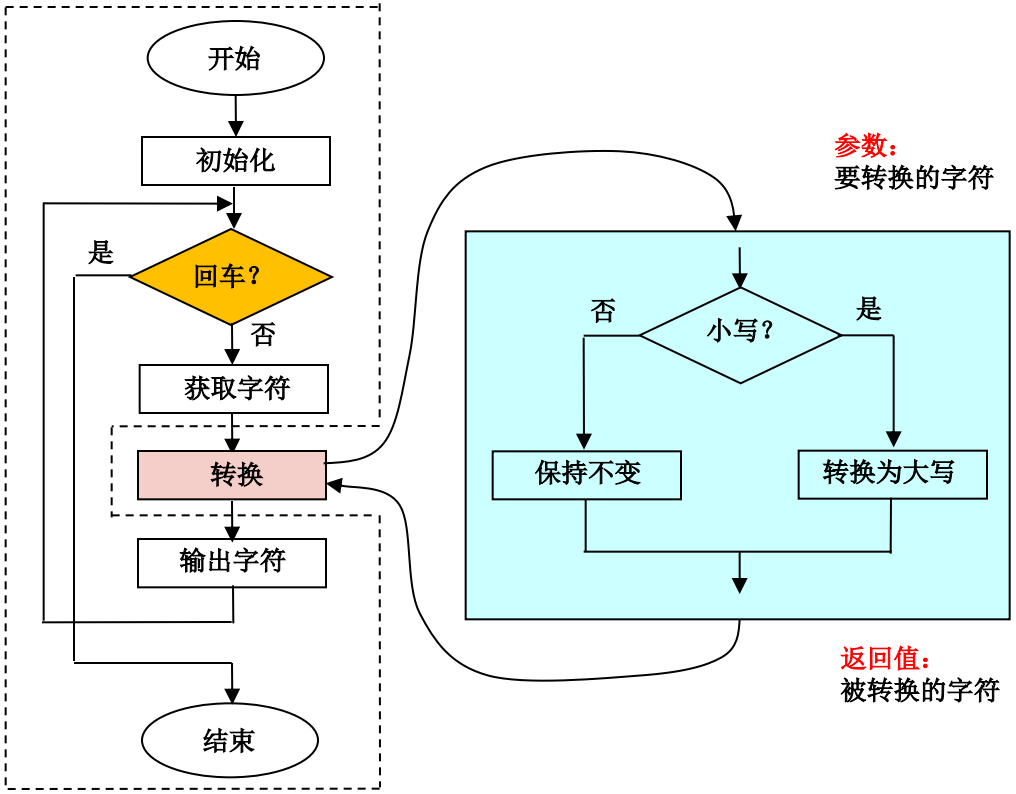
其它

- 函数声明
  - 函数定义在前，可以不声明
  - 参数列表，可以只有类型
  - 返回值类型：void——空类型
- 函数定义
  - 参数名可与声明时不相同
- 函数调用
  - 可作为语句、表达式、函数的参数
  - 变元：变量、常量或表达式，类型须与声明的类型兼容
  - 变元的运算顺序：自右向左（VC）
    - Func1(i,++i);
  - 返回值：类型不同，自动转换

问题求解

- 自顶向下，分解一个问题
  - 任务中的“组件”——使用函数

小写转换为大写

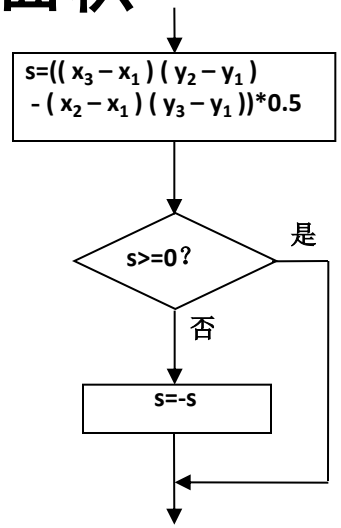

开始
初始化
是
回车？
否
获取字符
转换
输出字符
结束
参数：
要转换的字符
否
小写？
是
保持不变
转换为大写
返回值：
被转换的字符

```c
#include <stdio.h>

/*函数声明*/
char            ToUpper (char inchar);

/*主函数：*/
/*读入一个字符*/
/*转换为大写字母，输出，再读取下一个*/
int main ()
{
                char echo = 'A';          /*初始化输入的字符*/
                char upcase;

                while (echo != '\n')
                {
                                scanf ("%c", &echo);
                                upcase = ToUpper (echo);
                                printf ("%c", upcase);
                }
}
```

```c
/*ToUpper函数：*/
/*如果参数是小写字母*/
/*返回其大写字母的ASCII码*/
char            ToUpper (char inchar)
{
                char outchar;

                if ('a' <= inchar && inchar <= 'z')
                                outchar = inchar - ('a' - 'A');
                else
                                outchar = inchar;

                return outchar;
}
```

计算凸多边形的面积

- 首先，将多边形划分为多个三角形，然后计算各三角形的面积，通过累加各三角型的面积，得到多边形的面积



函数声明：

```
double AreaOfTriangle (double x1 , double x2, double
  x3 , double y1, double y2, double y3);
```

函数调用：

```
area = AreaOfTriangle (xa1 , xa2, xa3 , ya1, ya2, ya3)
  + AreaOfTriangle (xa1 , xa3, xa4 , ya1, ya3, ya4) +
  AreaOfTriangle (xa1 , xa4, xa5 , ya1, ya4, ya5) ;
```

函数的测试与调试

- 函数单独测试/调试
  - 当所有函数被整合起来，程序运行的机会更大
  - 模块化设计的一个益处
- 源水平调试器，单步命令的一些变种
  - Step Into
    - 函数调试
  - Step Over
    - 跳过函数，如I/O库函数、已完成测试/调试的函数等
  - Step Out
    - 一步完成函数调用

gdb命令

- n——step over,不会进入函数
- s——step into,进入函数
- fin——step out，运行至函数结束并跳出，并打印函数的返回值

使用断言技术

- 检查一个函数是否返回了一个在预期范围内的值
- 假如返回值超出了这个范围，就会显示一条错误信息

```
upcase = ToUpper (echo);    /*小写字母转换为大写字母*/

if (upcase != echo && (upcase < 'A' || upcase > 'Z'))
  printf ("Error in function ToUpper!\n");
```

语义错误/不完善的需求规格说明

```
19 int Factorial (int n) {          /*函数定义*/
20
21      int i;
22      int result;
23
24      for (i = 1; i <= n; i++)       /*计算阶乘*/
25              result = result * i;
26
27      return result;              /*返回调用者*/
28 }
```

C中的库函数

- 库是一个已测试的组件的集合，所有程序员都可以利用库编写代码
- 现代程序设计实践着重围绕库的使用，库体现了模块化设计的思想

计算直角三角形斜边长

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

double Squared (double x);        /*函数声明*/

int main () {

   double side1;                  /*一个直角边长*/
   double side2;                  /*另一个直角边长*/
   double hypot;                  /*斜边长*/

   printf ("Input side1: ");
   scanf ("%lf", &side1);
   printf ("Input side2: ");
   scanf ("%lf", &side2);
```

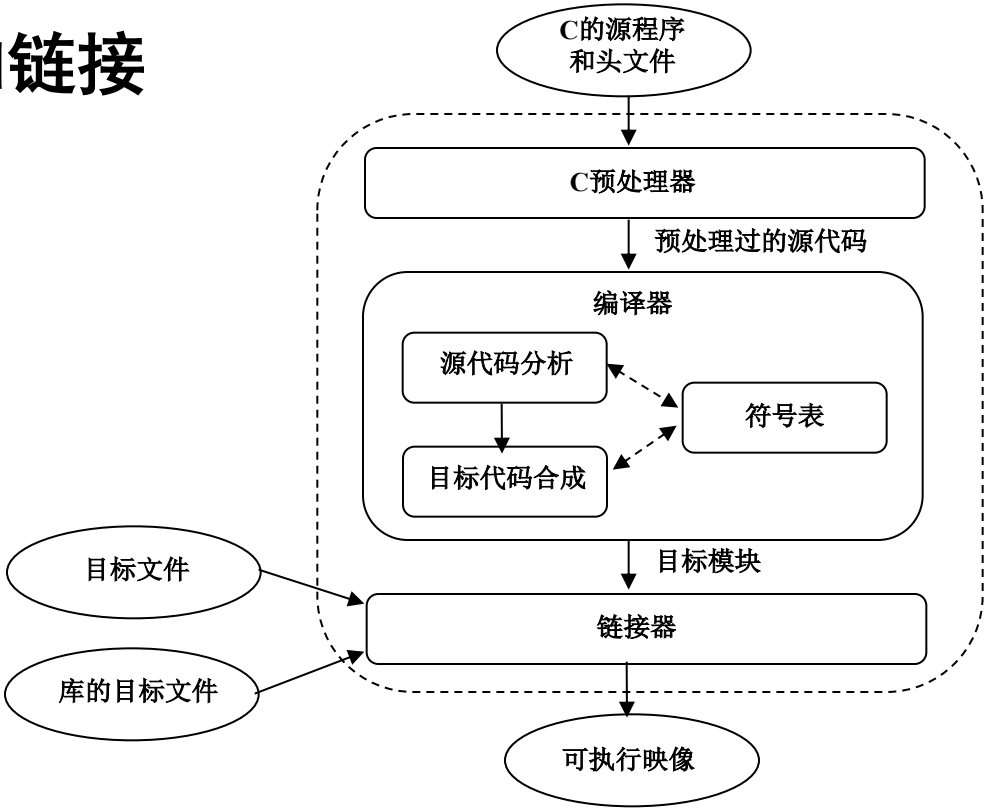
```
   hypot = sqrt (Squared (side1) + Squared (side2));/*函数调用*/
   printf ("The hypot is %f\n", hypot);
}

double Squared (double x) {  /*函数定义*/
   double answer;
   answer = x * x;
   return answer;                    /*返回调用者*/
}
```

头文件

- .h文件
  - 包含函数声明，预处理宏，以及其它信息
  - 不包含库函数的源代码
- 标准库中的函数按照功能分组
  - 每一组有一个相关的头文件
    - 数学函数，math.h
    - 标准I/O函数，stdio.h

编译

- 预处理、编译和链接

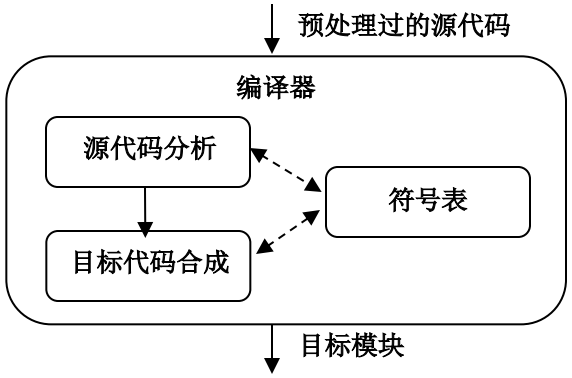

C的源程序
和头文件
C预处理器
预处理过的源代码
编译器
源代码分析
符号表
目标代码合成
目标模块
目标文件
链接器
库的目标文件
可执行映像

预处理器

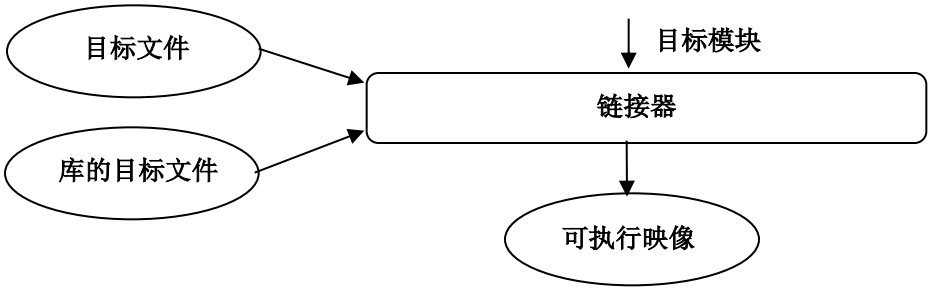
- 寻找以“#”开头的预处理指令，根据预处理指令执行
  - 与DLX汇编语言中的伪操作相似
    - #define X Y
    - #include <X.h>

编译器

- 目标模块
  - 一段机器代码
- 主要阶段
  - 分析
    - 读入、分析和构造原始程序的内部表示
    - 源程序被分解或者分析为其组成部分
  - 合成
    - 生成机器代码
    - 如果需要，进行代码优化



链接器



- 库的目标代码
  - 根据计算机系统被保存在一个特定地方
  - 例如，在UNIX中，/usr/lib目录
- 动态链接
  - 动态链接库[DLL]或共享库，根据需要被“链接”

I/O库函数

- 输出：屏幕，文件，网络等
- 一组标准库函数
  - 与调用TRAP指令执行服务例程类似
  - 由ANSI C标准严格定义

C函数在底层的实现

- 函数：子例程
  - 调用/返回机制
- 调用函数步骤：
  - （1）调用者的变元传给被调用者，并且控制权被传给被调用者；
  - （2）被调用者执行它的任务；
  - （3）返回值被传回给调用者，并且控制权返回调用者
- 函数与调用者无关
  - 一个函数应能被任何一个函数调用

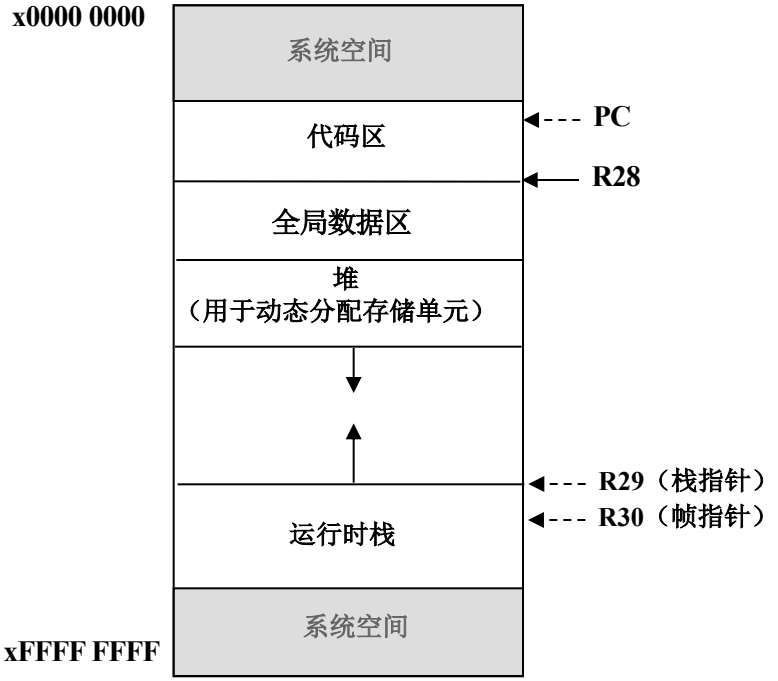
寄存器分配约定

R0	0
R1	汇编器保留
R2、R3	返回值
R4~R7	参数
R8~R15	临时值
R16~R23	局部变量
R24、R25	临时值
R26、R27	操作系统保留
R28	全局指针
R29	栈指针
R30	帧指针
R31	返回地址

- R1，是汇编器进行翻译时，如标记超过16位立即数表示的范围，将一条汇编指令翻译为多条机器指令，用到的临时寄存器
  - 模拟器使用R25
- R26、R27，操作系统，用于保存EPC的值，EPC是出现异常时的指令地址，在异常处理结束，使用JR R26/R27返回

存储器的组织（类UNIX）

- 当寄存器数量不足时，需要使用存储器



是否允许递归

- 问题：每个函数都有一个活动记录？还是每一次函数调用都有一个活动记录？
  - 区别在于是否允许函数调用它本身
  - ——“递归”

C语言支持递归

- 编译器为每一次函数调用，分配一个活动记录
  - 当函数返回时，它的活动记录被回收
  - 为局部数值获得它自己的空间
- 通过栈结构，实现活动记录的分配
  - “运行时栈”

支持递归

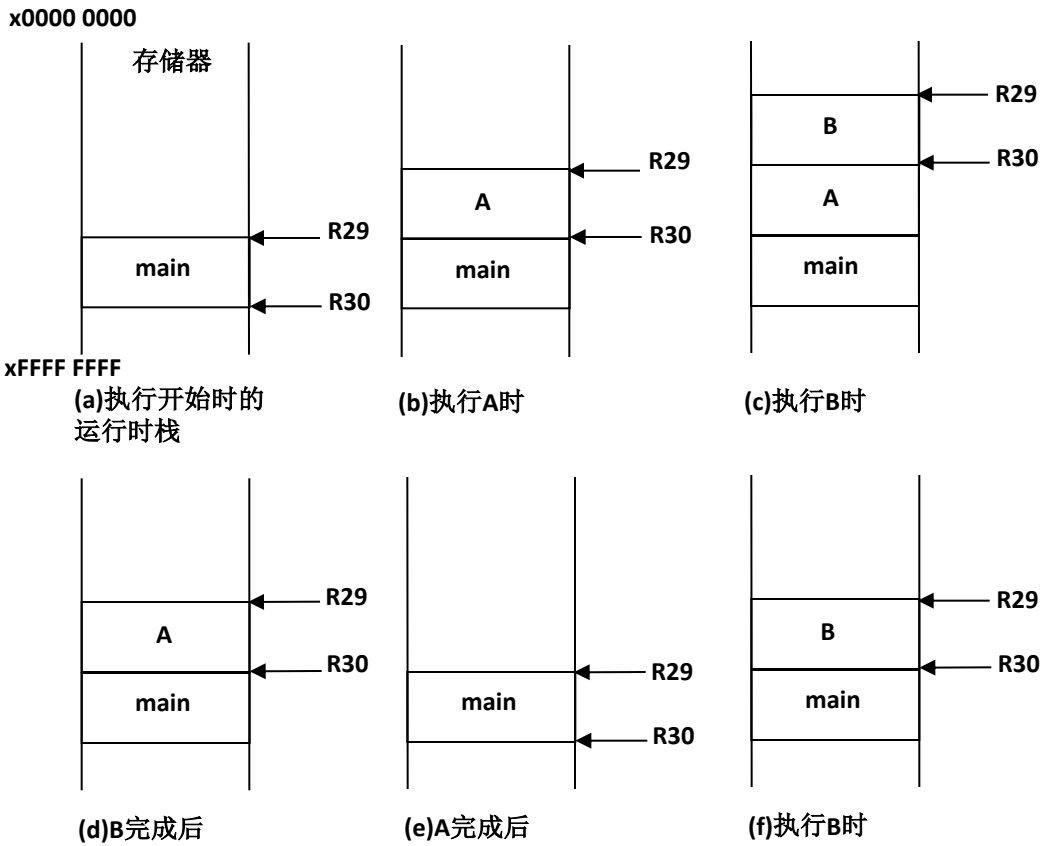
- 编译器为每一次函数调用，在存储器中分配一个活动记录
- 当函数返回时，它的活动记录将被回收，以便被分配给后面的函数
- 每一次函数调用都会在存储器中为其局部数值获得它自己的空间
- 可以通过一个栈结构，实现活动记录的分配
  - 采用“运行时栈”分配空间的原因

示例

```
int main() {
   int x1, x2, x3, x4, x5;
   int y;
   int z;
   ... ...
   y = A (x1, x2, x3, x4, x5);
   z = B (y);
   ... ...
}
```

```
int A (int i, int j, int k, int m, int n) {
    int x;
    int y;
    x = i + n ;
    ... ...
    y = B (x);
    ... ...
    return y;
}
int B (int n) {
    int x;
    int y;
    ... ...
    return y;
}
```

运行时栈


x0000 0000
存储器
R29
main
R30
xFFFF FFFF
(a)执行开始时的
运行时栈
R29
A
R30
main
(b)执行A时
R29
B
R30
A
main
(c)执行B时
R29
A
R30
main
(d)B完成后
R29
main
R30
(e)A完成后
R29
B
R30
main
(f)执行B时

函数调用机制

- 当一个函数被调用的时候，在机器层要进行很多工作
  - 变元必须被传递，活动记录被压入、弹出，控制从一个函数转移到另一个
  - 某些工作由调用函数完成，某些由被调用函数完成

步骤

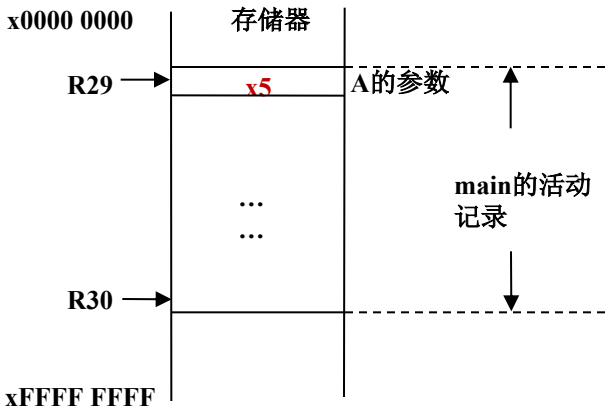
- 第一，调用函数
  - 使用参数寄存器R4~R7存储变元的值
  - 如果需要（参数 > 4个），将变元分配到运行时栈（被调用函数可以访问的存储区域）中
- 第二，被调用函数
  - 完成活动记录的分配
    - 将一些寄存器的值保存到运行时栈中，使得当控制返回到调用函数时，调用者的寄存器看起来好像没被动过——寄存器的保存
    - 如果需要（局部变量寄存器不足），将局部变量分配到运行时栈中
- 第三，被调用函数
  - 执行任务
    - 可以调用其它函数，或递归调用……
- 第四，被调用函数
  - 当完成工作时，活动记录从栈中弹出，并且控制返回到调用函数
- 最后，调用函数
  - 一旦控制返回到调用函数，执行代码取回被调用函数的返回值

第一步 函数调用

- y = A (x1, x2, x3, x4, x5);

```
;main函数使用R16~R20存储x1~x5的值
addi    r4, r16, #0    ; i，用R4~R7存储i~m
addi    r5, r17, #0    ; j
addi    r6, r18, #0    ; k
addi    r7, r19, #0    ; m
subi    r29, r29, #4
sw      0(r29) , r20   ; 压入x5，即n
jal     A
```

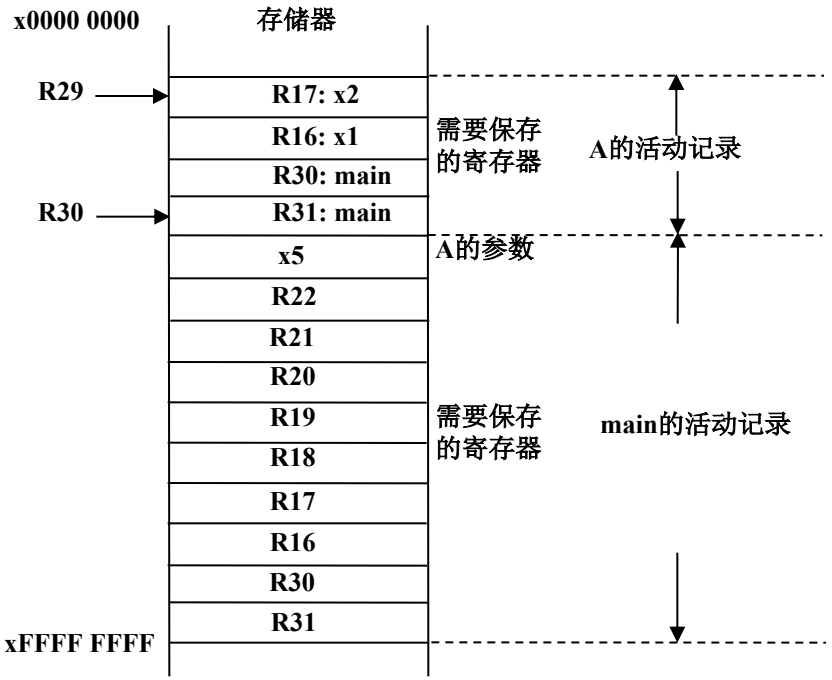
变元入栈



第二步 开始被调用函数

- 寄存器的保存
  - 1、被调用者把R31中的返回地址的一份副本压入栈中
  - 2、被调用者把R30中的动态链接（调用者的框架指针）的一份副本压入栈中
- 3、调整R30指向被调用者的活动记录的最底部
  - 4、被调用者将需要保存的其他寄存器压栈
- 5、被调用者为一些局部变量在栈上分配足够的空间，最后，R29指向栈的顶部

A的活动记录入栈


x0000 0000
存储器
R29
R17: x2
R16: x1
R30: main
R30
R31: main
x5
R22
R21
R20
R19
R18
R17
R16
R30
R31
xFFFF FFFF
需要保存
的寄存器
A的活动记录
A的参数
需要保存
的寄存器
main的活动记录

A

```
A:      subi    r29, r29, #4
        sw      0(r29), r31       ; 压入R31（返回地址）

        subi    r29, r29, #4
        sw      0(r29), r30       ; 压入R30（框架指针，动态链接）

        addi    r30, r29, #4      ; 设置新的框架指针

        subi    r29, r29, #4
        sw      0(r29), r16       ; 压入R16（寄存器的保存）
        subi    r29, r29, #4
        sw      0(r29), r17       ; 压入R17（寄存器的保存）
        ......                    ; 为局部变量分配空间（如果需要的话）
```

第三步 执行被调用函数

- 执行被调用函数

```
……
lw      r8, 4(r30)          ;对n，即变元x5的访问
add     r16, r4, r8         ; x = i + n
……
addi    r4, r16, #0         ;用R4存储x,即n(B函数)
jal     B
……
```

返回值的处理

- 最后是返回值的处理
- 在需要返回值的情况下
  - 函数A使用R2保存返回值，代码如下：

```
addi    r2, r17, #0    ; return y;
```

第四步 结束被调用函数

- 一旦被调用函数完成了它的工作，它在将控制权返还给调用函数之前，必须弹出当前的活动记录：
  - 1、将变元从栈中弹出（如果在此函数执行过程中，又存在函数调用，且有变元压栈）
  - 2、局部变量从栈中弹出（如果此函数有需要存储器保存的局部变量）
  - 3、恢复保存的寄存器
  - 4、恢复动态链接
  - 5、恢复返回地址
  - 6、JR R31，将控制权返回给调用程序

A

```
.....                          ; 参数出栈
.....                          ; 局部变量出栈
lw      r17, 0(r29)            ; 恢复寄存器R17
addi    r29, r29, #4
lw      r16, 0(r29)            ; 恢复寄存器R16
addi    r29, r29, #4

lw      r30, 0(r29)            ; 动态链接出栈
addi    r29, r29, #4
lw      r31, 0(r29)            ; 返回地址出栈
addi    r29, r29, #4

jr      r31
```

第五步 从调用函数返回

- 当被调用函数执行了JR R31指令之后，控制被传回调用函数
- 注意返回值的处理
- JAL后面的代码如下所示：

```
jal     A
addi    r21, r2, #0    ; y = A (x1, x2, x3, x4, x5);
```

寄存器的保存与恢复

- 在一个函数的执行过程中，R8~R15和R24、R25作为临时寄存器，存储函数执行过程中用到的临时数据；R4~R7作为参数寄存器，是否也需要在运行时栈中保存？
- 在如下情况下，需要保存和恢复：
  - 如果调用者在调用后还将用到临时寄存器（R8~R15和R24、R25）或参数寄存器（R4~R7）

重要概念——重申

- 任何调用者的局部变量对于被调用函数都是不可见的
- 在C语言中，调用者的变元作为值被传给被调用者

I/O库 printf和scanf

- 传给这两个函数的参数的个数是可变的——**可变参数列表**
- 格式用字符串中的每一个转换说明和出现在格式用字符串后面的每个参数之间存在着一个一一对应的关系

- 如果printf函数的转换说明与参数不对应？
  - printf（“The value of nothing is %d\n”）;
  - 假定正确的参数个数被写进栈中，所以它从栈中为说明%d盲目的读入一个数值
- scanf函数
  - 可以利用scanf函数的返回值，检测该函数在输入流中成功扫描的格式说明的个数

C语言源水平调试器

- 根据来自编译过程的信息，可以在断点处检查程序的所有执行状态，例如，变量、存储器、栈以及寄存器的值
- 有些调试器还允许查看编译器为源代码产生的汇编代码，并对汇编代码进行单步调试

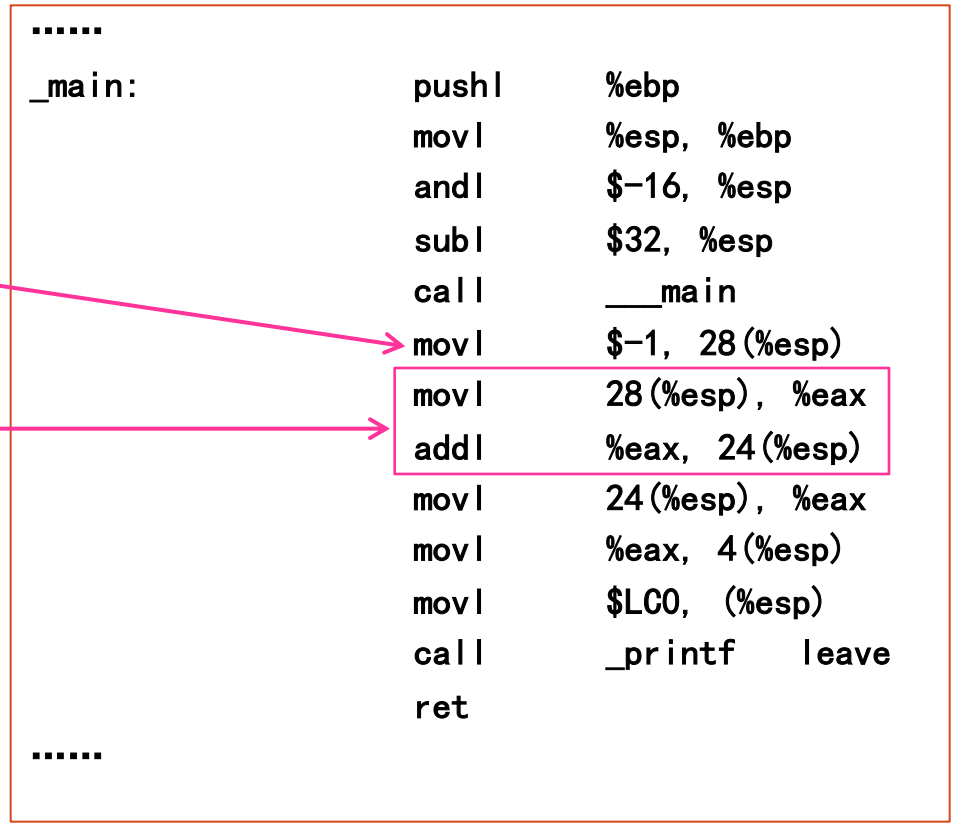
IA-32

- 将变量均分配到存储器中（压栈）
  - 寄存器ebp，扩展基址指针寄存器(extended base pointer)，帧指针
  - 寄存器esp，栈指针
  - 寄存器eax，累加器

IA-32 示例

```
#include <stdio.h>
int main(){
    int x = -1;
    int y;
    y = y + x;
    printf("%d\n", y);
}
```

```
.....
_main:          pushl   %ebp
                movl    %esp, %ebp
                andl    $-16, %esp
                subl    $32, %esp
                call    ___main
                movl    $-1, 28(%esp)
                movl    28(%esp), %eax
                addl    %eax, 24(%esp)
                movl    24(%esp), %eax
                movl    %eax, 4(%esp)
                movl    $LC0, (%esp)
                call    _printf    leave
                ret
.....
```



IA-32 示例

```
……
LC0:
            .ascii    "%d\12\0"              ;格式用字符串
            .text                            ;代码区
            .globl    _main
            ……
_main:      pushl     %ebp                   ;寄存器保存，框架指针，%ebp类似R30
            movl      %esp, %ebp             ;调整框架指针至新的运行时栈
            andl      $-16, %esp             ;对齐
            subl      $32, %esp              ;调整栈顶指针，%esp类似R29
            call      ___main
            movl      $-1, 28(%esp)          ;M[%esp+28] ← -1,  “将-1存入栈中;”
            movl      28(%esp), %eax         ; %eax ← M[%esp+28] ,  “从栈中取出x的值”
            addl      %eax, 24(%esp)       ; M[%esp+24] ← M[%esp+24] + %eax,  “y=y+x;”
            movl      24(%esp), %eax         ;%eax ← M[%esp+24],  “取出y的值”
            movl      %eax, 4(%esp)          ; M[%esp+4] ← %eax,  “将变元y压栈”
            movl      $LC0, (%esp)           ; 将LC0,格式用字符串地址压栈
            call      _printf                ; 调用printf函数
            leave
            ret                              ; 返回
……
```

习题

- 上机
  - 15.4
    - （1）y为整数
    - （3）
  - 15.5
  - 15.6
- 书面作业
  - 15.1
  - 15.2
  - 15.3
  - 15.9