C Language 进阶

1.非局部跳转语句---setjmp和longjmp函数

特点
非togo语句在函数内实施跳转,而是在栈上跳过若干调用帧,返回到当前函数调用路径上的某一语句.
头文件包含#include<setjmp.h>.

```
Void longjmp(jmp_buf env,int val);
```

返回值：若直接调用则返回0，若从longjmp调用返回则返回非0值
注：setjmp参数evn的类型是一个特殊的类型jmp_buf,这一数据类型是某种形式的数组，其中存放在调用longjmp时能用来恢复栈状态的所有信息.因为需要在另一个函数中引用env变量，所以规范的处理方式是将env变量定义为全局变量。

使用方法
在希望返回到的位置调用setjmp, 当检查到一个错误时,则以两个参数调用longjmp函数,第一个就是在调用setjmp时所用的env，第二个参数是具有非0值的val，它将成为从setjmp处返回的值.使用第二个参数的原因是对于一个setjmp可以有多个longjmp。
注:在使用longjmp跳转到setjmp中时，程序主动的退出了！相当于抛出一个异常退出！
使用setjmp和longjmp要注意以下几点:

1. setjmp与longjmp结合使用时，它们必须有严格的先后执行顺序，也即先调用setjmp函数，之后再调用longjmp函数，以恢复到先前被保存的"程序执行点"。
2. 不要假设寄存器类型的变量将总会保持不变.在调用longjmp之后，通过setjmp所返回的控制流中，程序中寄存器类型的变量将不会被恢复。 寄存器类型的变量一般都是临时变量，在C语言中，通过register定义，或直接嵌入汇编代码的程序。
3. longjmp必须在setjmp调用之后，而且longjmp必须在setjmp的作用域之内.

2. 位字段(bit-field)

在存储空间很宝贵的情况下,有可能需要将多个对象保存在一个机器字中,一种常用的方法是:使用类似于编译器符号表的单个二进制位标志集合,外部强加的数据格式(如设备接口等寄存器)经常需要从字的分值中读取数值.
通常采用的方法是:定义一个于相关位的位置对应的"屏蔽码"集合,如:

```
#define KEYWORD  (1<<0)
#define EXTRENAL (1<<2)
#define STATIC   (1<<3)
```

或者

```
enum{
    KEYWORD  = 01,
    EXTRENAL = 02,
    STATIC   = 04
};
```

这些数字必须是2的幂,这样就可以用移位运算,屏蔽运算以及补码运算进行简单的操作.比如:

```
flags |= EXTEERNAL | STATIC;//置1
flags &= ~(EXTEERNAL | STATIC);//置0
```

尽管这样的方法容易掌握,但是C语言提供了一种可以替代的方法,即直接定义和方位一个位字段的能力,不必通过以上的逻辑运算符,即位字段.通过位字段,以上的#define定义可以用以下的语句替代:

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern : 1;
    unsigned int is_static : 1;
}flafs;
```

这里定义一个变量flags,它包含3个1位的字段,冒号后的数字表示字段的宽度(用二进制位数表示),字段被声明为unsigned int,以保证它们的无符号量.
单个字段的引用方式与其他结构成员相同,例如:

```
flags.is_keyword,
flags.is_extern
```

等;字段的作用与小整数相似,同其他整数一样,字段可以出现在算数表达式中,因此,可以表示为:

```
flags.is_extern = flags.is_static = 1;//置1
flags.is_extern = flags.is_static = 0;//置0
if(flags.is_extern == 0 && flags.is_static == 0)
    ...//用于对is_extern和is_static的测试
```

字段的所有属性几乎都同具体的实现有关,字段可以不命名,无名字段(只有冒号和宽度)起填充作用,特殊宽度0可以用来强制在下一边界上对齐.字段不是数组,没有地址,不能做&取地址操作.

3. 结构数组

对于大小相同但是类型不同的数组,定义结构体数组对其很有帮组.例如:

```
char *keyword[NKEYS];
int   keycount[NKEYS];
```

这两个数组大小相同,因此 可以用另一种不同的组织方式,也就是结构数组.形如:

```
struct key{
    char *word;
    int   count;
}keytab[NKEYS];
```

因此两个数组用一个结构体数组即可定义.

4. (C++)inline关键字

背景

inline关键字用来定义一个类的内联函数，引入它的主要原因是用它替代C中表达式形式的宏定义。

表达式形式的宏定义如：

```
#define ExpressionName(Var1,Var2) ((Var1)+(Var2))*((Var1)-(Var2))
```

取代这种形式的原因如下：

1. C中使用#define这种形式宏定义的原因是因为，C语言是一个效率很高的语言，这种宏定义在形式及使用上像一个函数，但它使用预处理器实现，没有了参数压栈，代码生成等一系列的操作,因此，效率很高，这是它在C中被使用的一个主要原因。
2. 这种宏定义在形式上类似于一个函数，但在使用它时，仅仅只是做预处理器符号表中的简单替换，因此它不能进行参数有效性的检测，也就不能享受C++编译器严格类型检查的好处，另外它的返回值也不能被强制转换为可转换的合适的类型，这样，它的使用就存在着一系列的隐患和局限性。
3. 在C++中引入了类及类的访问控制，这样，如果一个操作或者说一个表达式涉及到类的保护成员或私有成员，你就不可能使用这种宏定义来实现(因为无法将this指针放在合适的位置)。
4. inline 推出的目的，也正是为了取代这种表达式形式的宏定义，它消除了宏定义的缺点，同时又很好地继承了宏定义的优点。

对上面的1-3点，阐述如下：

1. inline 定义的类的内联函数，函数的代码被放入符号表中，在使用时直接进行替换，（像宏一样展开），没有了调用的开销，效率也很高。
2. 很明显，类的内联函数也是一个真正的函数，编译器在调用一个内联函数时，会首先检查它的参数的类型，保证调用正确。然后进行一系列的相关检查，就像对待任何一个真正的函数一样。这样就消除了它的隐患和局限性。
3. inline 可以作为某个类的成员函数，当然就可以在其中使用所在类的保护成员及私有成员。

在何时使用inline函数：
首先，你可以使用inline函数完全取代表达式形式的宏定义。
另外要注意，内联函数一般只会用在函数内容非常简单的时候，这是因为，内联函数的代码会在任何调用它的地方展开，如果函数太复杂，代码膨胀带来的恶果很可能会大于效率的提高带来的益处。**内联函数最重要的使用地方是用于类的存取函数。**

简单提一下inline 的使用吧： 1.在类中定义这种函数：

```
class ClassName{
    ....
    //如果在类中直接定义，不需要用inline修饰,编译器自动化为内联函数
    INT GetWidth(){return m_lPicWidth;};//此说法在《C++ Primer》中提及
    ....
}
```

2.在类外定义前加inline关键字:

```
class Account {
public:
    Account(double initial_balance) { balance = initial_balance; } //与1相同
    double GetBalance(); //在类中声明
    double Deposit(double Amount);
    double Withdraw(double Amount);
private:
    double balance;
};
```

```
inline double Account::GetBalance() { return balance; } //在类外定义时添加inline关键字
inline double Account::Deposit(double Amount) { return ( balance += Amount ); }
inline double Account::Withdraw(double Amount) { return ( balance -= Amount ); }
```

注意

1. inline说明对编译器来说只是一种建议，编译器可以选择忽略这个建议。比如，你将一个长达1000多行的函数指定为inline，编译器就会忽略这个inline，将这个函数还原成普通函数。
2. 在调用内联函数时，要保证内联函数的定义让编译器"看"到，也就是说内联函数的定义要在头文件中，这与通常的函数定义不一样。但如果你习惯将函数定义放在CPP文件中，或者想让头文件更简洁一点，可这样做:

```
//SomeInline.h中
#ifndef SOMEINLINE_H
#define SOMEINLINE_H
inline Type Example(void);
//........其他函数的声明
#include“SomeInlie.cpp” //源文件后缀名随编译器而定
#endif
```

```
//SomeInline.cpp中
#include"SomeInline.h"
Type Example(void)
{
    //..........
}
//................其他函数的定义
```

以上方法是通用、有效的，可放心使用，不必担心在头文件包含CPP文件会导致编译错误。

linux内核和其他一些开源的代码中，经常会遇到这样的代码：

```
do{
 ...
}while(0)
```

这样的代码一看就不是一个循环，do..while表面上在这里一点意义都没有，那么为什么要这么用呢?

5. (C)do{...}while(0)的作用

实际上，do{...}while(0)的作用远大于美化你的代码。
总结起来这样写主要有以下几点好处：

1. 辅助定义复杂的宏

避免引用的时候出错： 举例来说，假设你需要定义这样一个宏：

```
#define DOSOMETHING()\
                foo1();\
                foo2();
```

这个宏的本意是，当调用DOSOMETHING()时，函数foo1()和foo2()都会被调用。但是如果你在调用的时候这么写：

```
if(a>0)
    DOSOMETHING();
```

因为宏在预处理的时候会直接被展开，你实际上写的代码是这个样子的：

```
if(a>0)
    foo1();
foo2();
```

这就出现了问题，因为无论a是否大于0，foo2()都会被执行，导致程序出错。
那么仅仅使用{}将foo1()和foo2()包起来行么？

我们在写代码的时候都习惯在语句右面加上分号，如果在宏中使用{}，代码里就相当于这样写了：“{...};”，展开后就是这个样子：

```
if(a>0)
{
    foo1();
    foo2();
};
```

这样甚至不会编译通过。所以，很多人才采用了do{...}while(0);

```
#define DOSOMETHING() \
            do{ \
                foo1();\
                foo2();\
```

```
            }while(0)\

    ...

if(a>0)
    DOSOMETHING();
```

这样，宏被展开后，才会保留初始的语义。
GCC提供了Statement-Expressions用以替代do{...}while(0)；所以你也可以这样定义宏：

```
#define DOSOMETHING() ({\
                foo1(); \
                foo2(); \
})
```

2. 避免使用goto对程序流进行统一的控制

有些函数中，在函数return之前我们经常会进行一些收尾的工作，比如free掉一块函数开始malloc的内存，goto一直都是一个比较简便的方法：

```
int foo()
{
    somestruct* ptr = malloc(...);

    dosomething...;
    if(error)
    {
        goto END;
    }

    dosomething...;
    if(error)
    {
        goto END;
    }
    dosomething...;

END:
    free(ptr);
    return 0;

}
```

由于goto不符合软件工程的结构化，而且有可能使得代码难懂，所以很多人都不倡导使用，那这个时候就可以用do{}while(0)来进行统一的管理：

```
int foo()
{
    somestruct* ptr = malloc(...);

    do{
        dosomething...;
        if(error)
        {
            break;
        }

        dosomething...;
        if(error)
        {
            break;
        }
        dosomething...;
    }while(0);

    free(ptr);
    return 0;
}
```

这里将函数主体使用do()while(0)包含起来，使用break来代替goto，后续的处理工作在while之后，就能够达到同样的效果。

3. 避免空宏引起的warning

内核中由于不同架构的限制，很多时候会用到空宏，在编译的时候，空宏会给出warning，为了避免这样的warning，就可以使用do{}while(0)来定义空宏：

```
#define EMPTYMICRO do{}while(0)
```

4. 定义一个单独的函数块来实现复杂的操作：

当你的功能很复杂，变量很多你又不愿意增加一个函数的时候，使用do{}while(0);将你的代码写在里面，里面可以定义变量而不用考虑变量名会同函数之前或者之后的重复。

转载自：http://www.spongeliu.com/415.html

6. (常用数据结构)程序控制块

1. 程序控制块

从代码上看,程序控制块就是一个结构体.例如:

```
typedef struct tcb{
    char * tast_name; //任务名字
    int    p; //任务重要级别
    int    v_number; //版本号
    void (*fun)(void); //指向存储任务代码空间地址
}TCB;
```

操作系统可以通过这个结构体控制与之相关联的代码,因此把这种结构叫做程序控制块.
例子:

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

//TCB定义
typedef struct tcb{
    char * task_name; //任务名字
    int    p; //任务重要级别
    int    v_number; //版本号
    void (*fun)(void); //指向存储任务代码空间地址
}TCB;

//任务1
void Task1()
{
    int i;
    for (i=0; i<10; i++)
        printf("1111111111\n");
}

//任务2
void Task2()
{
    int i;
    for (i=0; i<10; i++)
        printf("222222222222\n");
}
//任务3
void Task3()
{
    int i;
    for (i=0; i<10; i++)
        printf("3333333333333\n");
}
//创建控制块函数
TCB GreatTCB(char *name, int pp, int vnum, void (*f)())
{
    TCB tcb;
    tcb.task_name = name;
    tcb.p = pp;
```

```
    tcb.v_number = vnum;
    tcb.fun = f;
    return tcb;
}

//主任务
int main()
{
    char name_buf[10];
    int t, i;

    //定义TCB数组大小
    TCB tcbTbl[3];

    //创建task
    tcbTbl[0] = GreatTCB("task1", 2, 1, Task1);
    tcbTbl[1] = GreatTCB("task2", 3, 4, Task2);
    tcbTbl[2] = GreatTCB("task3", 4, 4, Task3);

    printf("Input task name: ");
    gets(name_buf);

    t = 0;
    //seek
    for (i=0; i<3; i++)
    {
        if (strcmp(tcbTbl[i].task_name, name_buf) == 0)
        {
            tcbTbl[i].fun();
            t = 1;
        }

        if (i == 2 && t == 0)
            printf("No %s\n", name_buf);
    }
    return 0;
}
```

2. 控制块链表

为了方便管理和组织程序控制块,一版在TCB中再定义两个指针,一个前指针, 一个后指针,用于把TCB组织起来,方便管理; 并且当程序控制块数组过大时,还会单独定义一个数组,数组的各个元素分别按照顺序指向程序控制块链表,这样做的目的是为了提高程序运行速度,因为链表查询很耗时.

7. 位操作总结

位与&，位或|，位取反~，位异或^。

1. 位与&，位或|，位取反~，位异或^的特点总结

位与：（任何数，其实就是1或者0）与1位与无变化，与0位与变成0
位或：（任何数，其实就是1或者0）与1位或变成1，与0位或无变化
位异或：（任何数，其实就是1或者0）与1位异或会取反，与0位异或无变

2. 左移位<<和右移位>>的特点总结

（C语言的移位取决于数据类型）
对于无符号数，左移时右侧补0（相当于逻辑移位）
对于无符号数，右移时左侧补0（相当于逻辑移位）
对于有符号数，左移时右侧补0（叫算术移位，相当于逻辑移位）
对于有符号数，右移时左侧补符号位（如果正数就补0，负数就补1，叫算术移

3. 小记

~（0u） 是全1；
常与1做位运算，来得到想要的数； 通过宏来置位，复位。

```
#define SET_NTH_BIT(x,n) (x|((1U)<<(n-1)));
#define CLEAR_NTH_BIT(x,n) (x & ~((1U)<<(n-1)));
```

8. 宏offsetof和宏container_of

1. offsetof宏

```
#define offsetof(TYPE, MEMBER) ((int) &((TYPE *)0)->MEMBER)
```

作用：用宏来计算结构体中某一个元素相对结构体首地址的偏移量；
原理：虚拟一个type类型的结构体，然后用type.member的方式来访问那个member元素，继而得到member相对整个变量首地址的偏移量；
思路： `(TYPE *) 0`是一个强制类型转换，把0地址强制转换成一个指针，这个指针指向一个TYPE类型的结构体变量(实际上这个结构体变量可能不存在，但是只要我们不去引用这个指针就不会出错)。

2. container_of宏

```
#container_of(ptr,type,member) ({\
        const typeof(((type *)0)->member) *__mptr=(ptr);\
        (TYPE*)((char*)__mptr-offsetof(type,member));})
```

两条语句，使用括号和大括号，\表示换行。
作用：知道一个结构体中某一个元素的指针，反推这个结构体变量的指针。继而可以得到结构体中其他元素的指针。
typeof关键字的作用是获得变量的类型。 原理：先用typeof得到member元素的类型定义成一个指针，然后用

这个指针减去该元素相对于整个结构体变量的偏移量，
偏移量通过offsetof获得，减去之后得到的就是整个结构体变量的首地址，再把这个地址强制转换类型陈TYPE*。