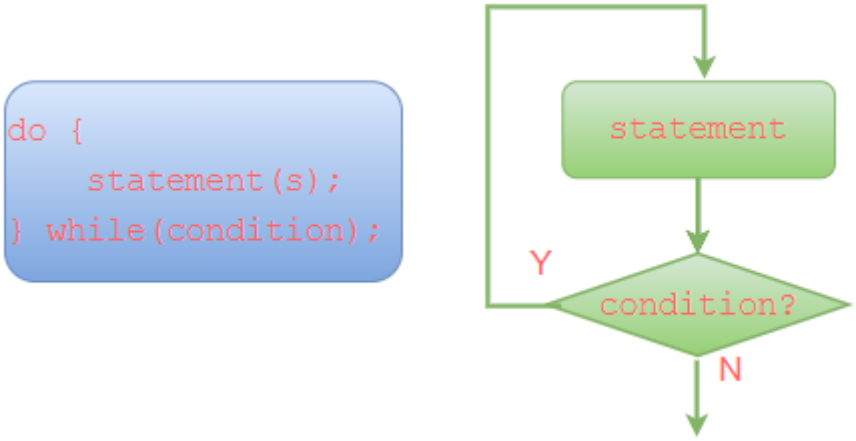
这是道哥的第010篇原创

前言

这篇文章讲解的知识点很小，但是在一些编程场合中非常适用，大家可以把这篇短文当做甜品来品味一下。

地球人都知道，do-while语句是C/C++中的一个循环语句，特点是:

> 至少执行一次循环体;
> 在循环的尾部进行结束条件的判断。



其实do-while还可以用在其他一些场合中，非常巧妙的处理你的一些难题，比如:

> 在宏定义中写复杂的语句;
> 在函数体中中止代码段的处理。

好像有点抽象，那我们就来具体一些，通过代码来聊聊这些用法。

也强烈建议您在平常的项目中把这些小技巧用起来，模仿是第一步，先僵化-再优化-最后固化，这是提高编程能力的最有效方法。
时间久了，用的多了，这些东西就是属于你的。

在宏定义中的妙用

错误的宏定义

```
// 目的：把两个参数分别自增一下
#define OPT(a, b)   a++; b++;

int main(int argc, char *argv[])
{
    int i = 1;
    int j = 1;
    OPT(i, j);
    printf("i = %d, j = %d \n", i, j);
    return 0;
}
```

测试一下，结果没有问题(代码的目的就是让i和j这个2个变量都自增1)：

```
i = 2, j = 2
```

而且`OPT(i, j);`中，最后的分号还可以省略，编译和结果都没有问题。

但是估计没有谁会在项目中这么使用宏吧？！看一下下面这个例子：
在调用OPT宏的外层添加一个if条件判断：

```
#define OPT(a, b)   a++; b++;

int main(int argc, char *argv[])
{
    int i = 1;
    int j = 1;
    if(0)
        OPT(i, j);
    printf("i = %d, j = %d \n", i, j);
    return 0;
}
```

打印结果是：

```
i = 1, j = 2
```

问题出现了：我们的本意是if条件为假，这2个变量都不要自增，但是输出结果却是：第二个参数自增了。

其实问题很明显，把宏扩展开就一目了然了。

```
if(0)
    a++; b++;
```

错误原因一目了然：由于if语句没有用大括号{}把需要执行的代码全部包裹住，导致只有`a++;`语句是在`if`语句的控制范围，而`b++;`语句无论如何都被执行了。

也许你会说，这个简单，使用`if`时，必须加上大括号{}。道理是没错，如果这个宏定义只有你自己使用，这不成问题。但是如果宏定义是你写的，而使用者是你的同事，那么你怎么要求别人必须按照你所规定的格式来编码？毕竟每个人的习惯是不一样的。

很多时候，要求别人是不现实的。更有效的方法是优化自己的输出，提供更安全的代码，让别人想犯错误都没机会。

比较好的宏定义

怎么做才能更安全？更通用呢？使用do-while！

```
#define OPT(a, b)   do{a++;b++;}while(0)
```

也就是说，只要宏定义中存在多条语句，就可以用do-while把这些语句全部包裹起来，这样无论怎么使用这个宏，都不会有问题。

例如：

```
if(0)
    OPT(i, j);
```

宏扩展之后代码为：

```
if(0)
    do {
        a++;
        b++;
    }while(0);
```

如果给if加上大括号，视觉上会更好一些：

```
if(0) {
    OPT(i, j);
}
```

宏扩展之后代码为：

```
if(0) {
    do {
        a++;
        b++;
    }while(0);
}
```

可以看到，无论是否加上大括号{}，从语法和语义上都不存在问题。

这里还有一个小细节可以留意一下：`OPT(i,j);`这行代码中，尾部是加了分号的。

如果没有加分号，那么宏扩展之后代码为：

```
if(0)
    do {
        a++;
        b++;
    }while(0) // 注意：这里没有分号
```

因为while(0)没有分号，所以编译会出错。为了不对宏的使用者提出要求，可以在宏的最后加一个分号即可，如下：

```
#define OPT(a, b)   do{a++;b++;}while(0);
```

小结：使用do-while语句来包裹宏定义中的多行语句，解决了宏定义的安全问题。

但是，任何事情都不可能是完美的，例如：在宏定义中使用do-while就无法返回一个结果。

也就是说：如果我们需要从宏定义中返回一个结果，那么do-while就派不上用场了。那应该怎么办？

另一个也不错的宏定义

如果宏定义需要返回一个结果，最好的方式就是：使用`({...})`把宏定义中的多行语句包裹起来。如下：

```
#define ADD(a, b, c) ({ ++a; ++b; c=a+b; })

int i = 1;
int j = 2;
int k;
printf("k = %d \n", ADD(i, j, k));
```

下面这张图来自GNU官方文档：

> A compound statement enclosed in parentheses may appear as an expression in GNU C. This allows you to use loops, switches, and local variables within an expression.
>
> Recall that a compound statement is a sequence of statements surrounded by braces; in this construct, parentheses go around the braces. For example:
>
> ```
> ({ int y = foo (); int z;
>    if (y > 0) z = y;
>    else z = - y;
>    z; })
> ```
>
> is a valid (though slightly more complex than necessary) expression for the absolute value of foo ()

翻译过来就是：

> GNU C中，在圆括号()中写复杂语句是合法的，这样你就可以在一个表达式中使用循环、switch、局部变量了。
> 什么是复杂语句呢？就是被大括号{}包裹的多行语句。
> 在上面的实例中，圆括号要放在大括号的外层。

使用`({...})`定义宏，因为是多行语句，可以返回一个结果，比do-while更胜一筹。

这里既然提到了在宏定义中使用局部变量，那我们再提供一个小技巧来提高代码的执行效率。

看一下这个宏定义：

```
#define max(a,b) ({ (a) > (b) ? (a) : (b) })

float i = 1.234;
float j = 4.321;
float max = max((i / 0.8 + 5) / 3, (j * 0.8) / 1.5);
```

宏扩展之后, a或者b中，肯定有一个被计算2次。当然，这里的示例比较简单，体现不出差距。如果是对时间要求特别苛刻的场合，计算量又很大，那么这个宏中由于两次计算所耗费的时间就必须考虑了，那应该如何优化呢？使用局部变量!

```
#define max(a,b)  ({ int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

通过增加局部变量_a和_b来缓存计算结果，就消除了2次计算的问题。

这个例子还可以再继续优化，这里的局部变量类型是int，这是写死的，只能比较两个整型的变量。如果写成这样：

```
#define max(a, b)  ({ typeof(a) _a = (a), _b = (b); _a > _b ? _a : _b; })
```

也就是用typeof来动态获取比较变量的类型，这样的话，任何数值类型的变量都可以使用这个宏了。

关于typeof的说明，请看GNU的这张图，在文末的参考链接中，可以看到更加详细的官方说明。

> ### 6.7 Referring to a Type with `typeof`
>
> Another way to refer to the type of an expression is with `typeof`. The syntax of using of this keyword looks like `sizeof`, but the construct acts semantically like a type name defined with `typedef`.
>
> There are two ways of writing the argument to `typeof`: with an expression or with a type. Here is an example with an expression:

在函数体中的妙用

先来看2段代码。

函数功能：返回错误代码对应的错误字符串

```
char *get_error_msg(int err_code)
{
    if (1 == err_code) {
        return "invalid name";
    } else if (2 == err_code) {
        return "invalid password";
    } else if (3 == err_code) {
        return "network error";
    }

    return "unkown error";
}
```

思考：一个设计良好的函数只有一个出口，也就是return语句，但是这个函数有这么多的return语句，是不是显得很乱？示例代码体积很小，似乎没有感觉。但是上百行的函数在项目中还是比较常见的，在这种情况下如果给你来个十几个return语句，你会不会想把写代码的那个家伙拎过来扇几巴掌？

函数功能：通过TCP Socket连接服务器

```
void connect_server(char *ip, int port)
{
    int ret, sockfd;
    sockfd = socket(...);
    if (sockfd < 0) {
        printf("socket create failed! \n");
        goto end;
    }

    ret = connect(sockfd, ...);
    if (ret < 0) {
        printf("connect failed! \n");
        goto end;
    }

    ret = send(sockfd, ...)
    if (ret < 0) {
        printf("send failed! \n");
        goto end;
    }

end:
```

```
    其他代码
}
```

思考：TCP socket编程中，需要按照固定的顺序调用多个系统函数。这段代码中调用系统函数后，对结果进行了检查，这是非常好的习惯。如果在某个调用中发生错误，需要中止后面的操作，进行错误处理。虽然C语言中不禁止goto语句的使用，但是看到这么多的goto，难道就没有美观、更优雅的做法吗？

总结一下上面这2段代码，它们共同的特点是：

> 在一连串的语句中，只需要执行一部分的语句，也就是从代码块的某个中间位置中止执行。

中止执行，我们首先想到的就是break关键字，它主要用在循环和switch语句中。
do-while循环语句首先执行循环体，在尾部才进行循环的判断。
那么就可以利用这一点来解决这2段代码面对的问题。

解决多个return的问题

```
char *get_error_msg(int err_code)
{
    char *msg;
    do {
        if (1 == err_code) {
            msg = "invalid name";
            break;
        } else if (2 == err_code) {
            msg = "invalid password";
            break;
        } else if (3 == err_code) {
            msg = "network error";
            break;
        } else {
            msg = "unkown error";
            break;
        }
    }while(0);

    return msg;
}
```

解决goto的问题

```
void connect_server(char *ip, int port)
{
    int ret, sockfd;
    do {
        sockfd = socket(...);
        if (sockfd < 0) {
            printf("socket create failed! \n");
            break;
        }

        ret = connect(sockfd, ...);
        if (ret < 0) {
            printf("connect failed! \n");
            break;
        }

        ret = send(sockfd, ...)
        if (ret < 0) {
            printf("send failed! \n");
            break;
        }
    }while(0);

    其他代码
}
```

这样的代码，是不是看起来顺眼多了？

总结

do-while的主要作用是循环处理，但是在这篇文章中，我们利用的点并不是循环功能，而是代码块的包裹和中止执行的功能。这些细小的点在一些牛逼的开源代码中很常见，看到了我们就要学习、模仿、使用，用的多了它就是你的了！

是不是开始喜欢上do-while语句了？

参考文档:

[1] https://gcc.gnu.org/onlinedocs/gcc/Typeof.html
[2] https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html
[3] https://stackoverflow.com/questions/9495962/why-use-do-while-0-in-macro-definition
[4] https://gcc.gnu.org/onlinedocs/gcc-6.2.0/gcc/Statement-Exprs.html#Statement-Exprs

【原创声明】

> 作者：道哥(公众号: IOT物联网小镇)
>
> 知乎：道哥
>
> B站：道哥分享
>
> 掘金：道哥分享
>
> CSDN：道哥分享

如果觉得文章不错，请转发、分享给您的朋友。

我会把十多年嵌入式开发中的项目实战经验进行总结、分享，相信不会让你失望的！

长按下图二维码关注，每篇文章都有干货。

<

推荐阅读

[1] 原来gdb的底层调试原理这么简单
[2] 生产者和消费者模式中的双缓冲技术
[3] 深入LUA脚本语言，让你彻底明白调试原理
[4] 一步步分析-如何用C实现面向对象编程