第 10 章C 预处理与 C 库

（视频讲解：1.5 小时）

C 的预处理指令有哪些，这些指令有什么作用，在什么场景下使用它们？C 语言是如何实现可变参数的？。学习本章，你将掌握：

- **预处理的指令及如何使用这些指令**
- **编写可变参数的函数**

10.1 预处理简介

C 预处理器是 C 语言、C++语言的预处理器。用于在编译器处理程序之前预扫描源代码，完成**头文件的包含,宏扩展,条件编译，行控制（line control）**等操作。

前面的几章用到过#define 与#include 指令，但没有深入讨论。这些指令（以及我们还没有学到的指令）都是由预处理器处理的。预处理器是一个小软件，它可以在编译前处理 C 程序。C 语言（和 C++语言）因为依赖预处理器而不同于其他的编程语言。

预处理器是一种强大的工具，但它同时可能是许多难以发现的错误的根源，此外，预处理也可能被用来错误的编写出一些难以读懂的程序,尽管有些 C 程序员非常依赖预处理器（嵌入式开发程序员），但个人建议适度使用它。

10.2 预处理的工作原理

预处理器的行为是由预处理指令（由#字符开头的一些命令）控制的。我们已经在前面的章节中遇见过其中两种指令，#include 和#define。

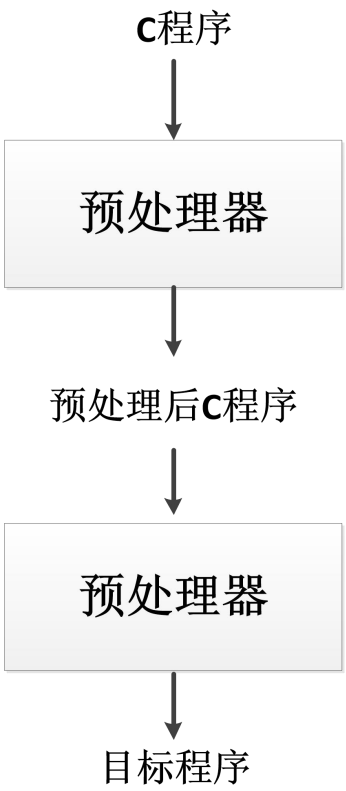
#define 指令定义了一个宏——用来代表其他东西的一个名字，例如常量（我们之前用的 PI）或常用的表达式（）。预处理器会通过将宏的名字和它的定义存储在一起来响应#define 指令。当这个宏在后面的程序中使用到时，预处理器“扩展”宏，将宏替换为其定义值。

#include 指令告诉预处理器打开一个特定的文件，将它的内容作为正在编译的文件的一部分“包含”进来。例如，代码行

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

指示预处理器打开一个名字为 Stdio.h 的文件，并将它的内容加到当前的程序中。（Stdio.h 包含了 C 语言标准输入/输出函数的原型。）

下图说明了预处理器在编译过程中的作用。预处理器的输入是一个 C 语言程序，程序可能包含预处理指令（带#号的指令）。预处理器会执行这些措令，并在处理过程中删除这些指令。预处理器的输出是另一个 C 程序：原程序编辑后的版本，不再包含指令（带#号的指令）。预处理器的输出被直接交给编译器，编译器检查程序是否有错误，并将程序翻译为目标代码（机器指令）。



首先我们编写如下的 main.c 程序

```
//展示预处理效果的代码
#include <stdio.h>
#include <stdlib.h>

#define cal(X) X*X

int main()
{
    printf("result=%d\n",cal(3+2));
    system("pause");
}
```

通过第2章的2.4.1我们讲解了如何得到预处理后的main.i文件，main.i文件如下图所示，main.c最上面的注释在预处理后会消失，#include <stdio.h>和#include <stdlib.h>头文件都会被展开，图中可以看到这两个头文件被展开后总结11560行，预处理后的文件以#号开头的代表是注释，11560和int main()之间有三个空行，是因为预处理后空行不会消失，同时#define开头的会被替换为空行，通过图中的预处理的替换效果可以得知printf最终的打印不是25，而是11，因此会看到实际大家的#define会写为 #define cal(X) (X)*(X)，如果写成这样，最终得到的值即为25。

```
11556   #pragma pack(pop)
11557
11558   #line 933 "c:\\program files (x86)\\microsoft visual studio 11.0\\vc\\include\\stdlib.h"
11559
11560   #line 4 "e:\\code_30\\预处理及c库\\1.预处理基础\\main.c"
11561
11562
11563
11564   int main()
11565   {
11566       printf("result=%d\n",3+2*3+2);
11567       system("pause");
11568   }
11569
```

在 C 语言较早的时期，预处理器是一个单独的程序，它的输出提供给编译器。如今，预处理器通常和编译器集成在一起，然而，将预处理器和编译器认为足不同的程序仍然是有用的。实际上，大部分 C 编译器都提供了一种方法，使用户可以看到预处理器的输出。再指定某个特定的选项(GCC 编译器用的是-E)时编译器会产生预处理器的输出。其他一些编译器会提供一个类似于集成的预处理器的独立程序。

注意，预处理仅解析少量的 C 语言规则，因此，它在执行指令时非常有可能产生非法的程序。经常是源程序看起来没问题，使错误查起来很难。对于较复杂的程序，检查预处理的输出可能是找到这类错误的有效途径。

10.3 预处理指令

大多数预处理指令都属于下面 3 种类型之一。

•宏定义。#define 桁令定义一个宏，#undef 指令删除一个宏定义。

•文件包含。#include 指令导致-个指定文件的内界被包含到程序中。

•条件编译。#if、#ifdef、#ifndef #elif、#else 和#endif 指令可以让预处理器以测试的条件来确定是将一段文本块包含到程序内还是将其排除在程序之外。

剩下的#error、#line 指令是更特殊的指令，较少用到。本章将深入研究预处理指令。

在进一步讨论之前，先来看几条适用于所有指令的规则。

•指令都以#开始。#符号不需要在一行的行首，只要它之前只有空白字符就行。在#后是指令名，接着是指令所需要的其他信息，

•在指令的符号之间可以插入任意数量的空格或水平制表符。例如，下面的指令是合法的：

```
#     define    N   100
```

•指令总是在第一个换行符处结束，除非明确地指明要延续。如果想在下一行延续桁令，我们必须在当前行的末尾使用\字符。例如，下面的指令定义了一个宏来表示硬盘的容量，按字节计算：

```
#define DISK_CAPACITY  (SIDES * \
                        TRACKS_PER_SIDE * \
                        SECTORS_PER_TRACK *   \
                        BYTES_PER_SECTOR)
```

上面宏的含义是 磁盘容量=盘面*每个盘面的磁道数目*每个磁道的扇区数目*每个扇区的字节数

•指令可以出现在程序中的任何地方。但我们通常将#define 和#include 指令放在文件的开始，其他指令则放在后面，甚至可以放在函数定义的中间。

•注释可以与指令放在同一行。实际上，在宏定义的后面加一个注释来解释宏的含义是一种

比较好的习惯：

#define PI 3.1415f /*圆周率精度设置 */

•**作用域**。其作用域为宏定义命令起到源程序结束。

•**宏名在源程序若用引号括起来，则预处理程序不对其作宏替换**。

•**宏定义允许嵌套**、在宏定义的字符串中可以使用已经定义的宏名。在宏展开时由预处理程序层层替换。

10.4 宏定义

10.4.1 简单的宏

简单的宏，就是没有参数的宏（C 标准中称为对象式宏）的定义有如下格式：

[#define 指令（简单的宏）] 即 #define 标识符 替换列表

替换列表是一系列的预处理记号。本章中我们提及“记号”时，均指的是“预处理记号”。

宏的替换列表可以包括标识符、关键字、数值常量、字符常量、字符串字面量、操作符和排列。当预处理器遇到一个宏定义时，会做一个“标识符”代表“替换列表”的记录.在文件后面的内容中，不管标识符在哪里出现，预处理器都会用替换列表代替它。

不要在宏定义中放咒任何额外的符号，否则它们会被作为替换列衣的一部分。

一种常见的错误是在宏定义中使用=:

#define N = 100 / *错误使用*/

int a[N]; /* 实际预处理后会变为 int a[= 100]；造成编译不通 */

在上面的例子中，我们（错误地）把 N 定义成两个记号（=和 100）。

在宏定义的末尾使用分号结尾是另一个常见错误：

#define N 100; /*错误使用*/

int a[N]; /*实际预处理后会变为 int a[100;]；造成编译不通*/

这里 N 被定义为 100 和;两个记号。

编译器可以检测到宏定义中绝大多数由多余符号所导致的错误。但是，编译器只会将每一个使用这个宏的地方标为错误，而不会直接找到错误的根源一宏定义本身，因为宏定义已经被预处理器删除了，所以这时我们发现某一行含有宏编译错误时，需要看一下宏定义本身，更进一步，就是看看预处理后的文件。

简单的宏主要用来定义那些被 Kemighan 和 Ritchie 称为“明示常量（manifest constant）的东西，我们可以使用宏给数字、字符值和字符串值命名。

```
#define STE_LEN 80
#define TRUE    1
#define FALSE   0
#define PI    3.14159
#define MEM_ERR “Error: not enough memory”
```

使用#define 来为常量命名许多显著的优点。

•程序会更易读。一个认真选择的名字可以帮助读者理解常量的意义。否则，程序将包含大量的“魔法数”，很容秘迷惑读者。

•程序会更易于修改。我们仅需要改变一个宏定义，就可以改变整个程序中出现的所有该常量的值。“硬编码的”常量会更难于修改，特别是当它们以稍微不同的形式出现时。

（例如，如果程序包含一个长度为 100 的数组，它可能会包含一个从 0 到 99 的循环。如果我们只是试阁找到程序中出现的所有 100,那么就会漏掉 99。）

•可以帮助避免前后不一致或键盘输入错误。假如数值常量 3.14159 在程序中大量出现，它可能会被意外地写成 3.1416 或 3.14195。

虽然简单的宏常用于定义常量名，但是它们还有其他应用。

•可以对 C 语法做小的修改。我们可以通过定义宏的方式给 C 语芑符号添加别名，从而改变 C 语言的语法。例如，对于习惯使用 Pascal 的 begin 和 end （而不是 C 语言的{和}）的程序员，可以定义下面的宏：

```
#define BEGIN {
#define END }
```

我们甚至可以发明自己的语言。例如，我们可以创造-个 LOOP “语句”，来实现一个无限循环：

```
#define LOOP for （;;）
```

当然，改变 C 语言的语法通常不是个好主意，因为它会使程序很难被其他程序员理解。

•对类型重命名。我们通过重命名 int 创让了一个布尔类型：

```
#define BOOL int
```

虽然有些程序员会使用宏定义的方式来实现此目的，但类型定义通过 7.1.3 的 typedef 仍然是定义新类型的最佳方法。

•控制条件编译。宏在控制条件编译中起重要的作用。例如，在程序中出现的下面这行宏定义时表明将程序在“调试模式”下进行编译，并使用额外的语句输出调试信息：

```
#define DEBUG
```

这里顺便提-下，如上面的例子所示，宏定义中的替换列表为空是合法的。

当宏作为常量使用时，C 程序员习惯在名字中只使用大写字母。但是并没有如何将用于其他目的的宏大写的统一做法。由于宏(特别是带参数的宏)可能是程序中错误的来源，所以一些程序员更喜欢全部使用大写字母来引起注意。有些人则倾向于小写，即按照 Kemighan 和 Ritchie 编写的 The C Programming Language 一书中的风格。

```
 //DEBUG开关的使用
#include <stdio.h>
#include <stdlib.h>

#define DEBUG
int main()
{
#ifdef DEBUG
    printf("这是调试信息\n");
#endif
    system("pause");
}
```

上面代码得到的预处理后的 main.i 文件效果如下图：

```
11560   #line 4 "e:\\code_30\\预处理及c库\\2.debug开关\\main.c"
11561
11562
11563   int main()
11564   {
11565
11566       printf("这是调试信息\n");
11567   #line 11 "e:\\code_30\\预处理及c库\\2.debug开关\\main.c"
11568       system("pause");
11569   }
```

如果去除#define DEBUG，得到的预处理后的main.i文件如下图所示，通过DEBUG调试开关的手法，我们可以在开发时，将一些打印调试信息包含在DEBUG开关内，但是当编译发布到生产环境时，我们需要除去DEBUG开关，因为打印调试会影响程序执行效率，但是生产环境执行出现了异常，我们需要定位问题时，这时我们就可以打开DEBUG开关，重新编译，这样所有关键节点的信息可以快速掌握，可以极大的提高定位问题的效率。

```
11560   #line 4 "e:\\code_30\\预处理及c库\\2.debug开关\\main.c"
11561
11562
11563   int main()
11564   {
11565
11566
11567
11568       system("pause");
11569   }
```

10.4.2 带参数的宏

带参数的宏(也称为函数式宏)的定义有如下格式：

[#define 指令(带参数的宏)] #define　标识符(x1,x2,…, xn)　替换列表

其中 x1,x2,…, xn 是标识符(宏的参数)。这些参数可以在替换列表中根据需要出现任意次。

注意：**在宏的名字和左括号之间必须没有空格**。如果有空格，预处理器会认为定义了一个简单宏。

当预处理器遇到带参数的宏时，会将宏定义存储起来以便后面使用。在后面的程序中，如果任何地方出现了标识符 x1,x2,…, xn)格式的宏调用，预处理器会使用替换列表一一替代。

例如，假定我们定义了如下的宏：

```
#define MAX(x,y)    ((x)>(y)?(x): (y))
#define IS_EVEN(n) ((n)%2==0)
```

(宏定义中的圆括号似乎过多，但下面看到，这样做是有原因的。)现在如果后面的程序中有如下语句：

```
i = MAX(j+k, m-n);
IS_EVEN(i)+3;
```

预处理器会将这些行替换为

```
i = ((j+k)>(m-n)?(j+k):(m-n)); //如果没有括号，会发现表达式的计算不正确
((i)%2==0)+3;//如果没有括号，将会导致(i)%2==0+3，这样无论 i 是偶数还是奇数，这个表达式整体值始终为零
```

如这个例子所示，带参数的宏经常用来作为简单的函数使用。MAX 类似一个从两个值中选取较大值的函数，IS_EVEN 则类似于一种当参数为偶数时返回 1,否则返回 0 的函数。

下面的宏也类似于函数，但更为复杂：

```
#define   TOUPPER(c)   ('a'<=(c)&&(c)<='z'?(c)-'a'+'A':(c))
```

这个宏检测字符 c 足否在'a'与'z'之间。如果在的话,这个宏会用 C 的值减去再加上'A',从而计算出 c 所对应的大写字母，如果 c 不在这个范围，就保留原来的 c。

使用带参数的宏替代真正的函数有两个优点。

•程序可能会稍微快些。程序执行时调用函数通常会有些额外开销一存储上下文信息、复制参数的值等，而调用宏则没行这些运行开销。(注意，C99 的内联函数为我们提供了一种不使用宏而避免这一开销的办法。)

•宏更“通用”。与函数的参数不同，宏的参数没有类型。因此，只要预处理后的程序依然是合法的，宏可以接受任何类型的参数。例如，我们可以使用 MAX 宏从两个数中选出较大的一个，数的类型可以 int、long、float、double 等。

但是带参数的宏也有一些缺点。

•编译后的代码通常会变大。每一处宏调用都会导致插入宏的替换列表，由此导致程序的源代码增加(因此编译后得到的可执行程序变大)。宏使用得越频繁，这种效果就越明显。当宏调用嵌套时，这个问题会相互叠加从而使秤序更加复杂。思考一下，如果我们用 MAX 宏來找出 3 个数中最大的数会怎样？

```
n = MAX(i, MAX(j, k));
```

下面是预处理后的语句：

```
n = ((i)>(((j)>k)?(j):(k)))?(i):(((j)>(k)?(j):(k))));
```

•宏参数没有类型检查。当一个函数被调用时，编译器会检查每一个参数來确认它们是否 S 正确的类型。如果不是，要么将参数转换成正确的类型，要么由编译器产生一条出错消息。预处理器不会检查宏参数的类型，也不会进行类型转换。

•无法用一个指针来指向一个宏。如在 5.5 节中将看到的，C 语言允许指针指向函数，这在特定的编程条件下非常有用。宏会在预处理过程中被删除，所以不存在类似的“指向宏的指针”。因此，宏不能用于处理这些情况。

•宏可能会不止一次地计算它的参数。函数对它的参数只会计算一次，而宏可能会计算两次甚至多次。如果参数有副作用，多次计算参数的值可能会产生不可预知的结果。考虑下面的例子，其中 MAX 的一个参数有副作用：

```
n = MAX(i++, j);
```

下面是这条语句在预处理之后的结果：

```
n = ((i++)>(j)?(i++):(j));
```

如果 i 大于 j，那么 i 可能会被(错误地)增加两次，同时 n 可能被赋予错误的值。

注意：由于多次计算宏的参数而导致的错误可能非常难于发现，因为宏调用和函数调用看起來是一样的。更糟糕的是，这类宏可能在大多数情况下可以正常工作，仅在特定参数有副作用时失效。为了自我保护，最好避免使用带有副作用的参数。

带参数的宏不仅适用于模拟函数调用，还经常用作需要重复书写的代码段模式。如果我们已经写烦了语句

```
printf("%d\n", i);
```

因为每次要显示一个整数 i 都要使用它，我们可以定义下面的宏，使显示整数变得简单些：

```
#define PRINT_INT(n)     printf("%d\n", n)
```

一旦使用 PRINT_INT,预处理器会将这行

```
PRINT_INT(i+j)
```

转换为

```
printf("%d\n", i+j);
```

10.4.3 操作符#

宏定义可以包含两个专用的运算符：#和##。编译器不会识别这两种运算符，它们会在预处理时被执行。

#运算符将宏的一个参数转换为字符串字面量。它仅允许出现在带参数的宏的替换列表中。（#运算符所执行的操作可以理解为“字符串化(stringization)”）。

#运算符有许多用途，这里只来讨论其中的一种。假设我们决定在调试过程中使用PRINT_INT 宏作为一个便捷的方法来输出整型变量或表达式的值。#运算符可以使 PRINT_INT 为每个输出的值添加标签。下面是改进后的 PRINT_INT:

```
#define PRINT_INT(n)  printf(#n " = %d\n", n)
```

n 之前的#运算符通知预处理器根据 PRINT_INT 的参数创建一个字符串字面量。因此，调用

```
PRINT_INT(i+j);
```

会变为

```
printf("i+j" " = %d\n", i+j);
```

C 语言中相邻的字符串字面贵会被合并。因此上边的语句等价于：

```
printf("i+j = %d\n", i+j);
```

当程序执行时，printf 函数会同时显示表达式 i+j 和它的值。例如，如果 i 是 5, j 是 6 的话，输出为 i+j =11

10.4.4 操作符##

与操作符#类似，操作符##也可用在带参宏中替换部分内容。该操作符将宏中的两个部分连接成一个内容。例如，定义如下宏：

```
#define VAR(n)      v##n
```

当使用一下方式引用宏：

```
VAR(1)
```

预处理时，将得到以下形式：

```
V1
```

如果使用以下宏定义：

```
#define FUNC(n)    oper##n
```

当实参为 1 时，预处理后得到一下形式：

```
oper1
```

##运算符不属于预处理器最经常使用的特性。实际上想找到一些使用它的情况是比较困难的。为了找到一个有实际意义的##的应用，我们来重新思考前面提到过的 MAX 宏。如我们所见，当 MAX 的参数有副作用时会无法正常工作。一种解决方法是用 MAX 宏来写一个 max 函数。迪憾的是，仅一个 max 函数是不够的，我们可能需要一个实际参数是 int 值的 max 函数、一个实际参数为 float 值的 max 函数，等等。除了实际参数的类型和返回值的类型之外，这些函数的其他部分都一样。因此，这样定义每一个函数似乎是个很蠢的做法。

解决的办法是定义一个宏，并使它展开后成为 max 函数的定义。宏只有一个参数 type，表示实际参数和返回值的类型。这里还有个问题，如果我们用宏来创建多个 max 函数，程序将无法编译。(C 语言不允许在同一文件中出现两个同名的函数。)为了解决这个问题，我们用##运算符为每个版本的 max 函数构造不同的名字。下面是宏的形式：

```
#define GENERIC_MAX(type)   \
```

```
type type##_max(type x, type y) \
{                          \
    return x > y ? x : y;   \
}
```

注意，宏的定义中是如何将 type 和_max 相连来形成新函数名的。

现在，假如我们需要一个针对 float 值的 max 函数。下面记使用 GENERIC_MAX 宏來定义这一函数的方法：

GENERIC_MAX(float)

预处理器会将这行展开为下面的代码：

float float_max(float x, float y) (return x > y ? x : y; }

如果是

GENERIC_MAX(int)

预处理器会将这行展开为下面的代码：

int int_max(int x, int y) (return x > y ? x : y; }

10.4.5 宏的通用属性

现在我们已经讨论过了简单的宏和带参数的宏，我们来看一下它们都要遵守的规则。

•宏的替换列表可以包含对其他宏的调用。例如，我们可以用宏 PI 来定义宏 TWO_PI：

#define PI 3.14159

#define TWO_PI (2*PI)

当预处理器在后面的程序中遇到 TW0_PI 时，会将它替换成(2*PI〉。接着，预处理器会重新检查替换列表，看它是否包含其他宏的调用(在这个例子中，调用了宏 PI)。

预处理器会不断重新检查替换列表，直到将所有的宏名字都替换掉为止。

•预处理器只会替换完整的记号，而不会替换记号的片断。此外，预处理器会忽略嵌在标识符、字符常量、字符串字面量之中的宏名。例如，假设程序含有如下代码行：

#define SIZE 256

int BUFFER_SIZE;

if (BUFFER_SIZE > SIZE)

puts ("Error : SIZE exceeded");

预处理后这些代码行会变为

int BUFFER_SIZE;

if (BUFFER_SIZE > 256)

puts("Error : SIZE exceeded");

尽管标识符 BUFFER_ZISE 和字符串常量"Error : SIZE exceeded"都包含 SIZE,但是它们没有被预处理影响。

•宏不可以被定义两遍，除非新的定义与旧的定义是一样的。小的间隔上的差异是允许的，但是宏的替换列表(和参数，如果有的话)中的记号都必须一致。

•宏可以使用#undef 指令“取消定义”。#undef 指令有如下形式：

[#undef 指令] #undef 标识符

其中标识符是一个宏名。例如，衍令

#undef N

会删除宏 N 当前的定义。(如果 N 没有被定义成一个宏，#undef 指令没有任何作用。)

10.4.6 较长的宏中的逗号运算符

在创建较长的宏时，逗号运算符会十分有用。特别是可以使用逗号运算符来使替换列表包含一系列表达式。例如，下面的宏会读入一个字符串，再把字符串显示出来：

#define ECHO(s) (get(s), puts(s))

gets 函数和 puts 函数的调用都是表达式，因此使用逗号运算符连接它们是合法的。我们甚至可以把 ECHO 宏当作一个函数来使用：

```
ECHO(str);    /* 替换为 (gets(str), puts(str)); */
```

除了使用逗号运算符，我们也许还可以将 gets 函数和 puts 函数的调用放在大括号中形成复合语句：

```
#define ECHO(s)  { gets(s);  puts(s);  }
```

遗憾的是，这种方式在下面的情况并不奏效。假如我们将 ECHO 宏用于下面的 if 语句：

```
if (echo_flag)
  ECHO(str);
else
  gets(str);
//将 ECHO 宏替换会得到下面的结果：
if (echo_flag)
  { gets(str); puts(str); };
else
  gets(str);
```

编译器会将头两行作为完整的 if 语句：

```
if (echo_flag)
  { gets(str);  puts(str); };
```

编译器会将跟在后面的分号作为空语句，并且对 else 子句产生出错信息，因为它不属于任何 if 语句。我们可以通过记住永远不要在 ECHO 宏后面加分号来解决这个问题。但是这样做会使程序看起来有些怪异。逗号运算符可以解决 ECHO 宏的问题，但并不能解决所有宏的问题。假如一个宏需要包含一系列的语句，而不仅仅是一系列的表达式，这时逗号运算符就起不到帮助的作用了。因为它只能连接表达式，不能连接语句。解决的方法是将语句放在 do while 循环中，并将条件设置为假。

10.4.7 宏定义中的 do-while 循环

do 循环必须始终随跟着一个分号，因此我们不会遇到在 if 语句中使用宏那样的问题了。为了看到这个技巧（嗯，应该说是技术）的实际作用，让我们将它用于 ECHO 宏中：

```
#define ECHO(s)          \
        do{              \
            gets (s) ;   \
            puts (s) ;   \
        } while  (0)
```

当使用 ECHO 宏时，一定要加分号：

```
ECHO(str);
  /* becomes do {  gets(str); puts(str); } while (0);  */
```

这样 10.4.6 小节中的 if 和 else 语句将不会报错

为什么在宏定义时需要使用 do-while 语句呢？我们知道 do-while 循环语句是先执行循环体再判断条件是否成立， 所以说至少会执行一次。当使用 do{ }while(0)时由于条件肯定为 false，代码也肯定只执行一次，肯定只执行一次的代码为什么要放在 do-while 语句里呢？这种方式适用于宏定义中存在多语句的情况。 如下所示代码：

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

if (expr)
    TEST(a, b);
else
    do_else();
```

代码进行预处理后，会变成：

```
if (expr)
    a++;b++;
else
    do_else();
```

这样 if-else 的结构就被破坏了 if 后面有两个语句，这样是无法编译通过的，那为什么非要 do-while 而不是简单的用{}括起来呢。 这样也能保证 if 后面只有一个语句。例如上面的例子，在调用宏 TEST 的时候后面加了一个分号， 虽然这个分号可有可无， 但是出于习惯我们一般都会写上。 那如果是把宏里的代码用{}括起来，加上最后的那个分号。 还是不能通过编译。所以一般的多表达式宏定义中都采用 do-while(0)的方式。

10.4.8 预定义宏

在 C 语言中预定义了一些有用的宏，见**表预定义宏**。这些宏主要是提供当前编译的信息。宏__LINE__和__STDC__是整型常量，其他 3 个宏是字符串常量。

表预定义宏:

名字	含义
__LINE__	被编译的代码在对应文件中行数
__FILE__	被编译的文件的名字
__DATE__	编译的日期（格式"mm dd yyyy"）
__TIME__	编译的时间（格式"hh:mm:ss"）
__STDC__	如果编译器接受标准 C，那么值为 1（微软的 VisualStudio 里没有，gcc 支持）

1、__DATE__宏和__TIME__宏指明程序编译的时间。例如，假设程序以下面的语句开始：

```
printf("Compiled on %s at %s\n", __DATE__,__TIME__);
```

每次程序开始执行，程序都会显示下面一行：

```
Compiled on Jul 23 2019 at 22:18:48
```

这样的信息可以帮助区分同一个程序的不同版本。

2、我们可以使用__LINE__宏和__FILE__宏来找到错误。考虑下面这个检测被零除的除法的发生位置的问题。当一个 C 程序因为被零除而导致中止时，通常没有信息指明哪条除法运算导致错误。下面的宏可以帮助我们查明错误的根源：

```
#define CHECK_ZERO(DIVISOR)    \
   if (DIVISOR == 0)    \
      printf("*** Attempt to divide byzero on line %d    "    \
```

```
                    "of file %s    ***\n",__LINE__, __FILE__)
```

CHECK_ZERO 宏应该在除法运算前被调用：

```
CHECK_ZERO(j);
k = i / j;
```

如果 j 是 0，会显示出如下形式的信息：

```
*** Attempt to divide by zero on line 9 of file FOO.c ***
```

类似这样的错误检测的宏非常有用。实际上，C 语言库提供了一个通用的、用于错误检测的宏——assert 宏

再如：

```
#line 838 "Zend/zend_language_scanner.c"
```

#line 预处理用于改变当前的行号（__LINE__）和文件名（__FILE__）。 如上所示代码，将当前的行号改变为 838，文件名 Zend/zend_language_scanner.c 它的作用体现在编译器的编写中，我们知道

编译器对 C 源码编译过程中会产生一些中间文件，通过这条指令，可以保证文件名是固定的，不会被这些中间文件代替，有利于进行调试分析。

10.4.9 空宏参数

C99 允许宏调用中的任意或所有参数为空。当然这样的调用需要有和一般调用一样多的逗号(这样容易看出哪些参数被省略了)。

在大多数情况下，实际参数为空的效果是显而易见的。如果替换列表中出现相应的形式参数名，那么只要在替换列表中不出现实际参数即可，不需要作替换。例如：

```
#define ADD(x,y) (x+y)
```

经过预处理之后，语句

```
i = ADD(j,k);
```

变成

```
i = (j+k);
```

而赋值语句

```
i = ADD( , k);
```

则变为

```
i = (+k);
```

当空参数是#或##运算符的操作数时，用法有特殊规定。如果空的实际参数被#运算符“字符串化”，则结果为""(空字符串)：

```
标 define MK_STR(x)    #x
char empty_string[]=MK_STR();
```

预处理之后，上面的声明变成：

```
char empty_string[] ="";
```

如果##运算符之后的-个实际参数为空，它将会被不可见的“位置标记”记号代替。把原始的记号与位置标记记号相连接，得到的还是原始的记号(位置标记记号消失了)。如果连接两个位置标记记号，得到的是一个位置标记记号。宏扩展完成后，位置标记记号从程序中消失。考虑下面的例子：

```
#define JOIN(x,y, z)    x##y##z
int JOIN(a,b,c)f JOIN(a,b,), JOIN(a,,c), JOIN(,,c);
```

预处理之后，声明变成：

int abc, ab, ac, c；
漏掉的参数由位置标记记号代替，这些记号在与非空参数相连接之后消失。JOIN 宏的 3 个参数可以同时为空，这样得到的结果为空。

10.5 条件编译

10.5.1 #if 指令和#endif 指令

假如我们正在调试一个程序。我们想要程序显示出特定变量的值，因此将 printf 函数调用添加到程序中重要的部分。一旦找到错误，经常需要保留这些 printf 函数调用，以备以后使用。
条件编译允许我们保留这些调用，但是让编译器忽略它们。
下面是我们需要采取的方式。首先定义一个宏，并给它一个非零的值：

```
#define DEBUG 1
```

宏的名字并不重要。接下来，我们要在每组 printf 函数调用的前后加上#if 和#endif：

```
#if DEBUG
    printf("debug info i=%d\n",i);
    printf("debug info j=%d\n",j);
#endif
```

在预处理过程中，#if 指令会测试 DEBUG 的值。由于 DEBUG 的值不是 0,因此预处理器会将这两个 printf 函数调用保留在程序中（但#if 和#endif 行会消失）。如果我们将 DEBUG 的值改为 0 并重新编译程序，预处理器则会将这 4 行代码都删除。编译器不会看到这些 printf 函数调用，所以这些调用就不会在目标代码中占用空间，也不会在程序运行时消耗时间。我们可以将#if 和#endif 保留在最终的程序中，这样如果程序在运行时出现问题，可以（通过将 DEBUG 改为 1 并重新编译来〉继续产生诊断信息。
一般来说，#if 指令的格式如下：
[#if 指令]　#if 常量表达式
#endif 指令则更简单：
[#endif 指令]　#endif
当预处理器遇到#if 指令时，会计算常量表达式的值。如果表达式的值为 0,那么#if 与#endif 之间的行将在预处理过程中从程序中删除：否则，#if 和#endif 之间的行会被保留在程序中，继续留给编译器处理——这时#if 和#endif 对程序没有任何影响。
值得注意的是，#if 指令会把没有定义过的标识符当作是值为 0 的宏对待。因此，如果没有定义 DEBUG，对应#if 与#endif 包含的内容将不会被编译，如果省略#if 后的 DEBUG，测试

```
#if
```

会提示编译失败，提示**无效的整数常量表达式**，而测试#if　!DEBUG 会成功。同时#if 必须和#endif 配对出现，如果没有#if 会出现　**意外的 #endif 报错**

10.5.2 defined 运算符

介绍过运算符#和##,还有一个专用于预处理器的运算 一一 defined。当 defined 应用于标识符时，如果标识符是一个定义过的宏则返回 1,否则返回 0。defined 运算符通常与#if 指令结合使用，可以这样写

```
#if defined（DEBUG）
代码
#endif
```

仅当 DEBUG 被定义成宏时，#if 与#endif 之间的代码会被保留在程序众。DEBUG 两侧的括号不是必须的，因此可以简单写成
#if defined DEBUG
由于#defined 运算符仅检测 DEBUG 是否有定义，所以不需要给 DEBUG 赋值，只需写为#define DEBUG 即可。

10.5.3 #ifdef 指令和#ifndef 指令

#ifdef 指令测试一个标识符是否已经定义为宏:
[#ifdef 指令]　#ifdef 标识符
#ifdef 指令的使用与#if 指令类似：
#ifdef　　标识符
当标识符被定义为宏时需要包含的代码
#endif
严格地说,
并不需要因为可以结合#if 指令和 defined 运算符来得到相同的
效果。换言之，桁令
#ifdef 标识符
等价于
#if defined（标识符）
#ifndef 指令与#ifdef 指令类似，但测试的是标识符是否没有被定义为宏:
[#ifndef 指令]　#ifndef 标识符
指令
#ifndef 标识符
等价于指令
#if　!defined（标识符）

为了避免同一个文件被 include 多次，可以使用以下两种方式来处理：
1　#ifndef 方式
2　#pragma once 方式

在能够支持这两种方式的编译器上，二者并没有太大的区别，但是仍然还是有一些细微的区别：

方式一：
```
#ifndef __HEAD_H__
#define __HEAD_H__
… … // 一些声明语句
#endif
```

方式二：
```
#pragma once
… … // 一些声明语句
```

#ifndef 的方式：依赖于宏名字不能冲突，这不光可以保证同一个文件不会被包含多次，也能保证内容完全相同的两个文件不会被不小心同时包含。当然，缺点就是如果不同头文件

的宏名不小心“撞车”，可能就会导致头文件明明存在，编译器却硬说找不到声明的状况

#pragma once 方式：则由编译器提供保证：同一个文件不会被包含多次。注意这里所说的“同一个文件”是指物理上的一个文件，而不是指内容相同的两个文件。带来的好处是，你不必再费劲想个宏名了，当然也就不会出现宏名碰撞引发的奇怪问题。对应的缺点就是如果某个头文件有多份拷贝，本方法不能保证他们不被重复包含。当然，相比宏名碰撞引发的“找不到声明”的问题，重复包含更容易被发现并修正。

方式一 由语言支持所以移植性好，方式二 可以避免名字冲突。

下面有一个例子，**当我们定义两个相同的结构体类型，就会编译报错，在 VS 的报错是 error C2011: “MyStruct”:“struct”类型重定义**，因为往往结构体类型定义是放在头文件的，如果 a.h 包含 b.h 和 c.h，b.h 包含 c.h，就会造成 a.h 包含了两次 c.h，就会发生头文件的重复包含

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

typedef struct MyStruct
{
    int num;
}stu;

typedef struct MyStruct
{
    int num;
}stu;
```

10.5.4 #elif 指令和#else 指令

#if 指令、#ifdef 指令和#ifndef 桁令可以像普通的 if 语句那样嵌套使用。当发生嵌套时，最好随着嵌套层次的增加而增加缩进。有些程序员对每一个#endif 都加注释，来说明每个对应的#if 指令对应测试哪个条件：

```
#if DEBUG
#endif      /* DEBUG •/
```

这种方法有助于更方便地找到#if 指令的起始位置。

为了提供更多的便利，预处理器还支持#elif 和#else 指令：

```
[#elif]   使用方法   #elif  常量表达式
[#else  指令]          #else
```

#elif 指令和#else 指令可以与#if 指令、#ifdef 指令#ifndef 指令结合使用，来测试一系列条件：

```
#if 表达式 1
当表达式 1 非 0 时需要包含的代码
#elif 表达式 2
当表达式 1 为 0 表达式 2 非 0 时需要包含的代码
#else
其他情况下需要包含的代码
#endif
```

虽然上面的例子使用#if 指令，但#ifdef 指令或#ifndef 指令也可以这样使用。在#if 指

令和#endif 指令之间可以有任意多个#elif 指令，但最多只能有一个#else 指令。

10.5.5 条件编译

条件编译对于调试是非常方便的，但它的应用并不仅限于此。下面是其他一些常见的应用：

•编写在多台机器或多种操作系统之间可移植的程序。下面的例子中会根据 WIN32. MAC_OS 或 LINUX 是否被定义为宏，而将三组代码之一包含到程序中：

```
#if defined(WIN32)
#elif defined(MAC_OS)
。。。。。
#elif defined(LINUX)
。。。。。
#endif
```

一个程序中可以包含许多这样的#if 衍令。在程序的开头会定义这些宏之一(而且只有一个)，由此选择了一个特定的操作系统。例如，定义 LINUX 宏可以指明程序将运行在 Linux 操作系统下。

•编写可以用不同的编译器编译的程序。不同的编译器可以用于识别不同的 C 语言版本，这些版本之间会有一些差异。一些会接受标准 C，另外一些则不会。一些版本会提供针对特定机器的语言扩展；有些版本则没有，或者提供不同的扩展集。条件编译可以使程序适应于不同的编译器。考虑一下为以前的非标准编译器编写程序的问题。__STDC__ 宏允许预处理器检测编译器是否支持标准(C89 或 C99);如果不支持，我们可能必须修改程序的某些方面，尤其是有可能必须用老式的函数声明。对于每一处函数声明，我们可以使用下面的代码：

```
#if __STDC__
函数原型
#else
老式的函数声明
#endif
```

做后台开发，使用多种编译器的概率较低，只有使用多种编译器时，才需要使用到该手法。

•为宏提供默认定义。条件编译使我们可以检测一个宏当前是否已经被定义了，如果没有，则提供一个默认的定义。例如，如果宏 BUFFER_SIZE 此前没有被定义的话，下面的代码会给出定义：

```
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 256
#endif
```

•临时屏蔽包含注释的代码。我们不能用/*...*/直接“注释掉”已经包含/*...*/注释的代码。然而，我们可以使用#if 指令来实现：

```
#if 0
包含注释的代码行
#endif
```

将代码以这种方式屏蔽掉经常称为“条件屏敝”。

10.6 其他指令

10.6.1 指令#line

更改预处理器中的当前行号和文件名。

语法

#line 行号 (1)

#line 行号 "文件名" (2)

解释

1) 将当前预处理器行号更改为 行号。此点之后，宏 __LINE__ 的展开将产生 行号 加上自此遇到的实际代码行数。

2) 还将当前预处理器文件名更改为 文件名。此点之后，宏 __FILE__ 的展开将生成 文件名。

任何预处理器记号（宏常量或表达式）都允许作为 #line 的实参，只要它们展开成合法的十进制整数，可选地后随一个合法的字符串即可。

行号 必须是至少有一个十进制位的序列（否则程序会出现编译错误），并且始终按十进制解释（即使它以 0 开始也是如此）。

若 行号 为 0 或大于 32767 (C++11 前)2147483647 (C++11 起)，则行为未定义。

注解

一些自动代码生成工具从以其他语言书写的文件产生 C 源文件时，会使用此指令。这种情况下，它们在所生成的 C 文件中插入 #line 指令，以指代原（人类编辑的）源文件的行号和文件名。

请看下面实例：

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

int main()
{
    printf("before line,__LINE__=%d,file name %s\n",__LINE__,__FILE__);
#line 50 "new.c"
    printf("__LINE__=%d,file name %s\n",__LINE__,__FILE__);
    system("pause");
}
```

运行结果如下：

通过上图可以得出，未使用 line 指令时，行号是实际在文件中的行号，代码路径 VS 给出的是一个绝对路径，当我们用了 line 指令之后，得到的输出结果是我们设置后的。

10.6.2 指令#undef

#undef 指令用于“取消”已定义的#define 指令。也就是说，假设有如下定义：

#define LIMIT 400

然后，下面的指令：

#undef LIMIT

将移除上面的定义。现在就可以把 LIMIT 重新定义为一个新值。即时原来没有定义 LIMIT，取消 LIMIT 的定义仍然有效。如果想使用一个名称，又不确定之前是否已经用过，为安全起见，可以用#undef 指令取消该名字的定义。

10.6.3 变参宏：...和__VA_ARGS__

一些函数（如 printf()）接受数量可变的参数。stdvar.h 头文件提供了工具，让用户自定义带可变参数的函数。C99/C11 也对宏提供了这样的工具。虽然标准中未使用“可变”这个词，但是它已成为描述这种工具的通用词。

通过把宏参数列表最后的参数写成省略号（即，3 个点...）来实现这一功能。这样，预定义宏__VA_ARGS__可用在替换部分中，表明省略号代表什么。例如，下面的定义：

#define PRINT(...) printf(__VA_ARGS__)

请看下面代码实例：

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

#define PRINT(...) printf(__VA_ARGS__)

int main()
{
    int ht=171;
    float wt=60.1;
    PRINT("How do you do");
    PRINT("height =%d,weight=%.1f\n",ht,wt);
    system("pause");
}
```

预处理展开后代码如下：

```
int main()
{
    int ht=171;
    float wt=60.1;
    printf("How do you do");
    printf("height =%d,weight=%.1f\n",ht,wt);
    system("pause");
}
```

10.7 内联函数（C99）

通常，函数调用都有一定的开销，因为函数的调用过程包含建立调用、传递参数、跳转到函数代码并返回。使用宏使代码内联，可以避免这样的开销。C99 还提供另一种方法：内联函数（inline function）。读者从字面上看，会认为内联函数会用内敛代码替换函数调用。其实 C99 和 C11 标准中叙述的是：“把函数变成内联函数建议尽可能快地调用该函数，其具体效果由实现定义”。因此，把函数变成内联函数，编译器可能会用内联代码替换函数调

用，并（或）执行一些其他的优化，但是也可能不起作用。

创建内联函数的定义有多种方法。标准规定具有内部链接的函数可以成为内联函数，还规定了内联函数的定义与调用该函数的代码必须在同一个文件中。因此，最简单的方法是使用函数说明符 inline 和存储类别说明符 static。通常，内联函数应定义在首次使用它的文件中，所以内联函数也相当于函数原型。

如下所示：

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

inline static void eatLine()//内联函数定义
{
    while(getchar()!='\n')
    {
        continue;
    }
}

int main()
{
    …
    eatLine();//函数调用
    …
}
```

编译器查看内联函数的定义，可能会用函数体中的代码替换 eatLine()函数调用。也就是说，效果相当于再函数调用的位置输入函数体中的代码：

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

inline static void eatLine()//内联函数定义
{
    while(getchar()!='\n')
    {
        continue;
    }
}

int main()
{
    …
    while(getchar()!='\n') //替换函数调用
    {
        continue;
    }
    …
```

}

由于并未给内联函数预留单独的代码块，所以无法获得内联函数的地址（实际上可以获得地址，不过这样做之后，编译器会生成一个非内联函数）。另外，内联函数无法在调试器中显示。

内联函数应该比较短小，而且是非常高频的调用，把较长的函数变成内联并未节约多少时间，因为执行函数体的时间比调用函数的时间长的多。

编译器优化内联函数必须知道该函数定义的内容。这意味着内联函数定义与函数调用必须在同一个文件中。鉴于此，一般情况下内联函数都具有内部链接。因此，如果程序有多个文件都要使用某个内联函数，那么这些文件中都必须包含该内联函数的定义。最简单的做法是，把内联函数定义放入头文件，并在使用该内联函数的文件中包含该头文件即可。

10.8 C 库中的可变参数 stdarg.h

在 10.6.3 提过可变参数宏，即该宏可以接受可变数量的参数。stdarg.h 头文件为函数提供了一个类似的功能，但是用法比较复杂。必须按如下步骤进行：

1. 提供一个使用省略号的函数原型；
2. 在函数定义中创建一个 va_list 类型的变量；
3. 用宏把该变量初始化为一个参数列表；
4. 用宏访问参数列表；
5. 用宏完成清理操作。

接下来

练习题：

答案解析：