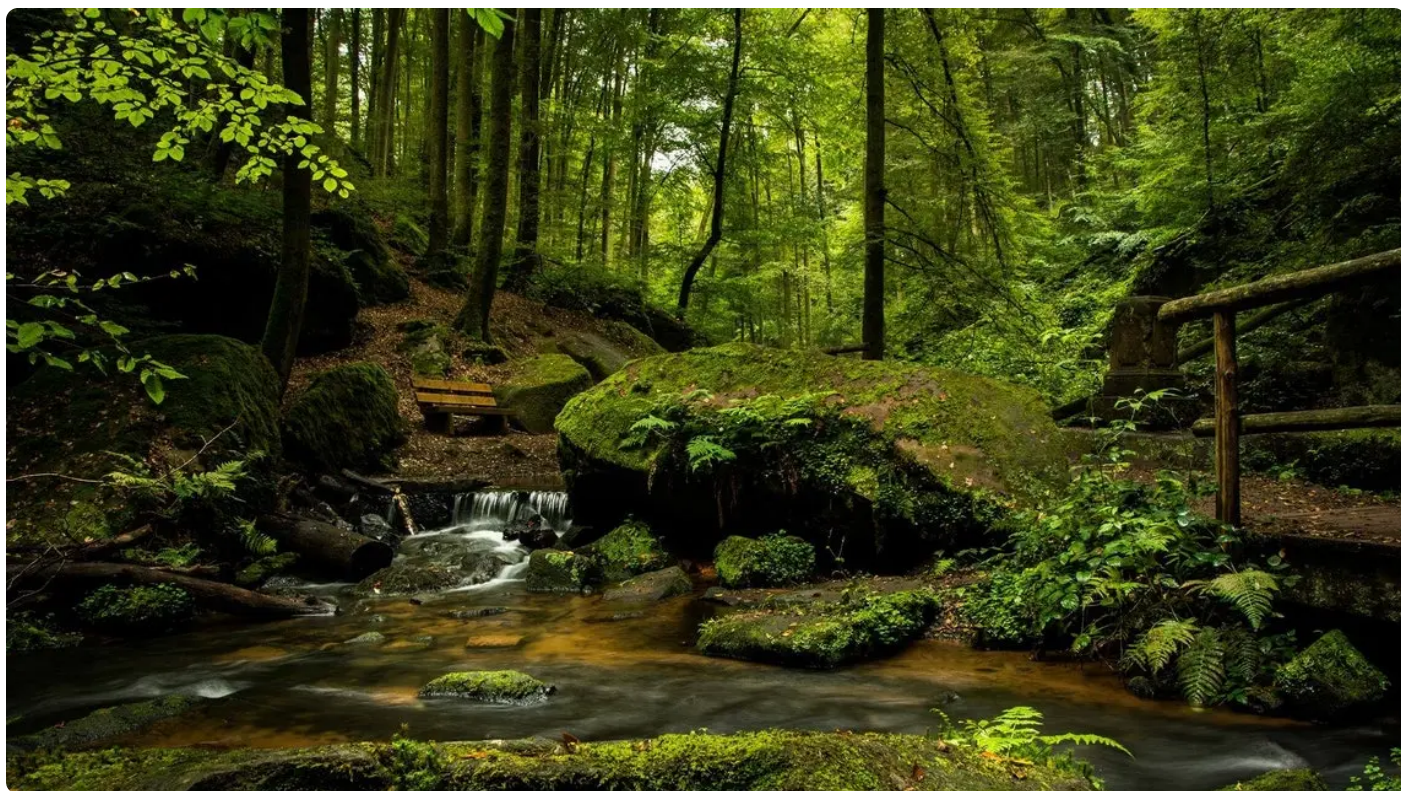


## 09 | 编译准备：预处理器是怎样处理程序代码的？

2021-12-27 于航

《深入C语言和程序运行原理》

[课程介绍 >](#)



讲述：于航

时长 13:26 大小 12.32M



你好，我是于航。

C 预处理器是 C 标准中的另一块重要内容。对代码进行预处理，是 C 源代码在被“真正”编译，并生成机器代码前的重要一环。合理使用预处理指令，可以让源代码根据不同的环境信息进行动态变化，并生成适合在当前环境下编译的 C 代码。这里我们提到的“环境”，一般与目标操作系统、CPU 体系架构，以及当前平台上各系统库的支持情况有关。

除此之外，预处理器还为我们提供了一定的能力，可以更加高效、灵活地组织 C 源代码。比如，我们可以对一个完整的 C 程序进行结构化拆分，根据代码在语法结构或功能定位上的不同，将它们分别整理在独立的 C 源文件中。而在包含有程序入口 `main` 函数的源文件内，我们便可以通过 `#include` 预处理指令，在编译器开始真正处理 C 代码前，将程序运行所需要的其他代码依赖包含进来。

领资料

那么今天，我们就来看看有关 C 预处理器的内容。接下来，我将介绍 C 预处理器的相关背景知识、预处理的基本流程，以及宏编写技巧和使用注意事项。

## C 预处理器的相关背景知识

预处理器被引入 C 标准的时间比 C 语言诞生晚了大约一年。1973 年左右，在贝尔实验室研究员 Alan Snyder 的敦促下，预处理器被正式整合至 C 语言中。它的最初版本只拥有基本的文件包含和字符串替换能力。而在此后不久，它被进一步扩展，加入了带参数的宏以及条件编译等功能。在随后发布的 ANSI C 标准中，预处理器的能力再次得到了加强。

另外你需要知道的是，C 预处理器并不仅仅可以用在 C 语言上，在 C++、Objective-C 等基于 C 的“后继语言”中，这套预处理器语法仍然适用。

除此之外，也许你也还并不清楚这一点：为什么使用预处理器语法 `#define` 定义出来的符号被称为宏（macro）？而那是因为在希腊语中，macro 通常会被作为一个单词前缀，用来表示体积或数量上的“大”和“多”。而在 C 代码中，当一个宏被展开和替换时，不是也有类似的效果吗？

到这里，有关 C 预处理器的一些背景知识我就介绍完了。下面，就让我们来看看编译器是如何对 C 代码进行预处理的。

## 预处理是怎样进行的？

对代码进行预处理是整个 C 程序编译流程中的第一环。在这一步中，编译器会对源代码进行分析，并通过查找以“`#`”字符开头的代码行，来确定预处理器指令的所在位置。接下来，通过下面这些步骤，编译器可以完成对代码的预处理工作：

1. 删除源代码中的所有注释；
2. 处理所有宏定义（`#define`），并进行展开和替换；
3. 处理所有条件预编译指令（如 `#if`、`#elif`），仅保留符合条件的代码；
4. 处理文件包含预编译指令（`#include`），将被包含文件的内容插入到该指令的所在位置；
5. 处理其他可以识别的预处理指令（如 `#pragma`）；
6. 添加其他具有辅助性功能的注释信息。



为了进一步观察编译器在预处理阶段对 C 代码的处理过程，这里我们可以进行一个简单的实验：将下面这段代码保存在文件“macro.c”中，并通过命令“gcc -O0 -Wall -E ./macro.c -o macro.i”对它进行编译。

 复制代码

```
1 #pragma GCC warning "Just FYI!"
2 #include <stdbool.h>
3 #define PI 3.14
4 #define SQUARE(x) (x * x)
5 int main(void) {
6     #if defined PI
7         // Some specific calculations.
8         const double area = SQUARE(5) * PI;
9         const bool isAreaGT100 = area > 100.0;
10    #endif
11    return 0;
12 }
```

眼尖的你会发现，我们在编译命令中使用到了名为“-E”的参数。该参数的作用是让编译器仅对源代码执行预处理阶段的一系列操作，之后就停止运行。当编译完成后，你可以在编译目录内找到名为“macro.i”的文件。而在这个文件中，便包含有源代码在经过预处理阶段后得到的中间代码结果。其中的内容如下所示（注意，不同的编译器和版本生成的文件内容可能有所不同）：

 复制代码

```
1 # 1 "macro.c"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 31 "<command-line>"
5 # 1 "/usr/include/stdc-predef.h" 1 3 4
6 # 32 "<command-line>" 2
7 # 1 "macro.c"
8
9 # 1 "/usr/lib/gcc/x86_64-redhat-linux/8/include/stdbool.h" 1 3 4
10 # 3 "macro.c" 2
11
12
13
14 int main(void) {
15
16     const double area = (5 * 5) * 3.14;
17     const
18 # 8 "macro.c" 3 4
19     _Bool
```

领资料



```
20 # 8 "macro.c"
21         isAreaGT100 = area > 100.0;
22     return 0;
23 }
```

可以看到，在这段文本中，所有之前在 C 源代码中以“#”开头的预处理指令都已经被移除；宏常量 **PI** 和宏函数 **SQUARE** 已经完成了展开和替换；头文件“**stdbool.h**”中的内容也已经被插入到了源文件中（该文件内为宏 **bool** 的定义，这里已被替换为了 **\_Bool**）。除此之外，一些带有辅助功能的信息也以 **linemarker** 的形式插入到了该文件中，供后续编译阶段使用。

因为这些信息的组成形式和作用跟编译器的具体实现密切相关，这里我就不深入解读了。如果你对 GCC 如何使用 **linemarker** 感兴趣，可以点击 [这个链接](#) 获得更多信息。

## 定义宏函数时的常用技巧

预处理器在进行宏展开和宏替换时，只会对源代码进行简单的文本替换。在某些情况下，这可能会导致**宏函数所表达的计算逻辑与替换后 C 代码的实际计算逻辑产生很大差异**。因此，在编写宏函数时，我们要特别注意函数展开后的逻辑是否正确，避免由 C 运算符优先级等因素导致的一系列问题。

接下来，就让我们一起看下：当在 C 代码中使用预处理器时，有哪些 **tips** 可以帮助我们避免这些问题。

### 技巧一：为宏函数的返回值添加括号

对于大多数刚刚接触 C 预处理器的同学来说，我们可能会在不经意间，写出类似下面这样的代码：

```
1 #include <stdio.h>
2 #define FOO(x) 1 + x * x
3 int main(void) {
4     printf("%d", 3 * FOO(2));
5     return 0;
6 }
```

 复制代码

领资料

这里，我们定义了一个名为 **FOO** 的宏函数，该函数接收一个参数 **x**，并返回这个参数在经过表达式 **1 + x \* x** 计算后的结果值。在 **main** 函数中，我们以数值 **2** 作为参数，调用了该宏



函数，并通过 `printf` 函数打印了数字 3 与宏函数调用结果值的乘积。

按照我们对函数的理解，这里的宏函数在被“调用”后，会首先返回表达式的计算结果值 5（ $1 + 2 * 2$ ）。随后，该结果值会再次参与到函数外部的乘积运算中，并得到最终的打印结果值 15。但事实真是如此吗？

实际上，宏函数的展开与 C 函数的调用过程并不相同。经过编译器的预处理后，上述代码中对第四行 `printf` 语句的调用过程会被变更为如下形式：

```
1 printf("%d", 3 * 1 + 2 * 2);
```

 复制代码

可以看到，这里的宏函数 `FOO` 在被展开时，并不会优先对其内部的表达式进行求值，相反，只是简单地对传入的参数进行了替换。因此，当编译器按照展开后的结果进行计算时，由于表达式中乘法运算符“`*`”的优先级较高，进而导致整个表达式的计算顺序发生了改变，所以，计算结果也就出现了偏差。那如果我们为宏函数的整个返回表达式都加上括号，结果会怎样呢？显而易见，此时表达式  $(1 + 2 * 2)$  会被优先求值。 $3 * \text{FOO}(2)$  的计算结果符合我们的预期。

## 技巧二：为宏函数的参数添加括号

用上面的技巧，我们已经对宏函数 `FOO` 进行了优化，以保证在某些情况下，“返回的”表达式能够被当作一个整体进行使用。但这样就万无一失了吗？其实，类似的问题还可能会出现在宏函数的参数上，比如下面这个例子：

```
1 #include <stdio.h>
2 #define FOO(x) (1 + x * x)
3 int main(void) {
4     printf("%d", FOO(1 + 2));
5     return 0;
6 }
```

 复制代码

领资料

在这里，我们改变了宏函数 `FOO` 的使用方式，直接将表达式  $1 + 2$  作为参数传递给了它。同样地，由于编译器在处理宏函数时，仅会进行实参在各自位置上的文本替换，传入函数的表

达式并不会在函数展开前进行求值。因此，经过编译器的预处理后，上述代码中第四行对 `printf` 语句的调用过程会被变更为如下形式：

 复制代码

```
1 printf("%d", (1 + 1 + 2 * 1 + 2));
```

由于乘法运算符 “\*” 的存在，此时整个表达式的求值顺序发生了改变。本该被优先求值的子表达式 `1 + 2` 并没有被提前计算。很明显，这并不是我们在设计 `FOO` 函数时所期望的。而通过为宏函数定义中的每个参数都加上括号，我们便可以解决这个问题。

### 技巧三：警惕宏函数导致的多次副作用

那么，现在的宏函数 `FOO` 还会有问题吗？让我们继续来看这个例子：

 复制代码

```
1 #include <stdio.h>
2 #define FOO(x) (1 + (x) * (x))
3 int main(void) {
4     int i = 1;
5     printf("%d", FOO(++i));
6     return 0;
7 }
```

在上面这段代码中，我们“装配”了经过两次优化升级的宏函数 `FOO`。只是相较于前两种使用方式，这里我们在调用该函数时，传入了基于自增运算符的表达式 `++i`。按照常规的函数调用方式，变量 `i` 的值会首先进行自增，由 `1` 变为 `2`。然后，该结果值会作为传入参数，参与到宏函数中表达式 `1 + (x) * (x)` 的计算过程，由此可以得到计算结果 `5`。

但现实往往出人意料：同之前的例子类似，传入宏函数的表达式实际上并不会在函数被“调用”时提前求值。因此，上述 `C` 代码中的第五行语句在宏函数实际展开时，会变成如下形式：

 领资料

 复制代码

```
1 printf("%d", (1 + (++i) * (++i)));
```

到这里，我想你已经知道了问题所在：经过宏替换的 C 代码导致多个自增运算符被同时应用在了表达式中，而该运算符对变量 `i` 的副作用产生了多次。

因此，在使用宏函数时需要注意，宏函数的“调用”与 C 函数的调用是完全不同的两种方式。前者不会产生任何栈帧，而只是对源代码文本进行简单的字符替换。所以，对于替换后产生的新代码，其计算逻辑可能已经发生了变化，而这可能会引起程序计算结果错误，或副作用产生多次等问题。

## 技巧四：定义完备的多语句宏函数

到这里，对于定义简单宏函数时可能遇到的一系列问题，相信你都能处理了。但当宏函数逐渐变得复杂，函数体内不再只有一条语句时，新的问题又出现了。

通常情况下，为了与 C 代码的风格保持一致，在调用宏函数时，我们也会习惯性地为每一个调用语句的末尾加上分号。但也正是因为这样，当含有多行语句的宏函数与某些控制语句一起配合使用时，可能会出现意想不到的结果。比如下面这个例子：

 复制代码

```
1 #include <stdio.h>
2 #define SAY() printf("Hello, "); printf("world!")
3 int main(void) {
4     int input;
5     scanf("%d", &input);
6     if (input > 0)
7         SAY();
8     return 0;
9 }
```

当我们习惯使用 C 代码的执行思维来看待宏函数时，上述代码的执行情况应该是这样的：程序接收用户输入的字符，并将其转换为数字值。若该值大于 0，则宏函数内的两条 `printf` 语句被执行，并输出字符串 “Hello, world!”。否则，程序直接退出。但现实的情况却是，无论用户输入何值，字符串 “world!” 都会被打印。

 领资料

而问题就出现在**宏函数 SAY 被展开和替换后**，原本“封装”在一起的两条 `printf` 语句被拆分开来。其中的第一条语句成为了 `if` 条件控制语句的执行内容；而第二条语句由于没有大括号的包裹，则直接被“释放”到了 `main` 函数中，成为了该函数返回前最后一条会被执行的语句。

那么，应该怎样解决这个问题呢？

既然我们的目的是让编译器在处理程序代码时，能够将宏函数内的所有语句当作一个整体进行处理，那么有没有一种合法的 C 语言结构，它可以作为一种复合语句使用，其内部的语句只会被执行一次，并且在语法上，它还需要以分号结尾？

很巧，迭代语句 `do...while` 便可以满足这个要求。如下面这段代码所示，我们使用该语句，改写了宏函数 `SAY` 的实现方式。

 复制代码

```
1 #include <stdio.h>
2 #define SAY() \
3     do { printf("Hello, "); printf("world!"); } while(0)
4 int main(void) {
5     int input;
6     scanf("%d", &input);
7     if (input > 0)
8         SAY();
9     return 0;
10 }
```

可以看到，通过将 `while` 关键字中的参数设置为 `0`，我们可以保证整个迭代语句仅会被执行一次。而 `do...while` 语句“天生”需要以分号结尾的性质，也正好满足了宏函数替换后的 C 语法规式要求。并且，对于 `while(0)` 这种特殊的迭代形式，大多数编译器也会通过相应的优化，去掉不必要的循环控制结构，以降低对程序运行时性能的影响。

## 何时使用预处理器？

讲了这么多预处理器的使用技巧，最后还是要提醒你：预处理器是一把“双刃剑”。对它的合理使用，可以让我们的程序具备更强的动态扩展能力；相反，如果任意乱用，就会导致程序源代码的可读性大大降低，甚至引入难以调试的 **BUG**。

通常，在以下三个场景中，你可以视情况选择是否使用预处理器：

- 定义程序中使用到的魔数。这里提到的魔数，主要是指那些用于控制程序运行状态、具有特定功能意义的参数。这些参数可以使用预处理器以宏的形式定义，并在程序编译前内联到源代码中使用；





- 基于特定运行环境的条件编译。我们可以通过特定的宏（比如编译器预定义宏、C 语言内置宏等）来检测当前编译环境的状态，并以此调整程序需要启用的特定功能；
- 封装代码模板。我们可以通过宏的形式，封装需要重复出现的代码片段，并将它们应用在循环展开等场景中。

当然，真实的使用场景并不局限于这三类。但还是要强调下：在使用预处理器时，保持谨慎小心是必要的。

## 总结

好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

今天我主要介绍了与 C 预处理器有关的一些内容，包括相关背景知识、编译器处理方式，以及使用技巧等。

通过 C 预处理器，我们可以让编译器在真正开始处理 C 语法代码前，先对源码进行一系列必要的转换。这些转换可以让我们引入代码正常编译所需要的各类依赖项，动态修改代码以适应不同的编译环境，甚至根据需要自动生成部分 C 代码。对预处理器的合理使用，可以让我们的程序具备一定的“弹性伸缩”能力，并使程序的可配置性大大增强。

另一方面，宏函数的运作方式又与 C 函数有着巨大的区别。其中，前者在“调用”时仅会做源代码文本上的匹配与替换。因此，在处理宏函数参数以及宏函数体内的多行语句时，需要通过添加括号、使用 `do...while` 语句等技巧，来对参数、返回值与函数体进行“封装”。

最后，需要注意预处理器的合适使用场景。在 C 语言中，预处理器通常可以被应用在程序魔数定义、代码条件编译，以及代码模版封装等场景中。只有谨慎、合理地使用，我们才能够享受到 C 预处理器带来的巨大价值。

## 思考题


你知道 C 预处理运算符 `#` 与 `##` 的用法吗？欢迎在评论区跟我讨论。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。



分享给需要的人，Ta 订阅超级会员，你最高得 50 元

Ta 单独购买本课程，你将得 20 元

 生成海报并分享

 赞 4  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 08 | 操控资源：指针是如何灵活使用内存的？

下一篇 10 | 标准库：字符、字符串处理与数学计算

## 更多课程推荐

# 操作系统实战 45 讲

## 从 0 到 1, 实现自己的操作系统

彭东  
网名 LMOS  
Intel 傲腾项目关键开发者



新版升级：点击「 请朋友读」，20 位好友免费读，邀请订阅更有现金奖励。

领资料

## 精选留言 (6)

 写留言



LDxy  
2021-12-27

看起来预处理器似乎是一个独立于编程语言的东西，那为何大多数语言不引入预处理器呢？这

样想用宏的人可以用宏，不想用宏的人可以不用，不是更能满足程序员吗

作者回复: 就像你说的一样，其实大部分“现代语言”都不希望将预处理器作为语言自身标准的一部分，这意味着你可以用一些第三方提供的预处理能力来对源码进行编译前的处理。总的来看，大部分人认为预处理器的过度使用通常会导致项目整体结构的混乱，且不利于代码调试。并且宏作为简单的文本替换也不具备语义上的完备性（宏定义时的形式无法决定其具体使用时产生的效果）。因此，从语言的角度来看，大家都更倾向于做编译时优化，尽量让语言本身的特性就可以满足预处理器的使用场景。个人理解哈~



2



小爷

2022-03-07

##的作用是连接的作用；而#的作用是就是把参数当作字符串代替。



1



贺志鹏

2022-02-24

#: 将宏的参数转换为字符串，这个操作称为"stringization"。

##: 将两个Token粘合成一个Token。



=

2022-01-08

#可以用来进行宏定义与替换、文件包含、条件编译、错误生成、行控制、预定义名称  
##可以用来进行token的联结



linker

2021-12-27

## 用来连接字符串

# 操作符#在字符串中输出实参



领资料



白花风信子

2021-12-27

# # 有次在重构自己的小作业用过，有点类似在预处理中预处理的感觉qwq。



