**概述**

C语言的编译预处理，可以做很多事情，最常见的#include头文件以及为了防止重复包含报错的#ifndef…#define这种，所有以#开头的行都是C代码中的预处理行，稍后会进行相关内容的整理，在此不做详述。

如果是狭义的编译器层面，不应当为预处理做任何事情。因为对于编译器而言，此时所有的预处理命令都被预处理器所完成了。

当然如果是我们通常意义上的类似于GCC这种编译器，那么其实包含了预处理器，编译器，汇编器和链接器等多个部件，肯定会包含预处理器。并且在C11标准的6.10部分，也是介绍了预处理的内容。

更重要的是，假如想要做一个能实际工作、验证或者自举而不是理论上正确的编译器，包含<stdio.h>等库文件进行一些平台依赖的处理，去做一些文字输出，是最起码的需求，因此，无论如何，预处理器的实现都是必须的。

在C11标准的5.1部分有描述C编译器工作的多个阶段，在这几个阶段中，其中前半部分大量的内容都是描述预处理器的工作方式。而实际上，真正的编译工作是从第7个阶段才开始的，之前的内容都是描述预处理器的工作。

在具体的实现中，预处理器可能会将这些内容合并起来——讲真多遍扫描罢了。

**预编译和库文件**

一个具体的问题是，预编译和库函数的问题，对于这些常用的库函数，大家为了能够编译速度快一点，都是用预编译技术去处理好那些公共的库文件的，什么stdio就是这样，然鹅这也造成一个问题就是，预编译器需要处理这些已经被编译好的头文件。GCC设计了一种gch文件，用于表示这种东西，而由于每个编译器的中间表示格式的不同，每家的文件格式还都不一样。

这种预编译技术的实质就是在文件中存储好在编译期需要用到的比如符号表等等东西，因此，在实际的编译预处理中，GCC在编译include的文件之前就会看看是否有预编译好的东西，有就直接拿来用而不是再处理一遍。这意味着我们需要在预处理器中做好同样的事情——当然，也可以不做，而是把那些头文件的源代码也编译进去罢了。

另一个让我非常头疼的问题是，当我试图使用linux的预处理器进行处理源文件，然后使用输出的i文件进行编译的时候，结果发现，在gcc给出的头文件中，存在非常多的例如attribute关键字之类的gcc自定义的东西，这使得我直接引用gcc的头文件变得几乎没有可能性（我想任何一个试图写一个只有标准C的编译器的人都无法绕开这些问题，他必须依赖现有的编译器去工作和调试，但是如果现有的编译器和C标准有一定的出入，那么这种矛盾是不可避免的）。

当然，不引用gcc的头文件也是可以的，但是，如果我想要让我的代码能够自举，那么一些公共的库函数是必不可少的，否则连基本的io都做不到，怎么读取源文件和输出s文件呢。

好在，花费了很多时间查资料和整理思路之后，我在此进行一定的说明。

首先需要理清楚的一件事情是，库函数的声明和实现是完全两回事。如果稍微有一点相关知识的人都应该明白，在头文件中进行了声明，只是为了过编译。而在链接阶段，才和具体的实现相连接起来。

在gcc当中，也是一样的，正常写代码所忽略的细节是，在gcc后端的链接器中，可以直接指定C标准库的so动态链接库，因此，翻译成汇编只需要一句call即可，在链接的时候，自然会和C库中的代码连起来。至于C runtime相关的东西，我就不说了，这不是重点的东西。

由于我们的目标只是一个编译器，能自举，那么，我们可以直接使用gcc的gas和linker，只是在前面的编译阶段使用我们自己的编译器罢了。

因此，只需要我们自己实现一遍C头文件中所需要的定义，随后在makefile的link过程语句中，使用-l指定对应的动态库进行链接即可。

最后，在思考这个事情怎么解决的过程中，我仍然认为，加入内联汇编是必要的——这源于我最开始试图手工调用系统调用而完全绕开gcc这种愚蠢的想法所造成的。

即便如此，我仍然选择保留了我想要选用的内联汇编，因为这是一个面向硬件的C编译器所必须考虑的事情。

接下来，我将根据不同的预处理语句，以及在C标准中陈述的6个预处理器的阶段，对我所做的C预处理的工作做一个描述。

* **阶段1**

先抄定义：

|  |
| --- |
| Physical source file multibyte characters are mapped, in an implementation-defined manner, to the source character set (introducing new-line characters forend-of-line indicators) if necessary. Trigraph sequences are replaced bycorresponding single-character internal representations. |

这个阶段1的定义就两句话，我也是真的没看懂。我的理解就是从磁盘读进来，什么转换为源文件字符集，这肯定啊，读进来就是这样了，还能怎么转换。

主要是第一句话，我觉得应该说的就是读取文件转成sourcecharacterset。第二句话，意思很简单，把所有的三字母词替换掉，我不做一一列举，因为标准中都有，当然三字母词肯定优先于注释，不然也不会有下面这个经典的例子（不懂得自己想办法吧）：

|  |
| --- |
| // 注释语句 ??/  a = b+c; |

（不用考虑嵌套，因为显然不可能，所以扫描一遍就好了）

* **阶段2**

继续引用原文：

|  |
| --- |
| Each instance of a backslash character (\) immediately followed by a new-linecharacter is deleted, splicing physical source lines to form logical source lines.  Only the last backslash on any physical source line shall be eligible for being partof such a splice. A source file that is not empty shall end in a new-line character,which shall not be immediately preceded by a backslash character before any suchsplicing takes place. |

好吧，反正这句话很简单，‘\’之后换行表示，不中断当前行然后把后面的行接续上去。这一步把这个事情给做了。阶段一的代码就是因为这步在三字母词之后，才会当做注释给忽略掉的。

* **阶段3**

来，黏贴原文：

|  |
| --- |
| The source file is decomposed into preprocessing tokens 7) and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment. Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined. |

这一步做了两个事情：

第一个事情是解析出预处理字符以及相关的空白符。

第二个事情，把注释换掉（所以阶段1里面的例子才会发生，他先替换三字母词，再接续行，再弄掉注释），换成一个空格键。但是他也说了，多余的空白字符，是保留还是全部替换成1个空格键，是由实现决定的。所以呢，这个我们就不把多余的空白字符给除去了哈。

* **阶段4**

继续：

|  |
| --- |
| Preprocessing directives are executed, macro invocations are expanded, and \_Pragma unary operator expressions are executed. **If a character sequence that matches the syntax of a universal character name is produced by token concatenation (6.10.3.3), the behavior is undefined.** A #include preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively. All preprocessing directives are then deleted. |

有这么几件事儿：

1、预处理指令执行

2、宏定义扩展

3、\_pragma指令被执行

4、#include包含的文件按照从阶段1到阶段4，递归执行。

5、然后删除所有预处理指令。

但是对于加粗的那行话，我是反正没看懂（这跟英文好不好没关系啊，这词是啥意思得去全文找），反正行为未定义就是了。至于那个（6.10.3.3）我查看了一下，就是那个“##”符号在宏定义里面的用处，大概就是宏替换相关的。

这一步的内容就比较多了，接下来，对各种预定义指令进行分析。

* **阶段5**

|  |
| --- |
| Each source character set member and escape sequence in character constants andstring literals is converted to the corresponding member of the execution characterset; if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character. 8) |

Sourcecharacterset和转义字符转换为执行字符集中的内容。

本来第二句我也没看懂，但是看了上下文，大概意思是说，这里有两个字符集，一个是源文件字符集sourcecharacterset，另一个是执行字符集，那需要将源文件字符集转换为执行字符集。

注意到阶段1当中的描述，用一种implement-defined的方式去进行这种转换。然而如果有特定编译器自己定义的字符，不在源文件字符集中的，要进行这种转换，需要定义好，并且转换之后必须不是null字符。

* **阶段6**

这里就只有一句话

|  |
| --- |
| Adjacent string literal tokens are concatenated |

这句话的意思也是我做了一些尝试和错误之后才弄明白的。

原来它指的就是，比如说我在一个char[2][10]中定义两个字符串“ab”和“cd”吧，但是，一不小心，中间分隔两个字符串的逗号给弄丢了，于是乎，编译器在这个阶段会非常神奇的把这两个字符串当成一个，头尾连接之后，给处理了，就变成了“abcd”。

在这之后的阶段7其实就应该是编译器的阶段了。

**预处理器**

* **Tips**
  + 源字符集和运行字符集，以及转义字符的处理

之所以会在这里特意提及，有很多原因。

第一个方面，正是由于这两个字符集在阶段5当中的转换，以及在编译阶段使用的字符集会影响词法分析器的分析的缘故。

换而言之，以制表符’\t’为例，如果源文件中出现这么一个符号，在阶段5和编译的词法分析，这两个阶段中，到底使用的是制表符’\t‘实际表示的符号，还是反斜杠加上符号t，也就是‘\’+’t’？

无论是哪种编译器的词法分析器，都必须分辨清楚，这两种不同的字符规则的处狸，否则必然会造成词法分析器规则的混乱。

我仔细看了标准文件对于这些的定义之后，认为，阶段5所说的，主要是针对编译器自定义的扩展字符集，而对这些转义字符，则不做转换。否则，在词法分析阶段，这些字符会被当成whitespace所忽略掉，这是不合理的。

第二个方面，则是由于各种标准码对于源字符集的扩展，这是我在处理unicode字符的时候所遇到的问题，标准的C11只定义了基本的字符集，但是很显然，对于源字符集，是基本字符集的一个超集。在词法分析器中，对于string词串，以u8,U,L等开头，后面的引号之中，原文的表述如下：

|  |
| --- |
| any member of the source character set except  the double-quote ", backslash \, or new-line character |

这里面的源字符集肯定是包括了扩展字符集合的，而不仅仅只是基本字符集，从而造成了一个在词法分析上面的问题，对于如下的一段代码：

|  |
| --- |
| char s1[]=u8"你好"; |

对于扩展字符集合支持Unicode的编译器来说，这个‘你好’是可以被编译通过的，但是在mcc中，这将会引起一个词法分析器的错误。

当然，这并不意味着mcc不支持Unicode，只不过，不能在源代码中直接打你好，而是改用他在Unicode里面实际的8进制或者16进制编码罢了。

这是由于支持的源字符集的大小不同所导致的问题。

我想这个案例，能够更加清楚的说明在阶段1和阶段5中，那两句莫名其妙的关于multibyte字符，源字符集和运行字符集转换之间的说明，自己意会吧。

因此这点说明放在了预处理方面来介绍，即mcc编译器部分所支持的字符集也就是C11标准所指定的基本字符集罢了（显然是和预处理器一个水平的那个狭义的编译器啊）。

（我只是做个能用的编译器，没必要加进去那么大一个Unicode字符集啊）。

但是，这并不意味着，这些支持可以不做，假如，源字符集当中，存在一个“你好”这样的字符，那么，在预处理阶段，按照阶段1和阶段5这两部分的说明，就应该将这么一个汉字编码，表达为实际的Unicode编码并进行替换。

* + 回车和换行

对于回车符和换行符，我们现在都默认两者一样，敲下回车键就是换行，但是实际并不一样。这是我在网上搜到的说明：

|  |
| --- |
| \n: UNIX 系统行末结束符  \n\r: window 系统行末结束符  \r: MAC OS 系统行末结束符 |

Mcc目前的编译环境是win下的linux子环境，那就是linux，但是我的test.c的测试文件是在win下的编辑器中编写的，mcc目前也没有指定的编译环境要求。

而预处理器中，每条预处理指令都占用一行，和这两个字符打交道是不可避免的，因此，这种跨平台的考量是不得不加入进去的。

由此引出了getline的实现，标准C中是没有getline的，据说gcc弄了一个，但是跟我关系不大，毕竟为了自举，最后肯定需要加标准库文件进去的。

Getline的处理，我就不多说了。