# CPP命令

在C++编译环境中实际上是个exe。主要完成预处理工作。

## 用途

完成C语言源文件上的文件包含和宏置换。

## 语法

/usr/ccs/lib/cpp [-C] [-P] [-qDBCS ] [-IDirectory ] [-UName ] [-DName [=Defin ition ] ] [-qlanglvl=Language ] [ InFile ] [ OutFile ]

## 描述

cpp 命令完成 C 语言源文件上的文件包含和宏置换。它读 InFile 并且写到 OutFile （缺省为标准输入和标准输出）。

cpp 命令被设计用来符合由文档“Draft American National Standard for Information Systems Systems - Programming Language C”（ X3J11/88-159 ）定义的 C 语言预处理伪指令和指令。

cpp 程序识别下列的特殊名字：

\_\_LINE\_\_ 当前行号。

\_\_DATE\_\_ 源文件的转化日期。

\_\_TIME\_\_ 源文件的转化时间。

\_\_STDC\_\_ 指示一个一致的实现。

\_\_FILE\_\_ 当前文件名。

\_\_STR\_\_ 指出编译器将为某些字符串函数（在 /usr/include/string.h 中定义）生成直接插入的代码。

\_\_MATH\_\_ 指出编译器将为某些数学函数（在 /usr/include/math.h 中定义）生成直接插入的代码。

\_\_ANSI\_\_ 指出 langlvl 被设定等于 ANSI。

\_\_SAA\_\_ 指出 langlvl 被设定等于 SAA。

\_\_SAA\_L2\_\_ 指出 langlvl 被设定等于 SAAL2。

\_\_EXTENDED\_\_ 指出 langlvl 被设定等于 extended。

\_\_TIMESTAMP\_\_ 指出源文件最近修改的日期和时间。

所有 cpp 伪指令行必须以一个 #（磅符号）开始。这些伪指令是：

#define Name TokenString 用 TokenString 取代随后的 Name 实例。

#define Name ( Argument,...,Argument ) TokenString 把随后的 Name（Argument, . . . ,Argument）序列的实例用 TokenString 取代，这里 Argument 在 TokenString 中的每次出现都被逗号分隔的列表中相应的记号取代。注意，Name 和左括号之间不能有任何空格。

#undef Name 从这点开始忽略任何 Name 定义。

#include " File " or #include <File> 在这点包含 File 的内容，这个文件将被 cpp 处理。

如果您给 File 加上 " " （双引号）， cpp 命令首先在 InFile 目录中搜索，然后在以 -I 标志指定的目录中搜索，最后在一个标准列表上搜索。

如果您使用 <File> 符号表示法，cpp 命令只在标准目录中搜索 File。它不搜索 InFile 驻留的目录。

#line Number ["File"] 使得实现表现得好像接下去的源行序列以具有用 Number 定义的行号的源行开始。如果提供 File，则假定的文件名被改为 File。

#error TokenString 产生一个包含 TokenString 的诊断消息。

#pragma TokenString 编译器的已定义实现的指令。

#endif

结束以一个测试伪指令（#if 、 #ifdef 或 #ifndef）开始的行部分。每个测试伪指令都必须有一个相匹配的 #endif。

#ifdef Name 把随后的行放到输出中，仅当：

Name 已经由先前的 #define定义

或者

Name 已经由 -D 标志定义,

或者

Name 是一个 cpp 命令可识别的特殊名字，

并且

Name 还没有被一个插入的 #undef 取消定义，

或者

Name 还没有被 -U 标志取消定义。

#ifndef Name

把随后的行放到输出中，仅当：

Name 从没有被先前的 #define 定义，

并且

Name 不是一个 cpp 命令可识别的特殊名字，

或者

Name 已经被先前的一个 #define 定义，但是它已经被一个插入的 #undef 取消定义，

或者

Name 是一个 cpp 命令可识别的特殊名字，但是它已经被 -U 标志取消定义。

#if Expression

把随后的行放到输出中，仅当 Expression 求值不是零。所有的二进制未分配 C 运算符， ?: 运算符，和一元运算符 -、!、和 -,在 Expression 中都是合法的。运算符的优先顺序和 C 语言中定义的相同。还有一个一元运算符 defined，它可以在 Expression 中以两种形式使用：

defined （Name）或 defined Name 这允许 #ifdef 和 #ifndef 在一个 #if 伪指令中使用。只有这些被 cpp 所知的运算符、整型常量和名字可以在 Expression 中使用。 sizeof 运算符不可用。

#elif Expression 把随后的行放到输出中，只要前面的 #if 或 #elif 伪指令中的表达式求值为 false 或未定义，并且这个 Expression 求值为 true。

#else 把随后的行放到输出中，只要前面的 #if 或 #elif 伪指令中的表达式求值为 false 或未定义（因此在 #if 之后，在 #else 之前的行都被忽略）。

每个测试伪指令条件都被依次检查。如果它求值为 false （0），它控制的分组被跳过。只通过确定伪指令的名字来处理伪指令以便于跟踪嵌套条件的层次；组中有其它的预处理记号，伪指令的其余预处理记号被忽略。只有控制条件为 true（非零）的第一组被处理。如果没有一个控制条件计算为 true，并且有 #else 伪指令，则由 #else 控制的那组被处理；缺少一个 #else 伪指令，到#endif 为止所有的组都被跳过。

## 标志

-C 从源文件拷贝 C 语言注释到输出文件。如果您省略了这个标志，cpp 命令除去除了 cpp 伪指令行中的所有 C 语言注释。

-D Name [=Definition] 如同在一个 #define 伪指令中那样定义 Name。Definition 的缺省值是 1。

-I Directory

首先查找 Directory ，再查找针对#include 文件的标准列表上的目录中不是以一个 / （正斜杠）开始的名字。参见先前的关于 #include 的讨论。

-P 预处理输入而不为 C 编译器的下一趟执行产生行控制信息。

-q DBCS 指定双字节字符集方式。

-U Name 除去所有 Name 的初始定义，这里 Name 是一个预处理器预定义的一个符号（除了四个预处理器方式指示符： \_\_ANSI\_\_、\_\_EXTENDED\_\_、 \_\_SAA\_\_ 和 \_\_SAA\_L2\_\_）。在 ANSI 方式中不识别这个标志。

-qlanglvl= Language 为处理选择一个语言级别。Language 可以是 ANSI 、SAA 、SAAL2 或扩展。缺省是扩展。

注： 当 Language 是扩展时， \_NO\_PROTO 不被自动定义。可以使用 -D 选项完成这些定义，这个选项在 /etc/xlc.cfg 文件中。

## 示例

1. 为了显示预处理器发给 C 编译器的文本，输入：

/usr/ccs/lib/cpp pgm.c

这将预处理 pgm.c 并且在工作站上显示结果文本。当在宏定义中寻找错误时，您也许会想看看预处理器的输出。

2. 要创建一个包含更多可读的预处理过的文本的文件，输入：

/usr/ccs/lib/cpp -P -C pgm.c pgm.i

这将预处理 pgm.c 并且存储结果到 pgm.i 中。它忽略供 C 编译器使用的行编号信息（-P），并且包含程序注释（-C）。

3. 要预定义宏标识符，输入

4. /usr/ccs/lib/cpp -DBUFFERSIZE=512 -DDEBUG

5. pgm.c

pgm.i

这将在预处理前定义 BUFFERSIZE 的值为 512 以及 DEBUG 的值为 1。

6. 要使用位于非标准目录的 #include 文件，输入：

7. /usr/ccs/lib/cpp -I/home/jim/include

pgm.c

这将在当前目录中查找引号引起来的 #include 文件，然后在 /home/jim/include中，最后在标准目录中找。它在 /home/jim/include 中查找角括号括起来的 #include 文件（< >），然后在标准目录中找。

8. 要预处理 ANSI 定义，输入：

/usr/ccs/lib/cpp -qlanglvl=ansi pgm.c