**Options Controlling the Kind of Output**[**¶**](https://gcc.gnu.org/onlinedocs/gcc/Overall-Options.html#Options-Controlling-the-Kind-of-Output)

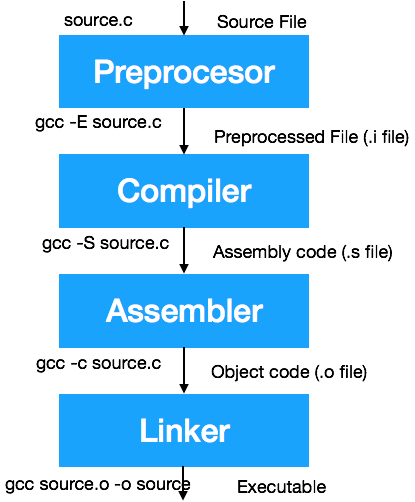
*file*.c

C source code that must be preprocessed.

*file*.d

D source code.

Khi sử dụng gcc với option -c, ta compile các file ‘.c’, ‘.i’, ‘.s’, etc., thành ‘.o’ nhưng không link



* -g: generates additional symbolic debuggging information for use with gdb debugger.
* -Wall: prints "all" Warning messages.
* -o: specifies the output executable filename.

**Compile and Link Separately** [**https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc\_make.html**](https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html)

The above command *compile* the source file into object file and link with other object files and system libraries into executable in one step. You may separate compile and link in two steps as follows:

// Compile-only with -c option

> **g++ -c -Wall -g Hello.cpp**

// Link object file(s) into an executable

> **g++ -g -o Hello.exe Hello.o**

The options are:

* -c: Compile into object file "Hello.o". By default, the object file has the same name as the source file with extension of ".o" (there is no need to specify -o option). No linking with other object files or libraries.
* Linking is performed when the input file are object files ".o" (instead of source file ".cpp" or ".c"). GCC uses a separate linker program (called ld.exe) to perform the linking.

**Compile and Link Multiple Source Files**

Suppose that your program has two source files: file1.cpp, file2.cpp. You could compile all of them in a single command:

> g**++ -o myprog.exe file1.cpp file2.cpp**

However, we usually compile each of the source files separately into object file, and link them together in the later stage. In this case, changes in one file does not require re-compilation of the other files.

> **g++ -c file1.cpp**

> **g++ -c file2.cpp**

> **g++ -o myprog.exe file1.o file2.o**

**Compile into a Shared Library**

To compile and link C/C++ program into a shared library (".dll" in Windows, ".so" in Unixes), use -shared option. Read "[Java Native Interface](https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html)" for example.

**Macro nối chuỗi:**

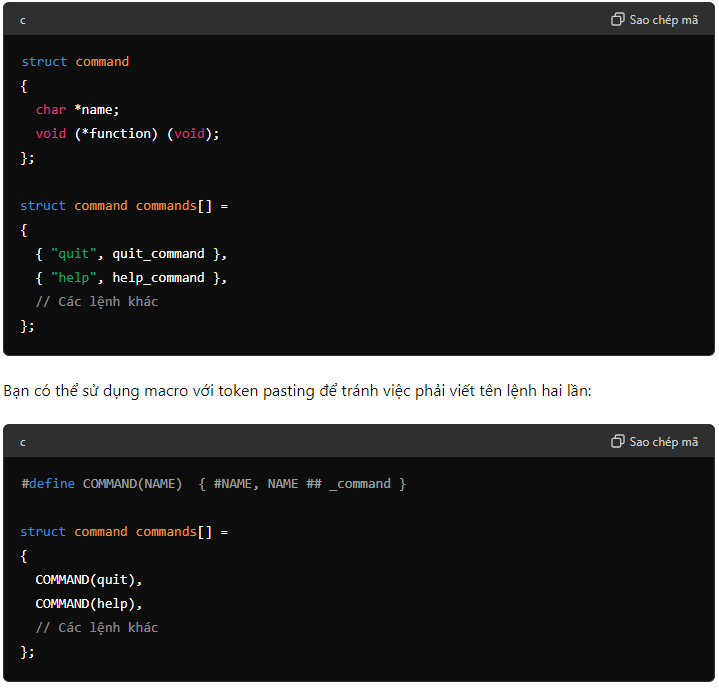
Token concatenation (gộp token) trong C là một kỹ thuật hữu ích trong quá trình mở rộng macro. Nó được thực hiện bằng cách sử dụng toán tử tiền xử lý **##**. Khi một macro được mở rộng, hai token nằm hai bên của toán tử **##** sẽ được kết hợp thành một token duy nhất, thay thế toán tử **##** và hai token gốc trong quá trình mở rộng macro.

**Ý nghĩa và cách hoạt động**

1. **Gộp Token**:
   * Toán tử **##** sẽ kết hợp hai token thành một token duy nhất.
   * Thông thường, cả hai token sẽ là các định danh (identifiers), hoặc một sẽ là định danh và một là số.
2. **Các Trường Hợp Hợp Lệ và Không Hợp Lệ**:
   * Các token hợp lệ khi kết hợp bao gồm định danh và số, hoặc các toán tử nhiều ký tự như **+=**.
   * Nếu hai token không thể tạo thành một token hợp lệ (ví dụ: **x** và **+**), preprocessor sẽ cảnh báo và phát ra hai token ban đầu.
3. **Ứng Dụng Khi Một Token Là Tham Số Macro**:
   * Token pasting hữu ích nhất khi một hoặc cả hai token là tham số của macro. Token này sẽ được thay thế bằng giá trị thực tế của tham số trước khi **##** thực hiện.
4. **Xử Lý Bình Luận**:
   * Bộ tiền xử lý chuyển đổi các bình luận thành khoảng trắng trước khi xét đến macro. Vì vậy, không thể tạo bình luận bằng cách kết hợp **/** và **\***.
5. **Ví dụ Cụ Thể**:
   * Khi viết một chương trình C có một bảng các lệnh, có thể sử dụng token pasting để tránh việc phải viết tên lệnh hai lần.

**Ví dụ**

Giả sử bạn có một chương trình với cấu trúc lệnh như sau:



**Giải Thích Ví Dụ**

1. **Định Nghĩa Macro COMMAND**

* **#NAME** chuyển tham số **NAME** thành một chuỗi (stringizing).
* **NAME ## \_command** kết hợp **NAME** với **\_command** thành một định danh duy nhất.

1. **Sử Dụng Macro COMMAND**:

* **COMMAND(quit)** mở rộng thành **{ "quit", quit\_command }**.
* **COMMAND(help)** mở rộng thành **{ "help", help\_command }**.

<https://gcc.gnu.org/onlinedocs/cpp/Concatenation.html>

**Dưới đây là một ví dụ cụ thể về quá trình biên dịch một chương trình C từ mã nguồn đến file thực thi:**

1. **Preprocessor**:
   * **File nguồn C** (hello.c):
   * #include <stdio.h>
   * #define GREETING "Hello, World!"
   * int main() {
   * // In greeting message
   * printf("%s\n", GREETING);
   * return 0;
   * }
   * **Sau Preprocessing** (hello.i):
   * int main() {
   * // In greeting message
   * printf("Hello, World!\n");
   * return 0;
   * }

**Thay đổi**: Các chỉ thị #include và #define đã được xử lý, và comment đã bị loại bỏ.

1. **Compiler**:
   * **Sau Compilation** (hello.s):
   * .section .text
   * .globl main
   * main:
   * pushq %rbp
   * movq %rsp, %rbp
   * movl $0, %eax
   * leaq .LC0(%rip), %rdi
   * call printf
   * movl $0, %eax
   * popq %rbp
   * ret
   * .section .rodata
   * .LC0:
   * .string "Hello, World!\n"

**Thay đổi**: Mã nguồn C đã được chuyển đổi thành mã hợp ngữ, mô tả các lệnh cụ thể cho CPU.

1. **Assembler**:
   * **Sau Assembly** (hello.o): Đây là một file nhị phân và không thể hiển thị dưới dạng văn bản, nhưng nó chứa mã máy tương ứng với mã hợp ngữ. **Thay đổi**: Mã hợp ngữ đã được chuyển đổi thành mã máy và được đóng gói trong một file đối tượng.
2. **Static Linking**:
   * **File thực thi** (hello.exe hoặc hello trên Linux): Đây là một file nhị phân thực thi và không thể hiển thị dưới dạng văn bản. Nó chứa mã máy cùng với tất cả các thư viện và phụ thuộc cần thiết để chạy độc lập. **Thay đổi**: Các file đối tượng và thư viện đã được liên kết để tạo ra một file thực thi hoàn chỉnh.

Quá trình này thể hiện sự chuyển đổi từ mã nguồn cao cấp sang mã máy thấp cấp, qua mỗi bước biên dịch, mã nguồn trở nên gần hơn với ngôn ngữ mà máy tính có thể hiểu và thực thi.

**Sự khác biệt của .o và .d:** [**https://stackoverflow.com/questions/15481056/how-to-use-o-and-d-files**](https://stackoverflow.com/questions/15481056/how-to-use-o-and-d-files)

The ".o" files are likely intermediate files from which the actual executable program should have been created.

The ".d" files are likely internal state used by the makefile, only important if you are making changes to the source code and then rebuilding "incrementally".

If, after running make, you have only these files but not the executable file, then the most likely explanation is that make encountered an error in creating the executable. If that's the case, then the last few lines of output generated by make should tell you more.