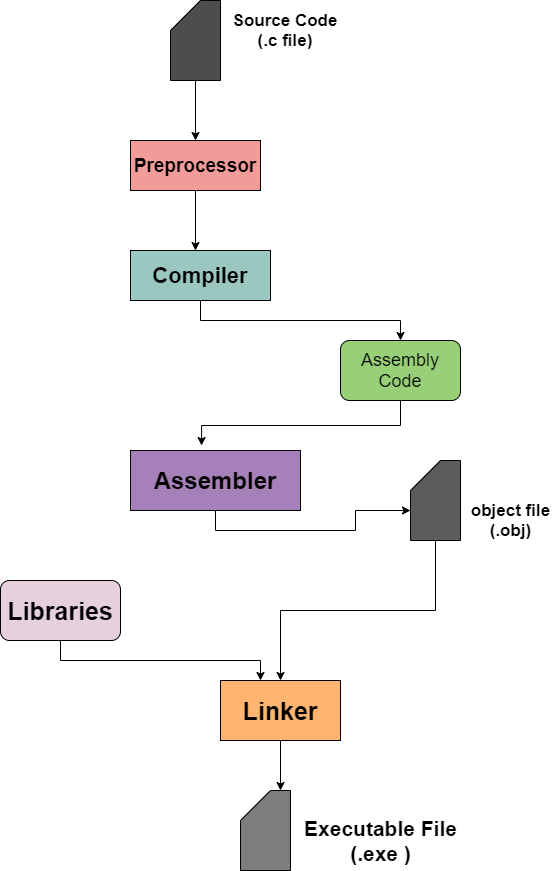
Sách The Book of C <https://jsommers.github.io/cbook/index.html#indices-and-tables>



Để tách Compile và Link ra thì ta dùng : gcc -c main.c (dùng để compile ra file main.o)

Điều này có thể giúp ta compile nhanh hơn nếu ta có nhiều file và ta chỉ chỉnh sửa 1 phần nhỏ (lúc này ta compile lại \*.o đó) rồi sử dụng : gcc -o main.exe file1.o file2.o … để link lại nhanh hơn

Mỗi lần include là sẽ chạy lại các definition, declaration của hàm và biến trong \*.h nên phải sử dụng #ifndef -> #define ->#endif để xử lý

**Thêm #define với gcc**

Để gọi pha tiền xử lý (preprocessing phase) khi biên dịch một chương trình bằng trình biên dịch gcc, bạn không cần làm gì đặc biệt. Tuy nhiên, bạn cũng có thể chỉ gọi pha tiền xử lý mà không biên dịch hoặc thực hiện bất kỳ công việc nào khác, và cũng có thể định nghĩa các ký hiệu mới của tiền xử lý trên dòng lệnh.

Để chỉ gọi pha tiền xử lý trong gcc, bạn có thể sử dụng lệnh:

```bash

gcc -E sourcefile.c

```

gcc cũng có một tùy chọn khác trên dòng lệnh để chỉ chạy pha tiền xử lý và kiểm tra cú pháp mã:

```bash

gcc -fsyntax-only sourcefile.c

```

Để định nghĩa các ký hiệu tiền xử lý mới (tương tự như #define), bạn có thể sử dụng tùy chọn -D của gcc, như sau:

```bash

gcc -DSYMBOL sourcefile.c

```

hoặc

```bash

gcc -DSYMBOL=VALUE sourcefile.c

```

Ví dụ, nếu bạn muốn biên dịch một chương trình và định nghĩa một ký hiệu tiền xử lý mới là DEBUG để bật chế độ debug, bạn có thể sử dụng lệnh sau:

```bash

gcc -DDEBUG -g -o output sourcefile.c

```

Trong ví dụ này, ký hiệu DEBUG sẽ được định nghĩa, và chương trình sẽ được biên dịch với tùy chọn -g để bật thông tin debug.

**Tạo mã assembly trong tệp sourcefile.s**

gcc -S sourcefile.c ->compile ra \*.o : gcc -c sourcefile.s -o file.o

**Tạo tệp đối tượng (binary machine code) trong sourcefile**. Đây là tùy chọn phổ biến hơn để sử dụng cho giai đoạn biên dịch. Khi tất cả các tệp nguồn đã được biên dịch thành mã đối tượng (.**o files**), tất cả các tệp .o có thể được liên kết để tạo ra một chương trình thực thi nhị phân.

gcc -c sourcefile.c

**-g và -Wall :** debug (-g) và báo lỗi mã nguồn (-Wall)

**-l link non-standard library** : <https://www.scaler.com/topics/header-files-in-c/>

**Non-Standard Header Files in C**

| **Non-Standard Header File** | **Uses** |
| --- | --- |
| <conio.h> | Console input/output functions (non-standard, primarily for MS-DOS) |
| <graphics.h> | Graphics functions (non-standard, primarily for MS-DOS) |
| <windows.h> | Windows API functions (non-standard, for Windows programming) |
| <unistd.h> | UNIX system calls (non-standard, for UNIX-like systems) |
| <process.h> | Process control functions (non-standard, for some systems) |

Các thư viện trên này nếu muốn compile thành công thì phải link vào, vd: -lprocess, -lpthread

Còn các thư viện standard thì không cần thêm -l:

At the moment, there are 29 standard headers for C:

* <assert.h>
* <ctype.h>
* <errno.h>
* <float.h>
* <limits.h>
* <locale.h>
* <math.h>
* <setjmp.h>
* <signal.h>
* <stdarg.h>
* <stddef.h>
* <stdio.h>
* <stdlib.h>
* <string.h>
* <time.h>

Added in C95:

* <iso646.h>
* <wchar.h>
* <wctype.h>

Added in C99:

* <complex.h>
* <fenv.h>
* <inttypes.h>
* <stdbool.h>
* <stdint.h>
* <tgmath.h>

Added in C11:

* <stdalign.h>
* <stdatomic.h>
* <stdnoreturn.h>
* <threads.h>
* <uchar.h>

**Sử dụng assert() để giới hạn các lỗi nhỏ có thể xảy ra**

**assert** trong C là một macro, không phải là một hàm

Kiểm tra không đổi và assert là một cách hiệu quả để phát hiện lỗi trong chương trình C. Trong C, việc truy cập ngoài phạm vi của mảng là một lỗi thời gian chạy rất phổ biến. Bạn có thể sử dụng hàm assert() để thêm các kiểm tra phạm vi của riêng bạn vào mã của bạn. Một vài giây để thêm các câu lệnh assert có thể tiết kiệm hàng giờ debugging.

Việc loại bỏ tất cả các lỗi là phần khó nhất và đáng sợ nhất khi viết một phần mềm lớn. Thêm các câu lệnh assert là một trong những cách dễ dàng và hiệu quả nhất để hỗ trợ trong giai đoạn khó khăn đó.

```c

#include <assert.h>

#define MAX\_INTS 100

void somefunction() {

// ...

int ints[MAX\_INTS];

i = foo(<một cái gì đó phức tạp>);

// i nên nằm trong phạm vi,

// nhưng thực sự là như thế nào?

assert(i>=0); // các câu lệnh kiểm tra an toàn

assert(i<MAX\_INTS);

ints[i] = 0;

// ...

```

Tùy thuộc vào các tùy chọn được chỉ định trong quá trình biên dịch, các biểu thức assert() sẽ được giữ lại trong mã để kiểm tra hoặc có thể được bỏ qua. Vì vậy, quan trọng là chỉ đặt các biểu thức không cần phải được đánh giá cho hoạt động chính xác của chương trình trong các kiểm tra assert().

```c

int errCode = foo(); // đúng --- ok

assert(errCode == 0);

if (assertfoo() == 0) ... // KHÔNG, foo() sẽ không được gọi nếu

// trình biên dịch loại bỏ assert()

```

Trong ví dụ này, các câu lệnh assert được sử dụng để kiểm tra xem giá trị của biến i có nằm trong phạm vi cho phép không. Nếu chương trình chạy vượt quá phạm vi, assert sẽ kích hoạt và chương trình sẽ dừng lại, giúp bạn phát hiện và sửa chữa lỗi một cách nhanh chóng.

Tóm lại assert() sử dụng cho các vị trí nhỏ dễ gây ra lỗi, khi điều kiện bao của ta đặt ra trong assert là đúng thì sẽ không báo lỗi gì (khác với lỗi cú pháp thì compiler sẽ báo lỗi cho ta)

**So sánh giữa Interpreter và Compiler**

|  |  |
| --- | --- |
| **Interpreter** | **Compiler** |
| Dịch từng câu lệnh trong chương trình ngay tại thời điểm thực hiện. | Quét toàn bộ chương trình trong một lần. |
| Sử dụng chủ yếu các ngôn ngữ Python, Perl, Matlab, Ruby... | Sử dụng ngôn ngữ C, C++ và C#. |
| Yêu cầu mã nguồn cho việc thực thi sau này. | Không yêu cầu mã nguồn thực thi sau này. |
| Mất ít thời gian phân tích mã nguồn, nhưng thời gian thực thi tổng thể khá chậm. | Mất nhiều thời gian để phân tích mã nguồn, nhưng thời gian thực thi tổng thể vô cùng nhanh chóng. |
| Dừng thông dịch nếu phát hiện ra lỗi. | Khởi tạo báo cáo lỗi sau khi chuyển đổi toàn bộ mã lệnh. |
| Lưu trữ ngôn ngữ máy dưới dạng mã máy trên đĩa. | Không tiết kiệm mã máy. |
| Dựa trên mô hình tải ngôn ngữ dịch thuật. | Dựa trên phương pháp diễn giải. |

**Debug với preprocessor**

[#define DEBUG(fmt,...) fprintf(stderr,fmt,\_\_VA\_ARGS\_\_) là một macro trong ngôn ngữ lập trình C1](https://stackoverflow.com/questions/1644868/define-macro-for-debug-printing-in-c). [Đây là một loại macro đặc biệt được gọi là macro đa tham số (variadic macros)](https://stackoverflow.com/questions/1644868/define-macro-for-debug-printing-in-c)[2](https://gcc.gnu.org/onlinedocs/cpp/Variadic-Macros.html).

* DEBUG(fmt,...) là tên của macro, với fmt và ... [là các tham sốfmt là chuỗi định dạng, giống như bạn sẽ sử dụng trong hàm printf1](https://stackoverflow.com/questions/1644868/define-macro-for-debug-printing-in-c).
* [fprintf(stderr,fmt,\_\_VA\_ARGS\_\_) là phần thân của macro, nó sẽ được thay thế cho DEBUG(fmt,...) khi macro này được sử dụng1](https://stackoverflow.com/questions/1644868/define-macro-for-debug-printing-in-c).
* [stderr là một con trỏ FILE, trỏ đến standard error stream, được sử dụng để xuất thông tin lỗi hoặc thông báo debug3](https://stackoverflow.com/questions/39002052/how-can-i-print-to-standard-error-in-c-with-printf).
* [\_\_VA\_ARGS\_\_ là một đại diện cho tất cả các tham số biến đổi được truyền vào macro](https://stackoverflow.com/questions/1644868/define-macro-for-debug-printing-in-c)[4](https://stackoverflow.com/questions/26053959/what-does-va-args-in-a-macro-mean). [Nó cho phép macro chấp nhận một số lượng tham số tùy ý2](https://gcc.gnu.org/onlinedocs/cpp/Variadic-Macros.html).

Về các macro tương tự, có thể có nhiều biến thể khác nhau tùy thuộc vào yêu cầu cụ thể của bạn. [Ví dụ, một số macro debug khác có thể bao gồm thông tin về tên file, số dòng và hàm từ đâu macro được gọi1](https://stackoverflow.com/questions/1644868/define-macro-for-debug-printing-in-c)[5](https://stackoverflow.com/questions/1941307/debug-print-macro-in-c):

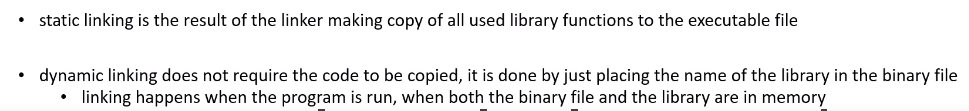
#define DEBUG\_PRINT(fmt, ...) do { if (DEBUG) fprintf(stderr, "%s:%d:%s(): " fmt, \_\_FILE\_\_, \_\_LINE\_\_, \_\_func\_\_, \_\_VA\_ARGS\_\_); } while (0)

[Trong đó, \_\_FILE\_\_, \_\_LINE\_\_, và \_\_func\_\_ là các macro tiền định nghĩa trong C, trả về tên file hiện tại, số dòng hiện tại và tên hàm hiện tại1](https://stackoverflow.com/questions/1644868/define-macro-for-debug-printing-in-c). [Macro này sẽ xuất ra thông báo debug kèm theo thông tin về vị trí trong mã nguồn mà nó được gọi1](https://stackoverflow.com/questions/1644868/define-macro-for-debug-printing-in-c).

<https://banvien.udemy.com/course/advanced-c-programming-course/learn/lecture/17962234#overview>

<https://banvien.udemy.com/course/advanced-c-programming-course/learn/lecture/17962238#overview> : debug with gdb CLI

**Static libraries in C \*.lib/\*.a vs shared object \*.so (ko gọi là dynamic libraries)**



**Static linking**

Link lúc compile, là library code 1 phần của app. Khi đã link và tạo app -> đổi code của library cần phải build lại TOÀN BỘ app -> tốn thời gian fix bug

**Dynamic linking**

Chỉ được link khi app được chạy

Được load vào ram bởi chương trình khi nó chạy

Library code chỉ được lưu trên máy (ko dính vào app) -> khi compile lại app thì chỉ compile các code được sửa/thêm vào library -> nhanh hơn

A white paper with black text

Description automatically generated