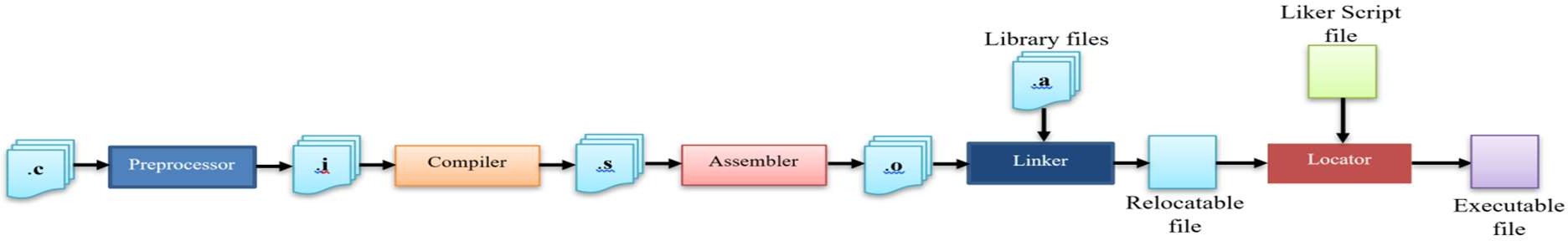
**C BUILD PROCESS**



**Hình 1.1:** C Build Process

**C Build Process** được coi là một trong những chủ đề quan trọng nhất trong lĩnh vực Phần mềm Nhúng và cũng là câu hỏi phổ biến nhất mà bạn sẽ được hỏi trong bất kỳ buổi phỏng vấn Phần mềm Nhúng nào.

Hôm nay, chúng ta sẽ thảo luận chi tiết về từng bước trong quy trình này để giúp bạn hiểu rõ hơn về quy trình này.

**C Build Process** là quá trình chuyển đổi mã nguồn cấp cao của phần mềm nhúng thành một tệp nhị phân thực thi. Quy trình này bao gồm nhiều bước và công cụ, nhưng ba bước chính và quan trọng nhất trong quy trình này là:

1. Mỗi tệp mã nguồn phải được biên dịch hoặc chuyển đổi thành một tệp đối tượng (object file).
2. Tất cả các tệp đối tượng tạo ra từ bước đầu tiên phải được liên kết với nhau để tạo thành một tệp đối tượng duy nhất, được gọi là chương trình có thể di dời (relocatable program).
3. Các địa chỉ bộ nhớ vật lý phải được gán cho các vị trí tương đối trong chương trình có thể di dời, trong một quy trình gọi là relocation (định vị lại).

Kết quả của bước cuối cùng là một tệp chứa hình ảnh nhị phân thực thi, sẵn sàng để chạy trên hệ thống nhúng.

Vậy, hãy bắt đầu thảo luận chi tiết về từng bước trong quy trình này để có thể hiểu rõ hơn.

### Pre-processor



#### Tổng quan

* Giai đoạn tiền xử lý là bước đầu tiên trong quy trình xây dựng C.
* Tệp đầu vào cho giai đoạn này là tệp **.c**.
* Tệp đầu ra là tệp **.i** (pre-processed file) chứa mã C thuần túy sau khi thực thi các chỉ thị tiền xử lý.

#### Công việc của Pre-processor

1. ***Loại bỏ chú thích (comments):***
   * Bộ tiền xử lý sẽ xóa tất cả các chú thích (//, /\* \*/) trong mã nguồn.
2. ***Thực thi các chỉ thị tiền xử lý (pre-processor directives):***
   * Các dòng bắt đầu bằng ký tự # (ví dụ: #include, #define, #ifdef,...) sẽ được xử lý.
   * Kết quả là mã nguồn C không chứa các chỉ thị tiền xử lý, chỉ còn mã lệnh thuần túy.
3. ***Thay thế định nghĩa macro:***
   * Tất cả các macro được định nghĩa bởi #define sẽ được thay thế bằng giá trị tương ứng.

**Chú ý lỗi:**

* Nếu xảy ra lỗi trong giai đoạn này, lỗi thường liên quan đến:
  + Chỉ thị tiền xử lý sai cú pháp.
  + Đường dẫn tệp header không đúng.
* Khó xác định chính xác vị trí lỗi vì đầu ra từ giai đoạn này được đưa thẳng vào trình biên dịch.

#### Cách sử dụng công cụ tiền xử lý

1. **Sử dụng cờ tiền xử lý trong GCC:**
   * Nếu bạn dùng GCC, bạn có thể tạo tệp pre-processed bằng lệnh:

|  |
| --- |
| gcc -E source.c -o source.i |

Trong đó:

-E là tùy chọn để yêu cầu GCC chỉ thực hiện giai đoạn tiền xử lý mà không biên dịch.

1. **Kiểm tra mã sau tiền xử lý:**
   * Mở tệp **.i** và kiểm tra mã C thuần túy đã loại bỏ chỉ thị tiền xử lý và chú thích.

#### Lợi ích của việc hiểu Pre-processor

* **Debugging:** Hiểu rõ các lỗi liên quan đến macro, include, hoặc cấu hình tiền xử lý.
* **Tối ưu hóa mã:** Sử dụng macro và include hiệu quả để giảm lỗi và tăng tính linh hoạt.
* **Phát triển thư viện:** Hiểu được cách tiền xử lý làm việc giúp bạn xây dựng các thư viện và mã nguồn có khả năng tái sử dụng cao hơn.

### Compiler

**A close-up of a sign

Description automatically generated**

Trong giai đoạn này, mã C được chuyển đổi thành mã hợp ngữ (**assembly**) cụ thể cho kiến trúc mục tiêu bởi trình biên dịch (**compiler**).

#### Tổng quan

* **Tệp đầu vào:** Tệp đã được tiền xử lý (.i).
* **Tệp đầu ra:** Tệp mã hợp ngữ (.s hoặc .asm).

#### Chức năng chính của Compiler

* Chuyển đổi mã C cấp cao thành mã hợp ngữ.
* Các dòng mã C không ánh xạ trực tiếp tới các dòng mã hợp ngữ, mà được chuyển thành các chuỗi lệnh hợp ngữ đơn giản hơn.
* Quá trình chuyển đổi này bao gồm 4 quá trình chung:
  + **Phân tích mã nguồn ( Phân tích từ vựng &Phân tích cú pháp & Phân tích ngữ nghĩa)**
  + **Tạo mã trung gian.**
  + **Tối ưu hóa.**
  + **Sinh mã hợp ngữ.**

Để hiểu rõ hơn về những gì thực sự xảy ra trong giai đoạn này, chúng ta cần nắm được cách trình biên dịch hoạt động và các thành phần của nó.

#### Thành phần của Compiler

A diagram of a computer component

Description automatically generated with medium confidence

**A. Front End**

* Phần đầu tiên của trình biên dịch được gọi là **Front End** (giao diện trước). Trong đó, **Front End** thực hiện việc phân tích mã nguồn để đảm bảo chương trình hợp lệ về cú pháp và ngữ nghĩa.
* Đây là phần đầu tiên của trình biên dịch, nơi mã nguồn của bạn được kiểm tra và phân tích để đảm bảo nó **hợp lệ**.
* Mục tiêu là biến mã nguồn của bạn thành một dạng trung gian mà trình biên dịch có thể làm việc dễ dàng hơn.

Có 3 bước chính trong phần này:

1. **Phân tích từ vựng (Lexical Analysis)**

* Tách mã nguồn thành các **Token** như từ khóa, toán tử, và biến.
* Phát hiện lỗi cơ bản như ký tự không hợp lệ.

1. **Phân tích cú pháp (Syntax Analysis):**

* Kiểm tra các **token** có tuân theo quy tắc ngữ pháp của ngôn ngữ C hay không.
* Xây dựng **cây cú pháp** (Syntax Tree) để biểu diễn cấu trúc chương trình.**Lỗi cú pháp:** Thiếu dấu ngoặc, lỗi cấu trúc lệnh.

1. **Phân tích ngữ nghĩa (Semantic Analysis)**
   * Kiểm tra ý nghĩa logic của mã nguồn, ví dụ: kiểu dữ liệu, phạm vi biến, hoặc phép toán không hợp lệ.
   * Quản lý thông tin về biến và hàm trong một bảng được gọi là **symbol table**.

**VD:**

* + Sử dụng biến chưa khai báo.
  + Sử dụng kiểu dữ liệu không tương thích (ví dụ: cộng số với chuỗi).

**B. Back End**

Back End chịu trách nhiệm chuyển đổi mã nguồn được phân tích thành mã máy cụ thể và tối ưu hóa hiệu năng.

1. **Code Generation (Sinh mã):**

* Chuyển đổi **cây cú pháp** hoặc **mã trung gian** thành mã hợp ngữ (assembly code) hoặc mã máy.
* Kết quả phụ thuộc vào kiến trúc phần cứng của hệ thống mục tiêu.

1. **Optimization (Tối ưu hóa):**

* Tối ưu mã để giảm kích thước hoặc tăng tốc độ thực thi.
* Các kỹ thuật bao gồm:
  + Gộp hàm (inline expansion).
  + Loại bỏ mã không sử dụng (dead code removal).
  + Sử dụng thanh ghi hiệu quả hơn (register allocation).

#### Quy trình hoạt động của Compiler

A diagram of a computer process

Description automatically generated

* **Giai đoạn đầu tiên** trong phần **Front End** của trình biên dịch là **quét văn bản đầu vào** và thực hiện **phân tách thành các token** (Tokenization). Trong quá trình này, trình biên dịch xác định các token như từ khóa (**keywords**), định danh (**identifiers**), toán tử (**operators**), và hằng số (**literals**). Sau đó, các token này được gửi đến công cụ phân tích cú pháp (**parsing tool**) để đảm bảo rằng chúng được tổ chức theo các quy tắc của ngôn ngữ C, nhằm tránh lỗi cú pháp trong trình biên dịch.
* **Giai đoạn thứ hai** trong phần **Front End** của trình biên dịch là kiểm tra xem câu lệnh đã được phân tích cú pháp có mang ý nghĩa chính xác hay không. Nếu việc kiểm tra ngữ nghĩa (**semantic check**) thất bại, bạn sẽ nhận được lỗi ngữ nghĩa (**Semantic Error**).
* Một trong những nhiệm vụ quan trọng của **phân tích ngữ nghĩa** là xử lý tất cả các biến có trong chương trình. Để thực hiện điều này, trình biên dịch duy trì thông tin về tất cả các biến đã được khai báo trong một cấu trúc gọi là **bảng ký hiệu** (**symbol table**). Sau khi biến được xác định, nó sẽ được gán các thuộc tính như kiểu (**type**), phạm vi (**scope**), và các đặc điểm khác.
* Khi câu lệnh được xác định là **có ý nghĩa ngữ nghĩa** và chính xác, trình biên dịch sẽ thực hiện hành động tiếp theo, đó là dịch câu lệnh này sang dạng biểu diễn nội bộ gọi là **biểu diễn trung gian** (**Intermediate Representation** - IR). Mục tiêu của bước này là chuyển các cấu trúc ngôn ngữ cấp cao, bất kể ngôn ngữ lập trình nào, sang một dạng gần với hợp ngữ (**assembly**) hơn, giúp biên dịch các ngôn ngữ khác nhau trên các kiến trúc mục tiêu khác nhau.

#### Các loại lỗi ngữ nghĩa (Semantic Errors):

* **Biến chưa được khai báo**: Khi một biến được sử dụng mà chưa được khai báo trước đó.
* **Biến không có sẵn trong phạm vi**: Biến đã được khai báo nhưng không thuộc phạm vi hiện tại (**scope**) nên không thể sử dụng.
* **Kiểu dữ liệu không tương thích**: Ví dụ, nếu một biến được tra cứu và biến đó có kiểu ký tự (**character**), thì việc sử dụng biến đó trong một phép toán cộng là không hợp lệ. Điều này bởi vì không có ý nghĩa khi cộng một ký tự với một giá trị khác.

Phần thứ hai của trình biên dịch được gọi là Back End: nơi diễn ra quá trình tối ưu hóa mã và tạo mã thực thi.

* Giai đoạn đầu tiên trong phần Back End của trình biên dịch là tối ưu hóa mã. Hiện nay, các trình biên dịch đủ thông minh không chỉ để biên dịch mã mà còn cung cấp một số tính năng tối ưu hóa.

**Optimisation:**

* **Tối ưu hóa (Optimization):**
* Tối ưu hóa chuyển đổi mã thành một dạng **tương đương về mặt chức năng** nhưng nhỏ hơn hoặc nhanh hơn.
* Quá trình tối ưu hóa thường diễn ra qua nhiều cấp độ. Một số tối ưu hóa phổ biến bao gồm:
  + **Mở rộng nội tuyến (Inline expansion)**: Thay thế lời gọi hàm bằng nội dung của hàm để giảm chi phí gọi hàm.
  + **Loại bỏ mã thừa (Dead code removal)**: Xóa bỏ các đoạn mã không cần thiết, không ảnh hưởng đến kết quả của chương trình.
  + **Mở rộng vòng lặp (Loop unrolling)**: Tăng hiệu suất bằng cách giảm số lần lặp.
  + **Cấp phát thanh ghi (Register allocation)**: Sử dụng các thanh ghi CPU một cách hiệu quả để lưu trữ giá trị tạm thời.
* Có nhiều hình thức tối ưu hóa khác nhau, tất cả đều nhằm biến đổi mã thành một phiên bản **nhỏ hơn, nhanh hơn**, nhưng vẫn **tương đương về mặt chức năng** với mã ban đầu.
* **Giai đoạn cuối cùng trong phần Back End của trình biên dịch là tạo mã (Code Generation):**
* Trong giai đoạn này, trình biên dịch chuyển đổi cấu trúc mã trung gian đã được tối ưu hóa thành mã hợp ngữ (**assembly code**). Quy trình này sẽ được giải thích chi tiết bên dưới.

#### Tạo mã (Code Generation):

* **Tạo mã** chuyển đổi cấu trúc mã trung gian đã được tối ưu hóa (**Intermediate Representation - IR**) thành các mã lệnh máy gốc (**native opcodes**) cho nền tảng mục tiêu.
* Sau khi quá trình tạo mã hoàn tất, trình biên dịch phân bổ bộ nhớ cho mã và dữ liệu trong các **phân đoạn bộ nhớ** (**Sections**).
  + Mỗi phân đoạn chứa một loại thông tin khác nhau và được xác định bởi **tên** hoặc các **thuộc tính** của thông tin lưu trữ trong đó.

Để hiểu cách trình biên dịch phân bổ bộ nhớ cho mã và dữ liệu, trước tiên chúng ta cần biết về các **phân đoạn bộ nhớ** và nội dung của chúng.

#### Phân bổ bộ nhớ (Memory Allocation):

* Trình biên dịch C phân bổ bộ nhớ cho mã và dữ liệu trong các **phân đoạn (Sections)**.
  + Mỗi phân đoạn chứa một loại thông tin khác nhau.
  + Các phân đoạn có thể được xác định bởi **tên** và/hoặc các **thuộc tính**, giúp xác định loại thông tin được lưu trữ bên trong.
* Thông tin thuộc tính này sẽ được sử dụng bởi **Linker** để định vị các phân đoạn trong bộ nhớ (chi tiết sẽ được giải thích sau).

**3. Mã (Code)**

* Các mã lệnh máy (**opcodes**) được tạo ra bởi trình biên dịch sẽ được lưu trữ trong một phân đoạn bộ nhớ riêng, thường được gọi là **.code** hoặc **.text**.
* Phân đoạn này chứa các lệnh thực thi của chương trình, là phần được CPU xử lý trực tiếp khi chạy.

A computer program code with text

Description automatically generated

**1. Dữ liệu tĩnh (Static Data):**

Vùng dữ liệu tĩnh thực tế được chia nhỏ thành hai phân đoạn sau:

* ▪ Một phân đoạn cho các biến **không khởi tạo** (ví dụ: int iVar1;)
* Một phân đoạn cho các biến **đã khởi tạo** (ví dụ: int iVar2 = 10;).

Do đó, không có gì bất ngờ nếu địa chỉ của iVar1 và iVar2 không liền kề nhau trong bộ nhớ.

Phân đoạn cho các **biến không khởi tạo** thường được gọi là phân đoạn **.bss** hoặc **ZI (Zero-Initialized)**.  
Phân đoạn cho các **biến đã khởi tạo** thường được gọi là phân đoạn **.data** hoặc **RW (Read-Write)**.

A computer code with text

Description automatically generated with medium confidence

**1. Hằng số (Constants):**

Hằng số có thể xuất hiện dưới hai dạng:

▪ **Đối tượng hằng số do người dùng định nghĩa** (ví dụ: const int c;).  
▪ **Literals** (các “số ma thuật” – **magic numbers**, định nghĩa bằng macro hoặc chuỗi ký tự).

* **Mô hình truyền thống của C** thường đặt các đối tượng hằng số do người dùng định nghĩa vào phân đoạn **.data**, cùng với các biến tĩnh không phải hằng số (**non-const statics**).
  + Do đó, chúng có thể không thực sự bất biến – đây là lý do tại sao C không cho phép sử dụng hằng số nguyên để khởi tạo các mảng, chẳng hạn.
* **Literals** thường được đặt trong phân đoạn **.text** hoặc **.code**.
  + Hầu hết các trình biên dịch sẽ tối ưu hóa các literals số và sử dụng trực tiếp giá trị của chúng khi có thể.

Nhiều bộ công cụ C hiện đại (**C toolchains**) hỗ trợ một phân đoạn riêng, gọi là **.const** hoặc **.rodata** (Read-Only Data), dành riêng cho các giá trị hằng.

* Phân đoạn này có thể được đặt trong bộ nhớ chỉ đọc (**ROM**) riêng biệt với phân đoạn **.data**.
* Tuy nhiên, điều này là một phần mở rộng công cụ (toolchain extension), không phải là một phần bắt buộc trong tiêu chuẩn C.

A computer code with a line connected to it

Description automatically generated with medium confidence

**Biến tự động (Automatic Variables):**

Phần lớn các biến được định nghĩa trong hàm và được phân loại là **biến tự động**. Điều này bao gồm:

▪ Các **tham số** (parameters).  
▪ Bất kỳ **đối tượng trả về tạm thời** (**Temporary-Returned-Object - TRO**) từ một hàm không phải void.

* **Mô hình mặc định** trong lập trình nói chung là bộ nhớ cho các đối tượng chương trình này được cấp phát từ **ngăn xếp (stack)**.
  + Đối với **tham số** và **TRO**, bộ nhớ thường được cấp phát bởi hàm gọi (bằng cách đẩy giá trị lên ngăn xếp).
  + Đối với các đối tượng cục bộ, bộ nhớ được cấp phát khi hàm được gọi.
* Một tính năng quan trọng của mô hình này là nó cho phép một hàm có thể tự gọi chính nó (**đệ quy – recursion**).
  + Tuy nhiên, đệ quy thường là một ý tưởng không tốt trong lập trình nhúng vì có thể gây ra vấn đề tràn ngăn xếp (**stack overflow**).
* Trong mô hình này, **bộ nhớ tự động** được thu hồi bằng cách **popping (loại bỏ)** ngăn xếp khi hàm thoát.

Điều quan trọng cần lưu ý là trình biên dịch **không tạo ra phân đoạn .stack**. Thay vào đó, các mã lệnh (**opcodes**) được tạo ra sẽ truy cập bộ nhớ tương đối so với một thanh ghi, **Stack Pointer** (con trỏ ngăn xếp), được cấu hình khi chương trình khởi động để trỏ đến đỉnh của phân đoạn ngăn xếp (xem chi tiết bên dưới).

Tuy nhiên, trên hầu hết các vi điều khiển hiện đại, đặc biệt là các kiến trúc RISC 32-bit, các biến tự động thường được lưu trữ trong các **thanh ghi tạm thời (scratch registers)** nếu có thể, thay vì lưu trên ngăn xếp.  
Ví dụ: **Tiêu chuẩn Gọi Thủ tục Kiến trúc ARM (ARM Architecture Procedure Call Standard - AAPCS)** định nghĩa các thanh ghi CPU nào được sử dụng cho:

* **Tham số hàm (function call arguments)**,
* **Kết quả trả về từ hàm**,
* Và các **biến cục bộ**.

A screenshot of a computer program

Description automatically generated

1. **Dynamic data:**

A screenshot of a computer program

Description automatically generatedMemory for dynamic objects is allocated from a section known as the *Heap*. As with the Stack, the **Heap không được trình biên dịch (compiler) cấp phát tại thời điểm biên dịch (compile-time)**, mà được **Linker** cấp phát tại thời điểm liên kết (**link-time**).

1. **Object files:**

 Trình biên dịch tạo ra các **tệp đối tượng có thể di dời** (**relocatable object files**) với phần mở rộng **.o**.

 Tệp đối tượng chứa mã nguồn đã biên dịch – bao gồm **các mã lệnh (opcodes)** và **các phân đoạn dữ liệu (data sections)**.

 Lưu ý rằng tệp đối tượng chỉ chứa các phân đoạn dành cho **biến tĩnh (static variables)**. Tại giai đoạn này, vị trí của các phân đoạn chưa được cố định.

Tệp .o chưa phải là tệp thực thi, vì mặc dù một số phần tử đã được xác định cố định (ví dụ: mã lệnh, địa chỉ tương đối so với PC, hằng số “ngay lập tức”), nhưng:

* + Địa chỉ của các biến **tĩnh** và **toàn cục** chỉ được biết dưới dạng **offset** (độ lệch) từ vị trí bắt đầu của các phân đoạn tương ứng.
  + Địa chỉ được định nghĩa trong các module khác hoàn toàn chưa được biết, ngoại trừ qua **tên**.

Tệp đối tượng chứa hai bảng – **Imports** và **Exports**:

▪ **Exports:**

* Chứa các định danh extern được định nghĩa trong đơn vị dịch thuật này (không bao gồm các biến tĩnh – **statics**).

▪ **Imports:**

* Chứa các định danh được khai báo (và sử dụng) trong đơn vị dịch thuật này nhưng không được định nghĩa trong đó.

A screenshot of a computer

Description automatically generated

Note the identifier names are in *name-mangled* form.

### Assembler:

A pink rectangular sign with black text

Description automatically generated

Trong giai đoạn này, mã hợp ngữ (assembly code) được sinh ra bởi trình biên dịch sẽ được chuyển đổi thành mã đối tượng (object code) bởi trình hợp dịch (assembler).

* **Tệp đầu vào** cho giai đoạn này là tệp có phần mở rộng \*.asm.
* **Tệp đầu ra** là tệp có phần mở rộng \*.o hoặc \*.obj.
* Lưu ý rằng ngày nay, các trình biên dịch hiện đại có thể tạo ra mã đối tượng mà không cần sử dụng trình hợp dịch độc lập.
* **Kết quả đầu ra** của giai đoạn này là tệp mã đối tượng chứa các mã máy (opcodes) và các phần dữ liệu (data sections).

Sau khi quá trình sinh mã kết thúc, trình biên dịch sẽ phân bổ bộ nhớ cho mã và dữ liệu trong các phân đoạn; mỗi phân đoạn chứa thông tin khác nhau và được định nghĩa bởi tên hoặc thuộc tính của thông tin được lưu trữ trong đó.

Để có thể hiểu cách trình biên dịch phân bổ bộ nhớ cho mã và dữ liệu, trước tiên chúng ta cần biết các đoạn bộ nhớ khác nhau và nội dung mà chúng chứa.

**Tệp mã đối tượng (Object file)** chứa các phần dành cho các biến tĩnh (static variables), những biến này khả dụng trong suốt thời gian chạy của chương trình.

**Bảng ký hiệu (Symbol table)** được sử dụng để lưu trữ tất cả các tên biến và các thuộc tính của chúng.

**Phần thông tin gỡ lỗi (Debug info section)** có chứa ánh xạ giữa mã nguồn ban đầu và thông tin cần thiết để trình gỡ lỗi (debugger) hoạt động.

**Phần xuất (Exports section)** chứa các ký hiệu toàn cục, bao gồm các hàm hoặc biến toàn cục, mà chương trình cung cấp.

**Phần nhập (Imports section)** chứa các tên ký hiệu mà chương trình cần từ các tệp mã đối tượng khác.

**Phần xuất, nhập và bảng ký hiệu** được trình liên kết (linker) sử dụng trong giai đoạn liên kết để kết hợp các tệp mã đối tượng lại thành một tệp thực thi duy nhất.

### Linker:

A diagram of a linker

Description automatically generated

A diagram of a linker

Description automatically generated

Trong giai đoạn này, các tệp mã đối tượng khác nhau được sinh ra bởi trình hợp dịch sẽ được chuyển đổi thành một tệp có thể định vị lại (relocatable file) bởi trình liên kết (linker).

* **Tệp đầu vào** cho giai đoạn này là tệp \*.o và các thư viện tiêu chuẩn C.
* **Tệp đầu ra** là tệp có thể định vị lại (relocatable file).
* Khi kết hợp các tệp mã đối tượng lại với nhau, trình liên kết thực hiện các thao tác sau:

1. **Giải quyết ký hiệu (Symbol resolution).**
2. **Định vị lại (Relocation).**
3. **Giải quyết ký hiệu (Symbol resolution)**

Trong chương trình gồm nhiều tệp, nếu có bất kỳ tham chiếu nào đến các nhãn (labels) được định nghĩa trong tệp khác, trình hợp dịch sẽ đánh dấu các tham chiếu này là “chưa được giải quyết” (unresolved). Khi các tệp này được đưa vào trình liên kết, trình liên kết sẽ xác định giá trị của các tham chiếu này từ các tệp mã đối tượng khác và sửa đổi mã với các giá trị chính xác. Nếu trình liên kết không tìm thấy tham chiếu đến các nhãn này trong bất kỳ tệp mã đối tượng nào, nó sẽ đưa ra lỗi liên kết: **“unresolved reference to variable”**.

Nếu trình liên kết phát hiện cùng một ký hiệu được định nghĩa trong hai tệp mã đối tượng, nó sẽ báo lỗi **“redefinition”**.

1. **Định vị lại (Relocation):**

Định vị lại là quá trình thay đổi các địa chỉ đã được gán cho các nhãn (labels). Quá trình này cũng bao gồm việc sửa đổi tất cả các tham chiếu để phản ánh địa chỉ mới được gán.

Chủ yếu, việc định vị lại được thực hiện vì hai lý do sau:

1. **Gộp các phân đoạn (Section Merging):** Kết hợp các phân đoạn từ các tệp mã đối tượng khác nhau để chúng có thể hoạt động như một đơn vị duy nhất.
2. **Sắp xếp phân đoạn (Section Placement):** Đặt các phân đoạn tại các vị trí bộ nhớ thích hợp để đảm bảo chương trình hoạt động chính xác khi được thực thi.
3. **Section Merging**:

A diagram of a computer program

Description automatically generated Trong mỗi tệp, trình liên kết (linker) chịu trách nhiệm hợp nhất các phần (sections) từ các tệp đầu vào thành các phần trong tệp đầu ra.

Theo mặc định, các phần có cùng tên từ mỗi tệp sẽ được đặt liên tiếp nhau, và các tham chiếu đến các nhãn (labels) sẽ được điều chỉnh để phản ánh địa chỉ mới.

**Định vị Vị trí Phần (Section Placement):**

Khi một chương trình được lắp ráp, mỗi phần (section) được giả định bắt đầu từ địa chỉ 0. Do đó, các nhãn (labels) được gán giá trị tương đối so với điểm bắt đầu của phần đó.

Khi tệp thực thi cuối cùng được tạo, phần này được đặt tại một địa chỉ cụ thể **X**.  
Tất cả các tham chiếu đến các nhãn được định nghĩa trong phần đó sẽ được tăng thêm một giá trị **X** để trỏ đến vị trí mới chính xác.

A screenshot of a computer

Description automatically generated

1. **Locator:**

A diagram of a linker script

Description automatically generated

Trong giai đoạn này, quá trình gán các địa chỉ vật lý cho tệp có thể di chuyển được (relocatable file) được tạo ra từ trình liên kết (linker) sẽ được thực hiện bằng cách sử dụng một công cụ định vị (locator).

* **Tệp đầu vào** cho giai đoạn này là tệp có thể di chuyển được và tệp script của trình liên kết (linker script file).
* **Tệp đầu ra** là tệp thực thi (executable file).
* **Công cụ định vị (locator)** là một công cụ thực hiện việc chuyển đổi từ chương trình có thể di chuyển được sang hình ảnh nhị phân thực thi (executable binary image).
* Tệp script của trình liên kết cung cấp cho công cụ định vị thông tin cần thiết về cấu trúc bộ nhớ thực tế, sau đó công cụ định vị thực hiện quá trình chuyển đổi để tạo ra một tệp nhị phân thực thi duy nhất.
* Lưu ý rằng công cụ định vị có thể được tìm thấy như một công cụ độc lập hoặc được tích hợp cùng với trình liên kết.

**6. Tệp Script của Trình Liên Kết (Linker Script File):**

Tệp Script của trình liên kết, hay tệp cấu hình liên kết (**Linker Configuration File - LCF**), có nhiệm vụ hướng dẫn trình định vị (**locator**) cách ánh xạ chương trình thực thi vào các địa chỉ phù hợp.

**Đặc điểm chính:**

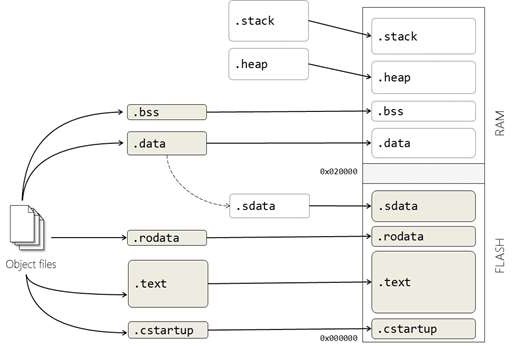
* **Kiểm soát bố cục bộ nhớ** của tệp đầu ra bằng cách cung cấp thông tin về bộ nhớ trên bo mạch mục tiêu làm đầu vào cho trình định vị.
* **LCF xác định bố cục bộ nhớ vật lý** (ví dụ: Flash/SRAM) và vị trí của các vùng chương trình khác nhau.
* **LCF phụ thuộc nhiều vào trình biên dịch**, do đó mỗi trình biên dịch sẽ có định dạng riêng.

A screenshot of a computer program

Description automatically generated

**Tệp cấu hình trình liên kết** được hiển thị ở trên dẫn đến một cách bố trí bộ nhớ khá điển hình như sau:

* **.cstartup** – mã khởi động hệ thống (**system boot code**) – được đặt rõ ràng ở đầu bộ nhớ Flash.
* **.text** và **.rodata** được đặt trong bộ nhớ Flash, vì chúng cần phải tồn tại lâu dài (**persistent**).
* **.stack** và **.heap** được đặt trong RAM.
* **.bss** được đặt trong RAM, nhưng ở thời điểm này (có thể) rỗng. Nó sẽ được khởi tạo với giá trị 0 trong quá trình khởi động (**start-up**).
* **.data** được đặt trong RAM (dành cho thời gian chạy – **run-time**), nhưng phân đoạn khởi tạo của nó, **.sdata**, nằm trong ROM.



The Linker will perform checks to ensure that your code and data sections will fit into the designated regions of memory.

Loading:

**ELF** hoặc **DWARF** là các định dạng tệp đầu ra độc lập với mục tiêu (**target-independent output file formats**).

Để có thể tải lên hệ thống mục tiêu, tệp **ELF** phải được chuyển đổi sang định dạng bộ nhớ flash / PROM gốc, thường là **.bin** hoặc **.hex**.

A diagram of a computer program

Description automatically generated

Kết quả cuối cùng của quá trình xây dựng C sẽ là một hình ảnh nhị phân thực thi từ mã nguồn, sẵn sàng để tải lên hệ thống nhúng mục tiêu.

**Những điểm chính:**

* ▪ Trình biên dịch tạo ra mã lệnh (opcodes) và phân bổ dữ liệu từ các tệp mã nguồn để tạo ra một tệp đối tượng (object file).  
  ▪ Trình biên dịch làm việc với một đơn vị dịch thuật (translation unit) tại một thời điểm.  
  ▪ Trình liên kết (linker) nối các tệp đối tượng và tệp thư viện để tạo thành một chương trình.  
  ▪ Trình liên kết chịu trách nhiệm phân bổ các phân đoạn ngăn xếp (stack) và bộ nhớ tự do (free store).  
  ▪ Hoạt động của trình liên kết được kiểm soát bởi một tệp cấu hình (configuration file), duy nhất dành cho hệ thống mục tiêu.  
  ▪ Các tệp đã liên kết phải được dịch sang một định dạng phụ thuộc vào mục tiêu để có thể tải lên hệ thống.