**BÀI 1: COMPILER – MACRO.**

1. **Compiler**

* Quá trình biên dịch là quá trình chuyển đổi mã nguồn (source code) của một chương trình viết bằng ngôn ngữ lập trình cấp cao (C/Cpp, Java) thành mã máy (machine code) mà CPU có thể hiểu và thực thi. Quá trình này gồm nhiều bước chính

1. Preprocessing: tiền xử lí
   * Giai đoạn này chỉ áp dụng cho các ngôn ngữ có bộ tiền xử lí như C và C++
   * Bộ tiền xử lí thực hiện các thao tác tước khi mã nguồn được biên dịch, bao gồm:
     + Xử lí các chỉ thị tiền xử lí (#include, #define,, #ifndef,…): ví dụ thay thế các macro bằng giá trị thực của chúng
     + Nhúng nội dung của các tệp tiêu đều (#include <stdio.h>) vào ãm nguồn
     + Loại bỏ các chú thích (//)

🡺 **Kết quả:**  mã nguồn sau khi tiền xử lý (preprocessed code), thường được lưu dưới dạng tệp .i hoặc .ii

1. **Compilation: (Biên dịch)**
   * Ở bước này, trình biên dịch sẽ dịch mã nguồn đã tiền xử lý sang mã Assembly (ngôn ngữ bậc thấp hơn gần với mã máy)
   * Quá trình này bao gồm:
     + Phân tích cú pháp: tách mã nguồn thành các token
     + Phâp tích cú pháp: kiểm tra cú pháp của mã nguồn, đảm bảo nó tuân theo quy tắc của ngôn ngữ lập trình
     + Phân tích ngữ nghĩa: Kiểm tra lỗi về kiểu dữ liệu, phạm vi biến, tính hợp lệ của toán tử,..
     + Tạo mã trung gian: chuyển mã nguồn thành mọt dạng trung gian dễ tối ưu hóa
     + Tối ưu hóa mã: Loại bỏ các đoạn mã dư thừa, cỉa thiện hiệu suất thực thi

🡺 Kết quả: tệp mã Assembly (.s)

1. **Assembly: dịch dang mã máy**
   * Mã Assembly được chuyển hành mã máy bằng trình biên dịch
   * Giai đoạn này tạp ra một tệp mã đối tượng (.o hoặc .obj), chứa mã máy nhưng chửa thể chạy độc lập vì chưa được liên kết với thư viện và các tệp khác

* **Kết quả**: tệp đối tượng .o hoặc .obj

1. **Linking:** liên kết
   * Sau khi biên dịch mã nguồn thành các tệp đối tượng (.o, .obj), chúng ta chưa thế chạy chương trình ngay lập tức. Lý do là các tệp nàu có thể chứa các tham chiếu đến  **các hàm trong tệp khác, thư viện chuẩ, biến toàn cục hoặc hàng số được khai báo nhưng chưa có địa chỉ cố định.**
   * Trình liên kết (Linker) kết hợp các tệp mã đối tượng (.o) với nhau và các thư viện tiêu chuẩn (như libc, libm) để tạo ra một tệp thực thi hoàn chỉnh.
   * Nếu chương trình gọi các hàm từ thư viện động (dynamic libraries), trình liên kết sẽ chỉ tham chiếu đến thư viện thay vì nhúng trực tiếp mã của nó.

* **Kết quả:** tệp thực thi (.exe , .out, hoặc mmojt tệp chạy được trên hệ điều hành)

1. **MARCO**

| **Loại** | **Mô tả** |
| --- | --- |
| * **Chỉ thị include** | Nhúng tệp tiêu đề (#include) |
| * **Chỉ thị define** | Định nghĩa macro (#define) |
| * **Chỉ thị điều kiện** | Kiểm soát biên dịch (#if, #ifdef, #ifndef, #else, #elif, #endif) |
| * **Chỉ thị undef** | Hủy bỏ macro (#undef) |
| * **Chỉ thị pragma** | Điều khiển trình biên dịch (#pragma) |
| * **Chỉ thị error & warning** | Báo lỗi hoặc cảnh báo (#error, #warning) |

| **Toán Tử** | **Tên** | **Mô tả** |
| --- | --- | --- |
| # | Stringizing | Chuyển đối số thành chuỗi |
| ## | Token Pasting | Nối các token lại với nhau |
| \_\_VA\_ARGS\_\_ | Variadic Macro | Xử lý danh sách đối số biến đổi |
|  |  |  |

**Variadic macro:**

* Là một dạng macro cho phép nhận một số lượng biến tham số có thể thay đổi
* Giúp định nghĩa các macro có thể xử lý một lượng biến đầu vào khác nhau

*#define MACRO\_NAME(…)*

**BÀI 2: THƯ VIỆN STDARG – ASSERT.**

1. **Thư viên stdarg**

* Va\_list: là một kiểu dữ liệu để đại diện cho danh sách các đối số biế đổi.
* Va\_start: Bắt đầu một danh sách đối số biến đổi. Nó cần được gọi trước khi truy cập các đối số biến đổi đầu tiên.
* Va\_arg: truy cập một dối số trong danh sách. Hàm này nhận một đối số của kiểu được xác định bởi tham số thứ 2. Khi gọi hàm này lần 1 thì đối số thứ 1 sẽ xuất hiện gọi lần 2 thì đối số thứ 2,…
* Va\_end: kết thúc việc sử dụng danh sách đối số biến đổi. Nó cần được gọi trước khi kết thúc hàm.

1. Thư viện assert

* Cung cấp macro assert
* Macro này được sử dụng để kiểm tra một điều kiện
* Nếu điều kiện đúng, không có gì xảy ra và chương trình tiếp tục thực thi.
* Nếu điều kiện sai, chương trình dừng lại và thông báo một thong điệp lỗi
* Dùng trong debug.

**BÀI 3: BITMASK.**

1. Khái niệm cơ bản.

* Bitmask là một kỹ thuật sử dụng các bit để lưu trữ và thao tác với các cờ hoặc trạng thái. Có thể sử dụng bitmask để đặt, xóa và kiểm tra trạng thái của các bit cụ thể trong một từ
* Bitmask thường được sử dụng để tối ưu hóa bộ nhớ, thực hiện các phép toán logic trên một cum bit và quản lý các trạng thái, quyền truy cập, hoặc các thuộc tính khác của một đối tượng.

1. Các toán tử bitwise
2. AND (&)
3. OR ( | )
4. XOR ( ^ )
5. NOT ( ~ )
6. Shift left và right ( << và >> )

**BÀI 4: POINTER**

1. Pointer

* Trong ngôn ngữ lập trình C, con trỏ là một biến chứa địa chỉ bộ nhớ của một đối tượng khác (biến, mảng, hàm). Việc sử dụng con trỏ giúp chúng ta thực hiện các thao tác trên bộ nhớ một cách linh hoạt hơn.

1. Void pointer:
   * Thường dùng để trỏ tới bất kỳ đại chỉ nào mà không cần biết tới kiểu dữ liệu của giá trị tại địa chỉ đó.
     + - Void \*ptr\_void;
2. Function pointer
   * Pointer to function là một biến mà giữ địa chỉ của một hàm. Có nghĩa là, nó tro đến vùng nhớ trong bộ nhớ chứa mã máy của hàm được định nghĩa trong chương trình.
   * Trong ngôn ngữ lập trình C, con trỏ hàm cho phép bạn truyền một hàm như là một đối số cho một hàm khác, lưu trữ địa chỉ của hàm trong một cấu trúc dữ liệu, hoặc thậm chí truyền hàm như một giá trị trả về từ một hàm khác.
     + - **<return\_type> (\*func\_ptr)(data\_type1,data\_type2);**
3. **Pointer to constant**
   * là cách định nghĩa một con trỏ không thể thay đổi giá trị tại địa chỉ mà nó trỏ đến thong qua dereference nhưng giá trị tại đại chỉ đó có thể thay đổi.
   * int const \*ptr\_const;
   * const int \*ptr\_const;
4. **Constant Pointer**
   * Định nghĩa là một con trỏ mà giá trị nó trỏ đến (địa chỉ) là không thể thay đổi. Tức là khi con trỏ này được khởi tạo thì nó sẽ không thể trỏ tới địa chỉ khác
   * Int \*const const\_ptr=&value;
5. **Pointer to Pointer**
   * Con trỏ đến con trỏ là một kiểu dữ liệu trong ngôn ngữ lập trình cho phép bạn lưu trữ địa chỉ của một con trỏ. Con trỏ đến con trỏ cung cấp một cấp bậc con trỏ mới, cho phép bạn thay đổi giá trị của con trỏ gốc. Cấp bậc này có thể hữu ích trong nhiều tình huống, đặc biệt là khi bạn làm việc với các hàm cần thay đổi giá trị con trỏ.
6. NULL Pointer
   * Là một con trỏ không trỏ đến bất kì đới tượng hoặc vùng nhớ cụ thể nào. Trong ngôn ngữ lập trình C, một con trỏ có thể được gán giá trị NULL để biểu diễn trạng thái NULL
   * Sử dụng NULL ointer thường hữu ích để kiểm tra xem một con trỏ đã được khởi tạo và có trỏ đến một vùng nhớ hợp lệ chưa. Tránh dereferencing (sử dụng giá trị mà con trỏ trỏ đến) một null pointer là quan trọng để tránh lỗi chương trình.

**BÀI 5: EXTERN -STATIC – VOLATILE - REGISTER**

1. **Extern**.
   * Khái niệm extern trong ngôn ngư lập trình C được sử dụng để thông báo rằng một biến hoặc hàm đã được khai ở một nơi khác trong chương trình hoặc trong một file nguồn khác. Điều này giúp chương trình hiểu rằng biến hoặc hàm đã được định nghĩa và sẽ được sử dụng từ một vị trí khác, giúp quản lý sự liên kết giữa các phần khác nhau của chương trình hoặc giữa các file nguồn.
2. **Static.**
3. **Static local**
   * + Khi static được sử dụng với local variables (biến cụ bộ - khai báo biến trong một hàm), nó giữ giá trị của biến qua các lần gọi hàm và giữ phạm vi của biến chỉ trong hàm đó.
4. **Static global**
   * + Khi static được sử dụng với phạm vi global variables (biến toàn cục – khai báo biến bên ngoài hàm), nó hạn chế phạm vi của biến đó chỉ trong file nguồn hiện tại,
     + ứng dụng: dùng để thiết kế thư viện
5. **static trong class**
   * + Khi một thành viên của lớp được khai báo là static, nó thuộc về lớp chứ không thuộc về các đối tượng cụ thể của lớp đó. Các đối tượng của lớp sẽ chia sẻ cùng một bản sao của thành viên static, và nó có thể được truy cập mà không cần tạo đối tượng. Nó thường được sử dụng để lưu trữ dữ liệu chung của tất cả đối tượng.
6. **Volatile.**
   * Từ khóa volatile trong ngôn ngưỡ lập trình C/C++ được sử dụng để báo hiệu cho trình biên dịch rằng một biến có thể thay đổi ngẫu nhiên, ngoài sự kiểm soát của chương trình. Việc này ngăn chặn trình biên dịch tối ưu hóa hoặc cóa bỏ các thao tác trên biến đó, giữ cho các thao tác trên biến được thực hiện như đã được định nghĩa.
7. **Register.**
   * Trong ngôn ngữ lập trình C, từ khóa register được sử dụng để chỉ ra ý muốn của lập trình viên tằng một biến được sử dụng thường xuyên và có thể được lưu trữ trong một thanh ghi máy tính, chứ không phải trong RAM. Việc này nhằm tăng tốc độ truy cập. Tuy nhiên, Lưu ý rằng việc sử dụng register chỉ là một đề xuất cho trình biên dịch và không đảm bảo rằng biến sẽ được lưu trữ trong thanh ghi. Trong thực tế, trình biên dịch có thể quyết định không tuân thủ lời đề xuất này.

**Bài 6: GOTO-SETJUMP**

1. **Goto**

* Go to là một từ khóa trong ngôn ngữ lập trình C, cho phép chương trình nhảy đến một nhãn (label) đã được đặt trước đó **trong cùng một hàm**. Mặc dù nó cung cấp khả năng kiểm soát flow của chương trình, nhưng việc sử dụng goto thường được xem là không tốt vì nó có thể làm cho mã nguồn trở nên khó đọc và khó bảo trì.

1. **Setjump.**

* Setjump.h là một thư viện trong ngôn ngữ lập trình C, cung cấp hai hàm chính là setjmp và longjmp. Cả hai hàm này thường được sử dụng để thực hiện xử lý ngoại lệ trong C, mặc dù nó không phải là một cách tiêu biểu để xử lý ngoại lệ trong ngôn ngữ này
* Setjmp(jmp\_buf env): lưu trạng thái của chương trình vào biến env để có thể quay lại sau
* Longjmp(jmp\_buf env, int val): nhảy trở lại điểm đã lưu bằng setjmp và tiếp tục thực thi từ đó, đông thời trả về giá trị val.

**BÀI 7: STRUCT – UNION**

1. **Struct.**

* Trong ngôn ngữ lập trình C, struct là một cấu trúc dữ liệu cho phép lập trình viên định nghĩa một kiểu dữ liệu mới bằng cách nhóm các biến có các kiểu dữ liệu khác nhau lại với nhau. Struct cho phép tạo ra một thực thể dữ liệu lớn hơn và có tổ chức hơn từ các thành viên của nó.
  1. Bit field
  + Trong C, “bit field” (trường bit) là một thành phần đặc biệt của cấu trúc (struct) cho phép bạn chỉ định số lượng bit cụ thể dùng để lưu trữ một biến số nguyên. That vì sử dụng toàn bộ kích thước của một kiểu dữ liệu, bạn có thể “cắt nhỏ ” bộ nhớ theo số bit cần thiết, giúp tiết kiệm không gian bộ nhớ và mô tả chính xác hơn ý nghĩa của dữ liệu
  + Không thể sử dung toán tử lấy địa chỉ & trên các thành viên bit field.

1. **Union.**

* Trong ngôn ngữ lập trình C, union là một cấu trúc dữ liệu giúp lập trình viên kết hợp nhiều kiểu dữ liệu khác nhau vào cùng một vùng nhớ. Mục đích chính của union là tiết kiệm bộ nhớ bằng cách **chi sẻ cùng một vùng nhớ** cho các thành viên của nó. Điều này có nghĩa là, trong một thời điểm, chỉ có 1 thành viên của union có thể được sử dụng. Điều này ứng dụng nhằm tiết kiệm bộ nhớ.

**BÀI 8: MEMORY LAYOUT**

* + 1. **Memory layout**
* Chương trình main.exe (trên window), main.hex (nạp vào vi điều khiển) được lưu ở bộ nhớ SSD hoặc FLASH. Khi nhấn run chương trình trên window (cấp nguồn cho vi điều khiển) thì những chương trình này sẽ được copy vào bộ nhớ RAM để được thực thi.

1. Text segment (Code segment)
   * Mã máy: chứa tập hợp các lệnh thực thi.
   * Quyền truy cập: Text segment thường có quyền đọc và thực thi, nhưng không có quyền ghi.
   * Lưu hằng số toàn cục (const) , chuỗi hằng – string literal (Clang- macOS, windows - minGW)
   * Tất cả các biến được lưu ở phân vùng Text đều không thể thay đổi giá trị mà chỉ đọc.
2. Data segment.
   * Initiazed Data Segment (Dữ liệu đã khởi tạo):
   * Chứa các biến toàn cục được khởi tạo với giá trị khác 0.
   * Chứa các biến static (global+local) được khởi tạo với giá trị khác 0.
   * Quyền truy cập là đọc và ghi, tức là có thể đọc và thay đổi giá trị của biến.
   * Tất cả các biến sẽ được thu hồi sau khi chương trình kết thúc.
3. Bss segment
   * Uninitialized Data Segment (Dữ liệu Chưa khởi tạo)
   * Chứa các biến toàn cục khởi tạo với giá trị bừng 0 hoặc không gán giá trị.
   * Chứa các biến static với giá trị khởi tạo bằng 0 hoặc không gán giá trị.
   * Quyền truy cập là đọc và ghi, tức là có thể đọc và thay đổi giá trị của biến.
   * Tất cả các biến sẽ được thu hồi sau khi chương trình kết thúc.
4. Stack
   * Chứa các biến cụ bộ (trừ static cục bộ), tham số truyền vào.
   * Hằng số cục bộ, có thể thay đổi thông qua con trỏ.
   * Quyền truy cập: đọc và ghi, nghĩa là có thể đọc và thay đổi giá trị của biến trong suốt thời gian chương trình chạy.
   * Sau khi ra khỏi hàm, **tự động thu hồi vùng nhớ.**
5. Heap:
   * Cấp pát động
   * Heap được sử dụng để ấp phát bộ nhớ độ trong quá trình thực thi của chương trình.
   * Điều này cho phép chương trình tạo ra và giải phóng bộ nhớ theo nhu cầu, thích ứng với sự biến đổi của dữ liệu trong quá trình chạy.
   * Các hàm như malloc(),calloc(),free(),realloc() được sử dụng để cấp phá và giải phóng bộ nhớ trên heap
   * Quyền truy cập: có quyền đọc và ghi, nghĩa là có thể đọc và thay đổi giá trị của biến trong suốt thời gian chương trình chạy.
6. Stack và heap.
   * Bộ nhớ Stack được dùng để lưu trữ các biến cục bộ trong hàm, tham số truyền vào… Truy cập vào bộ nhớ này rất nhanh và được thực thi khi chương trình được biên dịch
   * Bộ nhớ heap được dùng để lưu trữ vùng nhớ cho những biến được cấp phát động bởi các hàm malloc-calloc-realloc.
   * Stack: vùng nhớ Stack được quản lý bởi hệ điều hành, dữ liệu được lưu trong Stack sẽ tự động giải phóng khi hàm thực hiện trong công việc của mình.
   * Heap: Vùng nhớ Heap được quản lí bởi lập trình viên (C/C++), dữ liệu trong Heap sẽ không bị phân hủy khi hàm thực hiện xong, điều đó có nghĩa bạn phải tự tay giải phóng vùng nhớ bằng câu lệnh free (trong C), nếu không sẽ xảy ra hiện tượng rò rỉ bộ nhớ.
   * Stack: bời vì bộ nhớ Stack cố định nên nếu chương trình bạn sử dụng quá nhiều bộ nhớ vượt quá khả năng lưu trữ của Stack chắc chắn sẽ xảy ra tình trạng tràn bộ nhớ Stack (Stack overflow), các trường hợp xảy ra như bạn khởi tạo quá nhiều biến cục bộ, hàm đệ quy vô tận,…
   * Heap: Nếu bạn liên tục cấp phát vùng nhớ mà không giải phóng thì sẽ bị lỗi tràn vùng nhớ Heap (Heap overflow). Nếu bạn khởi tạo một vùng nhớ quá lớn mà vùng nhớ heap không thể lưu trữ một lần được sẽ bị lỗi khởi tọa vùng nhớ Heap thất bại.

* Malloc(size\_t size) cấp phát một khối bộ nhớ có kích thước size byte.
  + Nếu cập phát thành công trả con trỏ dến vùng nhớ mới, thất bại trả về NULL
  + Dữ liệu trong vùng nhớ do malloc() cấp phát không được khởi tạo về 0, mà có thể chứa giá trị rác.
    - * Int \*ptr=(int\*)malloc(5\*sizeof(int))
* Calloc(size\_t num, size\_t size) cấp phát bộ nhớ cho num phần tử, mỗi phần tử có kích thước size. Bộ nhớ được khởi tạo về 0
  + - * Int \*ptr=(int\*)calloc(5,sizeof(int))
* Realloc(void \*ptr, size\_t new\_size) thay đổi kích thước của bộ nhớ đã cấp phát bởi malloc() hoặc calloc()
  + Nếu mở rông, dữ liệu của được giữ nguyên, phần mở rộng có thể chứa dữ liệu rác.
  + Nếu giảm kích thước, dữ liệu bị cắt bớt.
  + Nếu ptr ==NULL, realloc() hoạt động như malloc(new\_size)
    - * Int \*ptr=(int\*)malloc(5\*sizeof(int))
      * ptr=(int\*)realloc(ptr,5\*sizeof(int))
* Free(void \*ptr) giải phóng bộ nhớ đã cấp phát trước đó bằng malloc(), calloc() hoặc realloc().