

Ngôn ngữ lập trình C


Bài 11. Xử lý tiền biên dịch

Soạn bởi: TS. Nguyễn Bá Ngọc

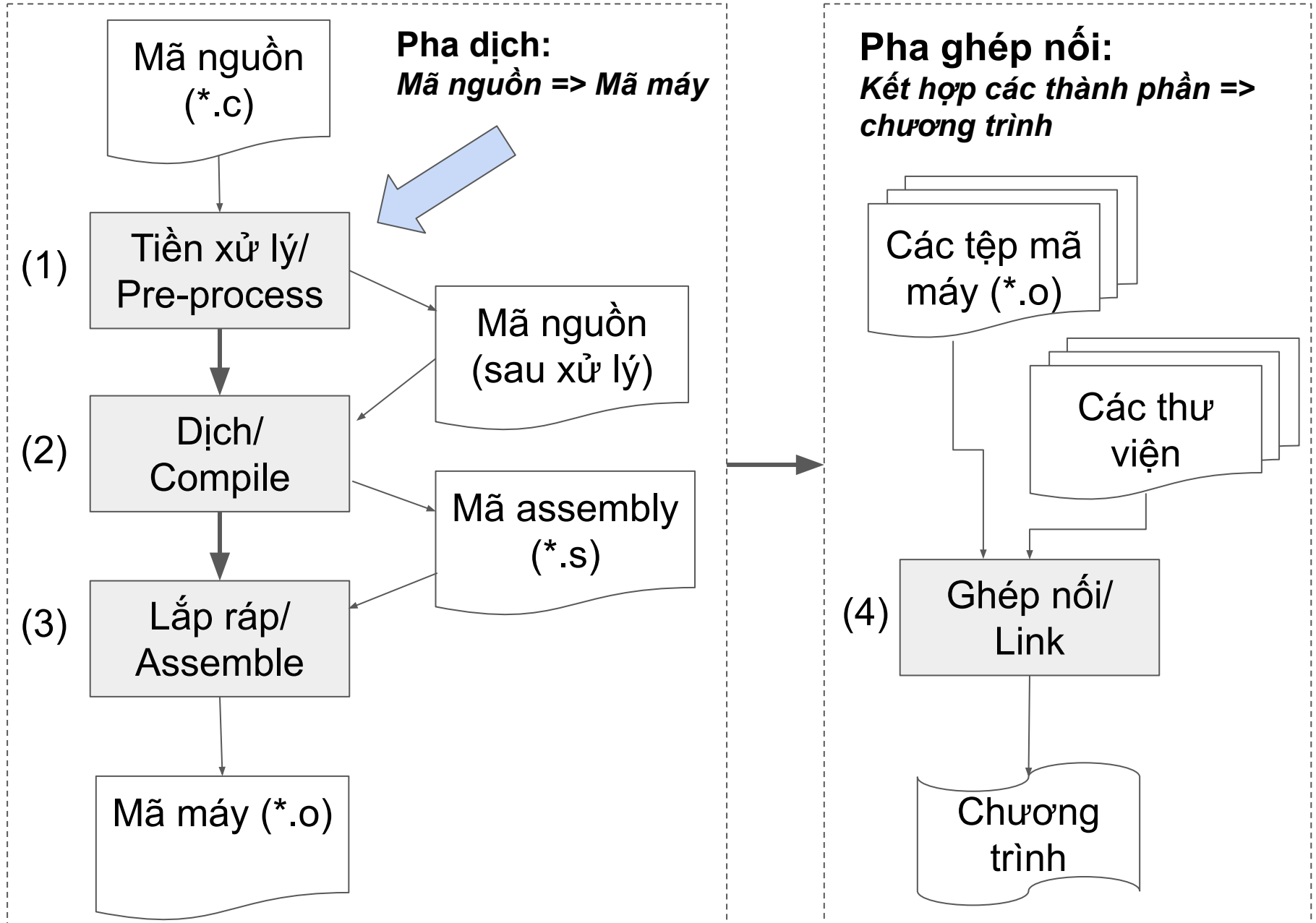
Nội dung

1. Các lệnh tiền biên dịch
2. Rẽ nhánh biên dịch
3. Một số ví dụ tổng hợp

Nội dung

- 
1. Các lệnh tiền biên dịch
 2. Rẽ nhánh biên dịch
 3. Một số ví dụ tổng hợp

Tiến trình biên dịch chương trình C với GCC



Macro đối tượng

Macro đối tượng về hình thức là định danh, không có tham số:

#define định-danh nội-dung-thay-thế

Ví dụ:

```
#define N 100
```

```
#define M 200
```

```
#define MN (M * N)
```

Các định nghĩa đơn giản thường kết thúc trong phạm vi 1 dòng.

!Lưu ý: Lệnh tiền biên dịch không có dấu ; ở cuối

Trình xử lý tiền biên dịch **thay** định-danh bằng nội-dung-thay-thế

```
#define N 100
```

```
int a[N]; => int a[100]; // N được thay bằng 100
```

Với

```
#define N 100;
```

```
int a[N]; => int a[100;]; // Lỗi
```

*Chạy lệnh với khóa -E để xem kết quả tiền xử lý:
gcc -E tệp.c*

Đặt tên cho hằng giá trị giúp mã nguồn dễ đọc & dễ quản lý hơn

Macro hàm

Macro với tham số có hình thức giống hàm (và cách sử dụng cũng có nhiều điểm tương đồng), vì vậy được gọi là Macro hàm

`#define tên-macro(x1, x2, ..., xn) nội-dung-thay-thế`

Không được có dấu cách ở vị trí này

Danh sách tham số

Lưu ý: Không được có dấu cách giữa tên-macro và dấu mở ngoặc của danh sách tham số (được nhận diện là Macro đối tượng nếu có).

Ví dụ:

```
#define MAX(x, y) ((x) < (y)? (y): (x))
```

```
int a = 10, b = 20; x được thay thế bằng a; y được thay thế bằng b  
MAX(a, b) được thay thế bằng ((a) < (b)? (b): (a))
```

```
int c = MAX(a, b); => c = ((a) < (b)? (b): (a));
```

```
#define SUM (x, y) (x + y)
```

```
c = SUM(a, b); // => c = (x, y) (x + y)(a, b); // Lỗi do SUM  
được nhận diện như Macro đối tượng
```

Macro hàm: Một số quy tắc hữu ích

Nên sử dụng các tham số như các biểu thức cơ bản trong phần nội-dung-thay-thế.

Ví dụ: Xét 1 triển khai chưa hoàn thiện của Macro lấy giá trị cực đại của 2 giá trị:

Các tham số x và y nên được đưa vào trong cặp ()
`#define EMAX(x, y) x < y? y: x`

Xét 1 số trường hợp sử dụng EMAX:

```
int a = 10, b = 20;
```

```
int c = EMAX(a, b); // => c = a < b? b: a; // OK
```

```
int d1 = EMAX(EMAX(a, b), c); // NOK != Max(a, b, c)
```

```
int d2 = EMAX(c, EMAX(a, b)); // NOK
```

Kiểm thử MAX với các trường hợp đã nêu

```
#define MAX(x, y) ((x) < (y)? (y): (x))
```

Macro hàm: Một số quy tắc hữu ích⁽²⁾

Phần nội dung thay thế nếu chứa toán tử thì nên được đặt trong cặp dấu () như biểu thức cơ bản

Ví dụ: Xét 1 triển khai chưa hoàn thiện của Macro tính tổng

```
#define ESUM(x, y) x + y
```

```
int a = 10, b = 20, c = 30;
```

Nên đưa vào trong ()

```
int d = ESUM(a, b); // => d = a + b; // OK
```

```
d = ESUM(a, b) * c; // d = a + b * c; // NOK - Thường không  
phải biểu thức cần tính do độ ưu tiên của các toán tử.
```

Sử dụng biểu thức cơ bản giúp kiểm soát được vấn đề với độ ưu tiên của các toán tử có trong tham số của Macro hoặc trong biểu thức hoàn chỉnh mà Macro là 1 phần của nó.

Viết lại EMAX và ESUM với các biểu thức cơ bản:

```
#define MAX(a, b) ((a) < (b)? (b): (a))
```

```
#define SUM(a, b) ((a) + (b))
```


Macro hàm: Một số quy tắc hữu ích⁽³⁾

Nên sử dụng các định danh khác biệt cho các đối tượng được định nghĩa trong Macro (nếu có) để tránh xung đột tên với phạm vi bên ngoài Macro.

Ví dụ:

```
#define print_array(a, n) \
    do { \
        for (int i = 0; i < n; ++i) { \
            printf(" %d", a[i]); \
        } \
    } while (0)
```

Nên sử dụng tên đặc biệt để tránh vô tình che lấp tên bên ngoài, ví dụ `_i` (giả sử người sử dụng không đặt tên biến bắt đầu với dấu gạch nối `_`)

Nếu phần nội dung thay thế của Macro có nhiều dòng, thì tất cả các dòng ngoại trừ dòng cuối cùng phải kết thúc bằng dấu `\`

```
int a[] = {1, 2, 3}, total = 0, i = 3;
```

```
print_array(a, 3); // 1 2 3: OK
```

```
print_array(a, i); // NOK: Không có giá trị nào được xuất ra
```

Macro hàm vs. Hàm

Ưu điểm của Macro:

- Nhanh hơn (1 chút, do không mất chi phí gọi hàm)
- Khái quát hơn (do được chèn vào ngữ cảnh sử dụng và có thể thao tác với chính các đối số, không yêu cầu thông tin về kiểu dữ liệu).

Nhược điểm của Macro:

- Có thể làm tăng kích thước mã nguồn (do nội dung thay thế có thể được lặp nhiều lần)
- Có thể dẫn đến lỗi khó tìm nếu sử dụng không đúng cách.

Các toán tử tiền biên dịch

Các toán tử #, và ## được xử lý ở bước tiền biên dịch và có thể được sử dụng trong nội dung của Macro

- Toán tử chuỗi hóa: #
 - Chuyển đổi tham số Macro thành hằng chuỗi ký tự (đặt trong " "), ví dụ:
 - `#define PRINT_INT(n) printf(#n " = %d", n)`
 - `int a = 10;`
 - `PRINT_INT(a); => printf("a " = %d", a); // Xuất ra màn hình a = 10`
- Toán tử ghép nối: ##
 - Ghép nối 2 từ thành 1, ví dụ:
 - `#define MAKE_ID(n) id##n`
 - `MAKE_ID(1) => id1`
 - `MAKE_ID(2) => id2`

Một số tính chất của Macro

- Macro có thể chứa Macro khác:
 - `#define MY_PI 3.14159` *Macro được xử lý nhiều lượt cho tới khi nội dung thu được không chứa Macro*
 - `#define MY_PI2 (MY_PI / 2)` *MY_PI2 => (MY_PI / 2) => (3.14159 / 2)*
 - `double angle = MY_PI2; // => (3.14159 / 2)`
- Trình xử lý tiên biên dịch thay thế nguyên 1 đơn vị (từ), không thay thế 1 phần của 1 đơn vị:
 - `printf("MY_PI = %f\n", MY_PI);`
 - MY_PI là 1 phần của 1 hằng chuỗi ký tự, vì vậy không được thay thế*
 - MY_PI là 1 tham số của hàm, vì vậy được thay thế*
 - `=> printf("MY_PI = %f\n", 3.14159);`
- Macro có hiệu lực từ sau `#define` cho tới `#undef` hoặc kết thúc đơn vị biên dịch, không phụ thuộc vào cấu trúc khối.
 - `#define tên-macro ...`
 - *`/* Phạm vi mà Macro được thay thế */`*
 - `#undef tên-macro // Hoặc kết thúc đơn vị biên dịch.`

Một số tính chất của Macro₍₂₎

- Định nghĩa Macro nhiều lần trong cùng 1 đơn vị biên dịch có thể phát sinh lỗi
 - Ví dụ trường hợp chèn nhiều tệp tiêu đề khác nhau (với `#include`) cùng định nghĩa 1 Macro
 - Các định nghĩa phải có *nội dung tương đương*,
 - Trình biên dịch thường đưa ra cảnh báo.
 - Người lập trình có thể `#undef` (vô hiệu) Macro hiện có trước khi định nghĩa lại Macro.

Một số Macro được định nghĩa sẵn

| Tên | Ý nghĩa |
|----------|---|
| __LINE__ | Chỉ số dòng trong chứa __LINE__ |
| __FILE__ | Tên tệp được biên dịch |
| __DATE__ | Ngày biên dịch (định dạng Mm dd yyyy) |
| __TIME__ | Thời gian biên dịch (định dạng hh:mm:ss) |
| __STDC__ | 1 nếu trình biên dịch tương thích với quy chuẩn C |

Thường được sử dụng để xuất thông tin hỗ trợ tìm lỗi, ví dụ:
`fprintf(stderr, "Phát sinh lỗi ở %s:%d\n", __FILE__, __LINE__);`

Chèn tệp

Lệnh chèn tệp (#include) được xử lý ở bước tiền biên dịch (lệnh tiền biên dịch), và có hai định dạng thường gặp:

`#include "đường-dẫn-tệp"`

`#include <đường-dẫn-tệp>`

Phạm vi tìm kiếm theo định dạng `#include <đường-dẫn-tệp>` là tập con của phạm vi tìm kiếm theo `#include "đường-dẫn-tệp"`. Vì vậy định dạng `#include "đường-dẫn-tệp"` còn được gọi là định dạng tìm kiếm mở rộng. Nếu không tìm thấy trong phạm vi mở rộng thì tiếp tục xử lý như `#include <đường-dẫn-tệp>`.

Định dạng mở rộng (") thường được sử dụng cho các tệp tiêu đề trong thư mục dự án, còn định dạng mặc định (<>) thường được sử dụng cho các tệp trong các thư mục hệ thống.

Triển khai trình biên dịch tự thiết lập quy tắc tìm kiếm cụ thể. ¹⁵

Quy tắc tìm tệp được chèn với GCC

Mặc định:


- Đối với `#include "đường-dẫn-tệp"`: Trước tiên tìm trong thư mục chứa tệp đang được xử lý, sau đó tìm trong các *đường dẫn tiêu chuẩn của hệ thống* (như định dạng `<>`).
- Đối với `#include <đường-dẫn-tệp>`: Chỉ tìm trong các *đường dẫn tiêu chuẩn của hệ thống*.

Có thể bổ xung các đường dẫn tìm kiếm với khóa `-I`:

- Các đường dẫn được bổ xung vào trong câu lệnh biên dịch (`-I thư-mục`) được tìm sau thư mục hiện hành.
- Có thể xem chi tiết các thư mục tìm kiếm bằng cách bổ xung tham số `-v` (verbose) vào câu lệnh biên dịch, ví dụ:
 - `gcc -v -o prog main.c`

[<https://gcc.gnu.org/onlinedocs/cpp/Search-Path.html>]

Nội dung

1. Các lệnh tiền biên dịch
 2. Rẽ nhánh biên dịch
 3. Một số ví dụ tổng hợp
- 

Rẽ nhánh biên dịch

Có thể lựa chọn sử dụng hoặc loại bỏ 1 phần mã nguồn với các lệnh rẽ nhánh ở bước tiền xử lý

Lệnh rẽ nhánh `#if ... #endif`, cú pháp:

`#if biểu-thức-hằng`

`/ Phần mã nguồn được sử dụng nếu biểu-thức-hằng đúng */`*

`#endif`

Ví dụ:

`#define DEBUG 1`

`#if DEBUG`

`printf("Đã nhập x = %d\n", x);`

`#endif`

Với `#define DEBUG 1` thì biến `x` được xuất ra,

Với `#define DEBUG 0` thì biến `x` không được xuất ra.

←
Trong ví dụ này câu lệnh `printf(...)` được bỏ qua nếu `DEBUG` không được định nghĩa hoặc được định nghĩa = 0

Toán tử tiền biên dịch defined

Toán tử defined được sử dụng để kiểm tra 1 Macro đã được định nghĩa hay chưa, ví dụ:

| | | |
|-------------------------------------|------|--------------------------------------|
| <code>#if defined(định-danh)</code> | Hoặc | <code>#if !defined(định-danh)</code> |
| <code>/*...*/</code> | | <code>/*...*/</code> |
| <code>#endif</code> | | <code>#endif</code> |

Chúng ta cũng có thể sử dụng các cú pháp thu gọn:

`#ifdef định-danh` Tương đương với `#if defined(định-danh)`
và

`#ifndef định-danh` Tương đương với `#if !defined(định-danh)`

Lệnh tiền biên dịch #elseif và #else

```
#if biểu-thức-hằng-1
/* ... */
#elseif biểu-thức-hằng-2
/* ... */
#else
/* ... */
#endif
```

Có lô-gic hoạt động tương tự

```
if (biểu-thức-1) {
    /* ... */
} else if (biểu-thức-2) {
    /* ... */
} else {
    /* ... */
}
```

...

Nhưng được thực hiện ở pha tiền biên dịch và được sử dụng để lựa chọn các phần mã nguồn.

Một số ứng dụng tiêu biểu

Rõ nhánh biên dịch thường được sử dụng để:

- Kiểm tra & tránh chèn 1 tệp tiêu đề nhiều lần, ví dụ:

```
#ifndef _STDIO_H
#define _STDIO_H    1
#endif /* <stdio.h> included. */
```

- Triển khai 1 chức năng đa nền tảng theo nhiều cách khác nhau tùy theo môi trường

```
#if defined(WIN32)
/* Triển khai cho Windows */
#elseif defined (MACOS)
/* Triển khai cho Mac */
#elseif defined (LINUX)
/* Triển khai cho Linux */
#endif
```

- V.V..

Nội dung

1. Các lệnh tiền biên dịch
2. Rẽ nhánh biên dịch
3. Một số ví dụ tổng hợp



Ví dụ 11.1. Macro

```
vd11-1.c
7  #define N 10
8  #define M 20
9  #define MN (M * N)
10
11 int a[N];
12
13 #define MAX(x, y) ((x) < (y)? (y): (x))
14 c = MAX(a, b);
15
16 #define SUM(a, b) (a + b)
17 total = SUM(a, b) * SUM(c, d);
18
19 #define print_array(a, n) \
20     do { \
21         for (int i = 0; i < n; ++i) { \
22             printf(" %d", a[i]); \
23         } \
24     } while (0)
25
26 print_array(a, n);
27 print_array(b, m);
28
29 printf("%s:%d", __FILE__, __LINE__);
```

Kết quả xử lý tiền biên dịch

gcc -E vd11-1.c

```
int a[10];

c = ((a) < (b)? (b): (a));

total = (a + b) * (c + d);
# 26 "vd11-1.c"
do { for (int i = 0; i < n; ++i) { printf(" %d", a[i]); } }
while (0);
do { for (int i = 0; i < m; ++i) { printf(" %d", b[i]); } }
while (0);

printf("%s:%d", "vd11-1.c", 29);
bangoc:$
```

Ví dụ 11.2. Rẽ nhánh biên dịch

```
vd11-2.c x
9  #include <stdio.h>
10
11  int main() {
12      int a, b;
13      printf("Nhập 2 số nguyên a và b: ");
14      scanf("%d%d", &a, &b);
15  #ifdef DEBUG
16      printf("Đã nhập: a = %d, b = %d\n", a, b);
17  #endif // DEBUG
18      int sum = a + b;
19      printf("sum = %d\n", a + b);
20      return 0;
21  }
```

```
bangoc:$gcc -DDEBUG -o prog vd11-2.c
bangoc:$./prog
Nhập 2 số nguyên a và b: 10 20
Đã nhập: a = 10, b = 20
sum = 30
bangoc:$gcc -o prog vd11-2.c
bangoc:$./prog
Nhập 2 số nguyên a và b: 10 20
sum = 30
bangoc:$
```