

Ngôn ngữ lập trình C

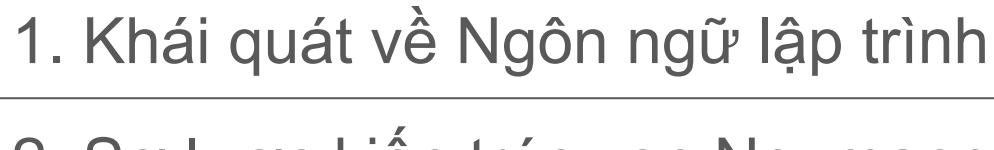
Bài 1. Tổng quan (*mở rộng*)

Soạn bởi: TS. Nguyễn Bá Ngọc

Nội dung

1. Khái quát về Ngôn ngữ lập trình
2. Sơ lược kiến trúc von Neumann
3. Lịch sử phát triển NNLT C
4. Môi trường lập trình
5. Tiến trình biên dịch với gcc

Nội dung

- 
1. Khái quát về Ngôn ngữ lập trình
 2. Sơ lược kiến trúc von Neumann
 3. Lịch sử phát triển NNLT C
 4. Môi trường lập trình
 5. Tiến trình biên dịch với gcc

Ngôn ngữ lập trình

- Không có một quan điểm thống nhất về mục đích chính của ngôn ngữ lập trình là gì? Ngôn ngữ lập trình có thể là: Công cụ ra lệnh điều khiển thiết bị tính toán; Phương tiện biểu diễn giải thuật; Công cụ để tiến hành thực nghiệm và phân tích dữ liệu; Công cụ để tự động hóa các thao tác với máy tính; Công cụ để tạo trang Web; v.v..
- Sự ảnh hưởng qua lại giữa ngôn ngữ lập trình và tư duy lập trình diễn ra theo cả 2 chiều. Ngôn ngữ lập trình thường giới hạn hệ thống khái niệm và cấu trúc có thể được sử dụng để lập trình. Tuy nhiên cũng có khi từ nhu cầu diễn đạt những ý tưởng lập trình đã dẫn đến việc sáng tạo ra những thành phần ngôn ngữ mới hay cả ngôn ngữ lập trình mới.

Ngôn ngữ lập trình₍₂₎

- Ngoài các mục đích sử dụng mà ngôn ngữ lập trình phải đáp ứng, thiết kế cơ bản của ngôn ngữ lập trình còn chịu ảnh hưởng của các yếu tố bên ngoài điển hình như:
Nguyên lý hoạt động của môi trường thực thi (phần cứng máy tính, máy ảo) và phương pháp phát triển phần mềm.
- Một vấn đề lập trình nếu có thể được giải quyết bằng một ngôn ngữ lập trình thì cũng có thể được giải quyết bằng các ngôn ngữ lập trình khác, tuy nhiên các lời giải có thể khác nhau đáng kể về độ khó và tính hiệu quả.
- Người lập trình nên học nhiều ngôn ngữ lập trình để biết nhiều khái niệm hay, để sử dụng sáng tạo và hiệu quả các công cụ để giải quyết các vấn đề lập trình.

(Các quan điểm cá nhân)₅

Phân lớp ngôn ngữ lập trình

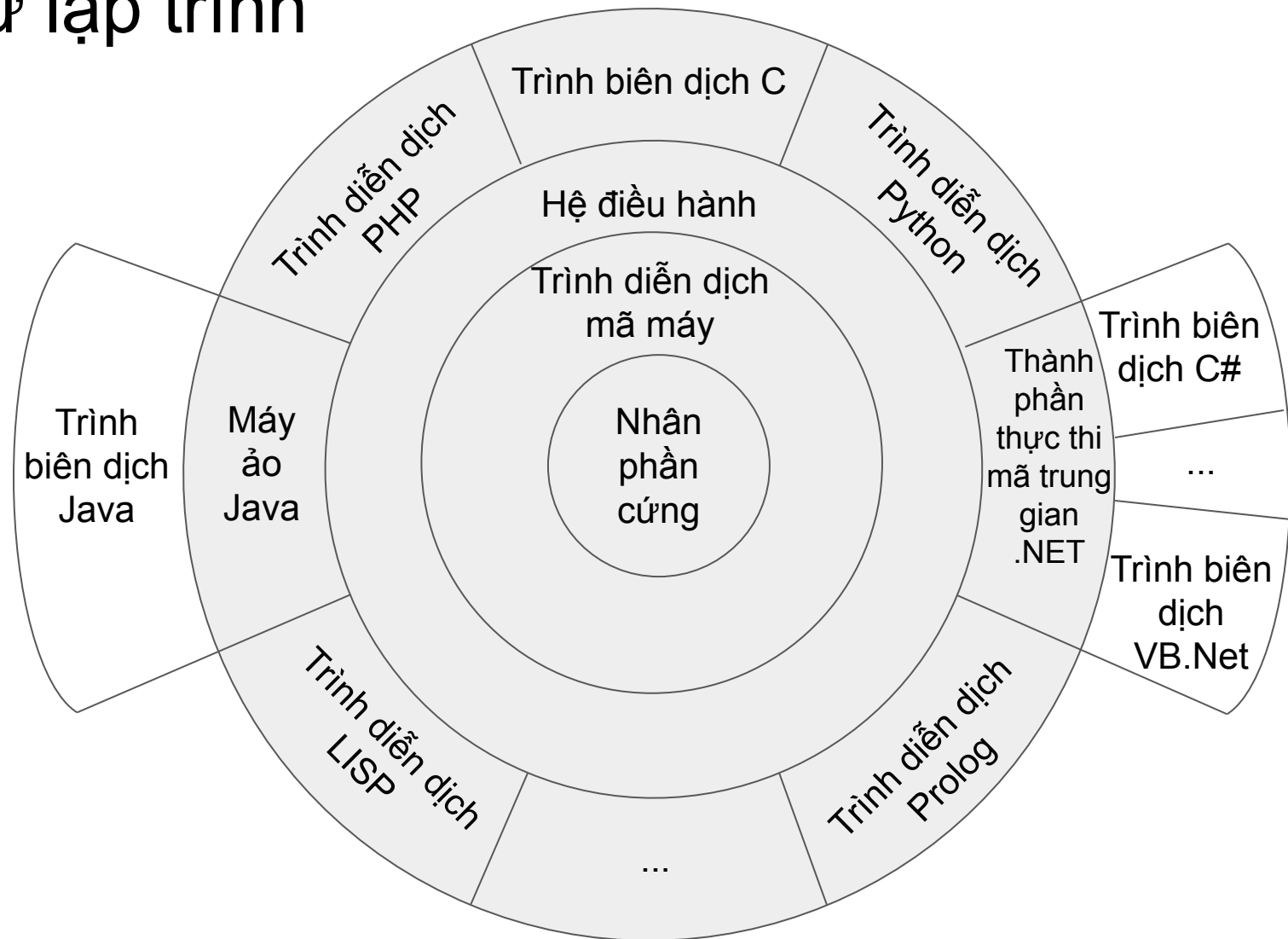
- Có thể phân lớp các ngôn ngữ lập trình theo 3 lớp chính:
 - **Mệnh lệnh/Imperative** (ví dụ C) - Có thiết kế tương thích với kiến trúc Von Neumann
 - Biến mô phỏng ô nhớ, phép gán đưa dữ liệu vào ô nhớ, chương trình mô tả chính xác trình tự thực hiện lệnh.
 - **Hàm/Functional** (ví dụ Lisp) - Áp dụng hàm cho các tham số khác nhau (không cần biến, phép gán, vòng lặp).
 - **Dựa trên luật/Logic** (ví dụ Prolog) - Công cụ thực hiện chương trình tự xác định trình tự thực hiện các mệnh đề lô-gic.
- Ngoài ra còn có các quan điểm phân lớp khác:
 - Hướng đối tượng/OOP (ví dụ Java)
 - Script (ví dụ Python, Javascript)
 - ... Tuy nhiên các ngôn ngữ Java, Python và Javascript cũng có đầy đủ các đặc trưng của lớp Mệnh lệnh.
 - Phân lớp ngôn ngữ lập trình có giới hạn mềm, một ngôn ngữ có thể thuộc nhiều hơn 1 lớp.

Các phương pháp triển khai

- Biên dịch/Compile: Chuyển đổi mã nguồn thành mã máy.
 - Ví dụ: Assembly, C, C++
- Diễn dịch/Interpret: Thực hiện bằng phần mềm
 - Ví dụ: Python, PHP
- Kết hợp biên dịch và diễn dịch:
 - Biên dịch sang mã trung gian sau đó diễn dịch bằng máy ảo - Ví dụ: Java, nền tảng .Net.
 - Biên dịch khi cần (Just-In-Time compilation, JIT) - Ví dụ: V8 (được sử dụng trong Chrome và Node.js) có cơ chế chọn lọc để biên dịch mã Javascript thành mã máy khi cần thiết nhằm đạt hiệu năng thực thi cao hơn.

Một ngôn ngữ có thể được triển khai theo nhiều cách khác nhau, ví dụ phát triển cả Trình biên dịch và Trình diễn dịch.

Các tầng hệ thống máy tính: Góc nhìn ngôn ngữ lập trình



Nội dung

1. Khái quát về Ngôn ngữ lập trình

2. Sơ lược kiến trúc von Neumann

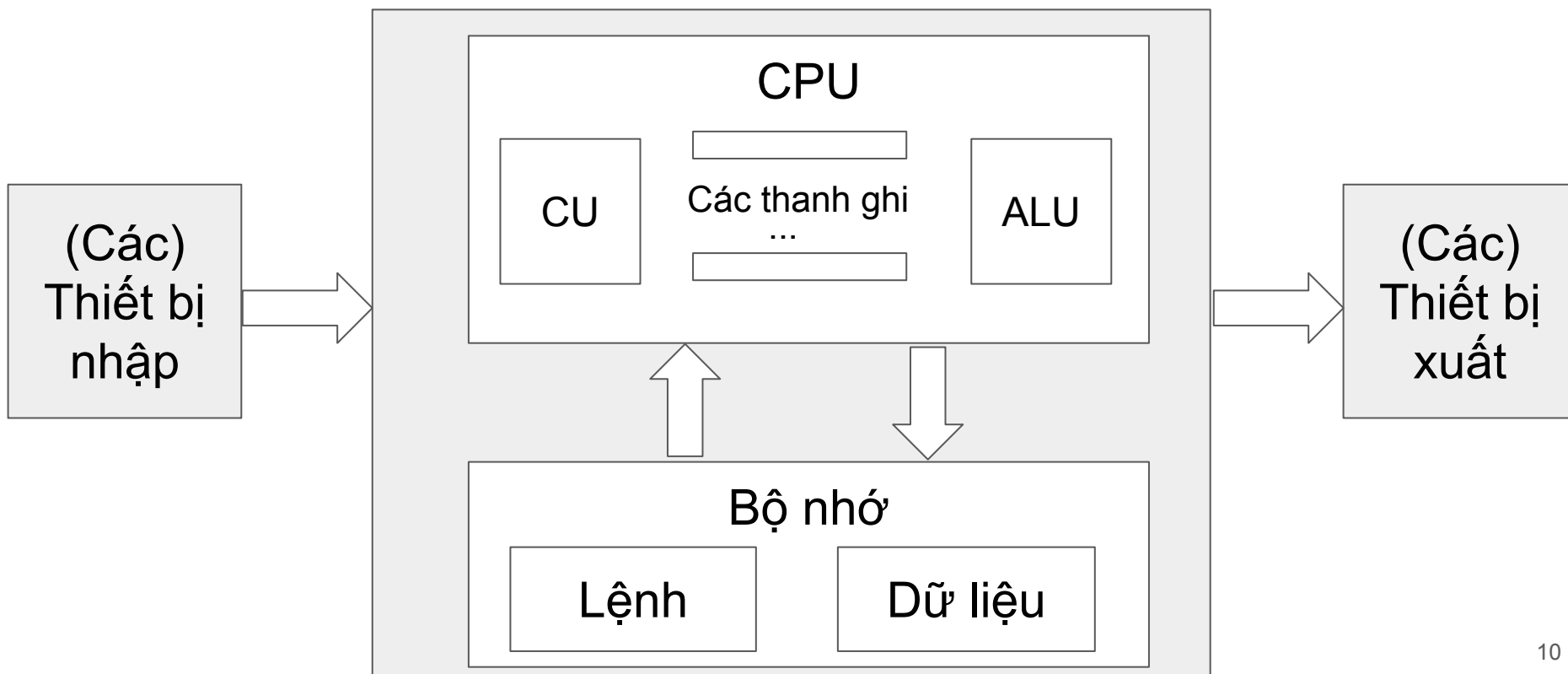
3. Lịch sử phát triển NNLT C

4. Môi trường lập trình

5. Tiến trình biên dịch với gcc

Kiến trúc Von Neumann

- Lệnh và dữ liệu có bản chất biểu diễn giống nhau và được lưu trong cùng 1 bộ nhớ.
- Bộ nhớ và CPU được tách rời nhau.
- Lệnh được xử lý về bản chất là theo tiến trình tuần tự.



Vòng lặp Nạp-Thực hiện lệnh

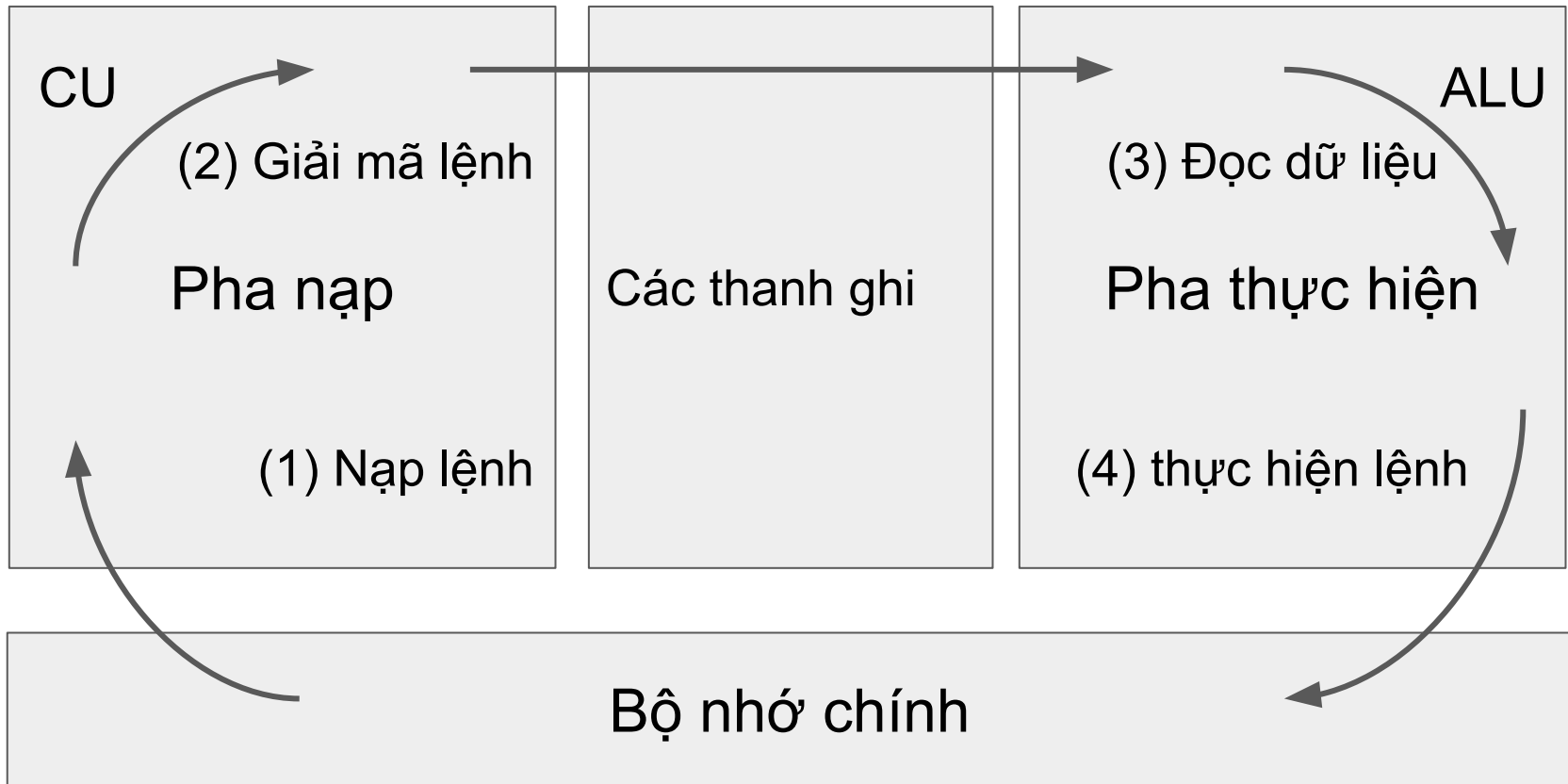
Mô tả nguyên lý thực thi chương trình theo kiến trúc von Neumann. Các bước thực hiện lệnh:

- 1) CU sao chép lệnh trong bộ nhớ có địa chỉ đang được lưu ở thanh ghi PC (**P**rogram **C**ounter) vào thanh ghi IR (**I**nstruction **R**egister);
- 2) CU chuyển đổi mã lệnh đang được lưu trong IR thành tín hiệu điều khiển;
- 3) Trong trường hợp lệnh đang được thực hiện có tham số thì CU đọc các tham số từ bộ nhớ vào các thanh ghi;
- 4) Các tín hiệu được truyền tới ALU để xử lý, kết quả thực hiện lệnh được sao chép sang bộ nhớ.

Giá trị của PC phải được cập nhật trước khi chuyển sang chu kỳ lệnh tiếp theo: Có thể được cộng thêm một hằng số thành địa chỉ của lệnh tiếp theo trong khối nhớ hoặc được cập nhật bởi lệnh vừa được thực hiện.

Sau khi hoàn thành một chu trình lệnh, chu trình lệnh tiếp theo được bắt đầu.

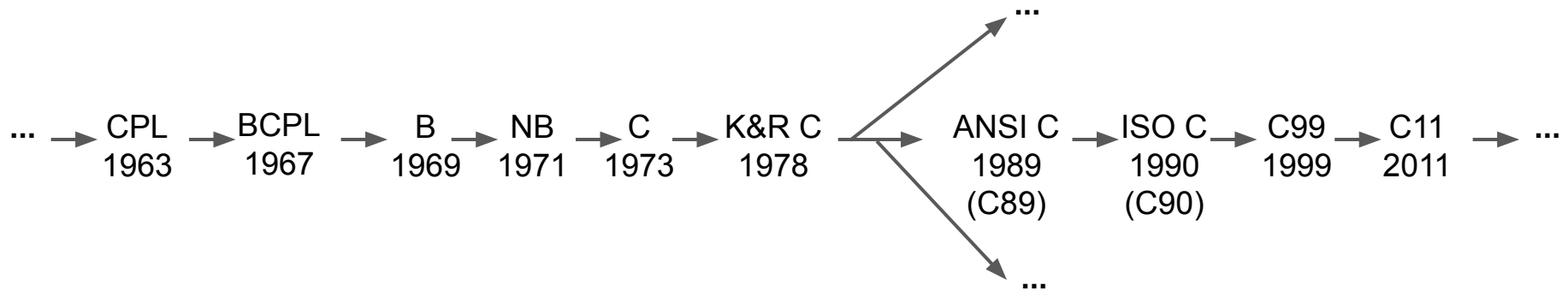
Sơ đồ vòng lặp Nạp-Thực hiện lệnh



Nội dung

1. Khái quát về Ngôn ngữ lập trình
2. Sơ lược kiến trúc von Neumann
3. Lịch sử phát triển NNLT C
4. Môi trường lập trình
5. Tiến trình biên dịch với gcc

Sơ lược lịch sử phát triển NNLT C



Lịch sử phát triển NNLT C

- Ban đầu (từ những năm 1970) C được phát triển để viết HĐH Unix
 - HĐH Unix ban đầu được viết bởi Ken Thompson bằng hợp ngữ ở phòng thí nghiệm của Bell. Sau đó ông đã quyết định viết lại HĐH Unix bằng ngôn ngữ bậc cao hơn, và ông cũng đã tạo ra ngôn ngữ B cho mục đích này.
 - Năm 1971 ngôn ngữ B bắt đầu bộc lộ các hạn chế và không đáp ứng được nhu cầu viết HĐH Unix trên PDP-11, vì vậy Dennis Ritchie sau khi gia nhập dự án Unix đã bắt đầu mở rộng B.
 - Ban đầu Ritchie gọi ngôn ngữ do mình phát triển là NB (*New B* - B mới), theo thời gian các khác biệt so với B ngày càng lớn, sau đó Ritchie đã đổi tên ngôn ngữ thành C.
 - Đến năm 1973 ngôn ngữ C đã có đủ các tính năng để có thể viết lại toàn bộ HĐH Unix bằng C.

Lịch sử phát triển NNLT C₍₂₎

- Sau đó phạm vi sử dụng C được mở rộng sang nhiều môi trường cho nhiều mục đích khác nhau và có nhiều trình biên dịch được phát triển độc lập
 - Năm 1978 tài liệu đầu tiên về NNLT C - *The C Programming Language* được viết bởi Brian Kernighan và Dennis Ritchie (phiên bản *K&R*) đã bắt đầu được xuất bản.
 - Bước sang những năm 1980, nhiều trình biên dịch C cho các môi trường khác nhau được phát triển, C được sử dụng ngày càng phổ biến cả bên ngoài cộng đồng Unix:
 - Các lập trình viên sử dụng *K&R* làm tham chiếu để viết trình biên dịch, nhưng một số tính năng ngôn ngữ được mô tả khá nhập nhằng trong *K&R* và các lập trình viên đã triển khai những tính năng đó theo cách riêng.
 - Ngoài ra C vẫn tiếp tục được phát triển sau khi *K&R* được xuất bản.
 - Những điều này đã dẫn đến nguy cơ bất tương thích giữa các trình biên dịch và làm mất tính khả chuyển của mã nguồn C.

Lịch sử phát triển NNLT C₍₃₎

- Để duy trì tính tương thích giữa các trình biên dịch khi NNLT C ngày càng được sử dụng rộng rãi hơn, các quy chuẩn cho NNLT C đã được biên soạn.
 - Quy chuẩn C của Mỹ bắt đầu được phát triển từ 1983 dưới sự điều hành của viện ANSI.
 - Viện ANSI thông qua quy chuẩn đầu tiên X3.159-1989 năm 1989, và không lâu sau đó được tổ chức ISO công nhận như quy chuẩn quốc tế ISO/IEC 9899:1990 năm 1990. Phiên bản này của C thường được gọi là C89 hoặc C90.
 - NNLT C vẫn liên tục được hiệu chỉnh, cập nhật và phát triển trong các phiên bản mới hơn:
 - ISO/IEC 9899:1999 - được gọi là C99
 - ISO/IEC 9899:2011 - C11
 - ISO/IEC 9899:2018 - C17 hoặc C18

Lịch sử phát triển NNLT C₍₄₎

- Ngoài các tính năng ngôn ngữ được mô tả trong quy chuẩn ISO, trình biên dịch còn có thể bổ xung thêm các tính năng chưa/không có trong quy chuẩn ISO. Các tính năng thêm vào được gọi là các mở rộng.
 - GCC và Clang không chỉ biên dịch mã nguồn C theo các quy chuẩn ISO, mà còn có thể biên dịch mã nguồn theo các quy chuẩn gnu:
 - gnu90 = C90 + các mở rộng
 - gnu11 = C11 + các mở rộng
 - gnu17 = C17 + các mở rộng
 - Các mở rộng, ví dụ điển hình như các mở rộng của GNU và CLang, thường có thể được mô phỏng bởi các tính năng có trong quy chuẩn (nhưng các mô phỏng có thể phức tạp hơn trong một số trường hợp).

Vì sao học lập trình C?

- Để minh họa những vấn đề lập trình đơn giản: C có thiết kế nhỏ gọn và các cấu trúc lập trình cơ bản trong C cũng rất đơn giản và dễ học.
- Để sử dụng chuyên sâu: C là ngôn ngữ chính để lập trình ở tầng cơ bản:
 - Có thể biên dịch thành mã máy tối ưu, hoạt động nhanh.
 - Có khả năng mạnh mẽ để làm việc với phần cứng, truy cập trực tiếp vào bộ nhớ của hệ thống máy tính.
 - Có thể tạo chương trình kích thước nhỏ, sử dụng hiệu quả các tài nguyên tính toán, kể cả các hệ thống máy tính có nguồn tài nguyên hạn chế.

Mục tiêu

Có khả năng viết **chương trình C** tương thích với **quy chuẩn ISO** và có tính khả chuyển.

Nội dung

1. Khái quát về Ngôn ngữ lập trình
2. Sơ lược kiến trúc von Neumann
3. Lịch sử phát triển NNLT C
4. Môi trường lập trình
5. Tiến trình biên dịch với gcc

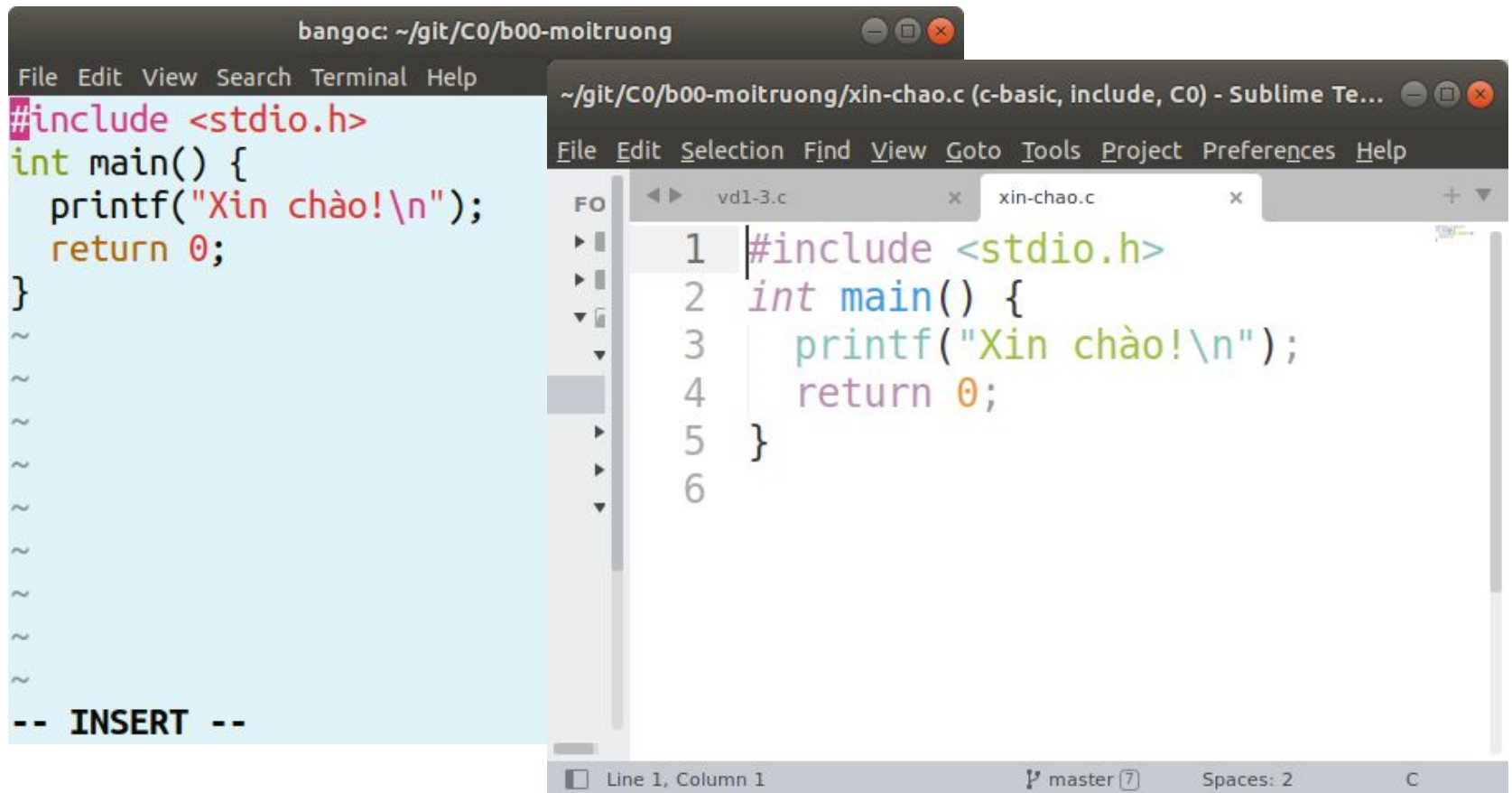


Môi trường lập trình

- Khuyến khích sử dụng hệ điều hành (dựa trên) Linux
 - Linux được sử dụng phổ biến cho các máy chủ
 - Môi trường thuận lợi để lập trình
 - Các minh họa trong quá trình học được thực hiện trong môi trường Linux
- Các trường hợp khác
 - Windows: Có thể cài đặt máy ảo Linux và chia sẻ thư mục với máy chủ
 - Hoặc có thể tiếp tục lập trình C trên Windows
 - macOS: Có nhiều điểm tương đồng với Linux

Soạn thảo mã nguồn C

- Sử dụng trình soạn thảo mã nguồn hiểu cú pháp C và có nhiều phiên bản cho các HĐH khác nhau
 - Ví dụ: Sublime Text, Visual Studio Code, v.v..
- Đặt tên tệp với phần mở rộng .c



```
#include <stdio.h>
int main() {
    printf("Xin chào!\n");
    return 0;
}

-- INSERT --
```

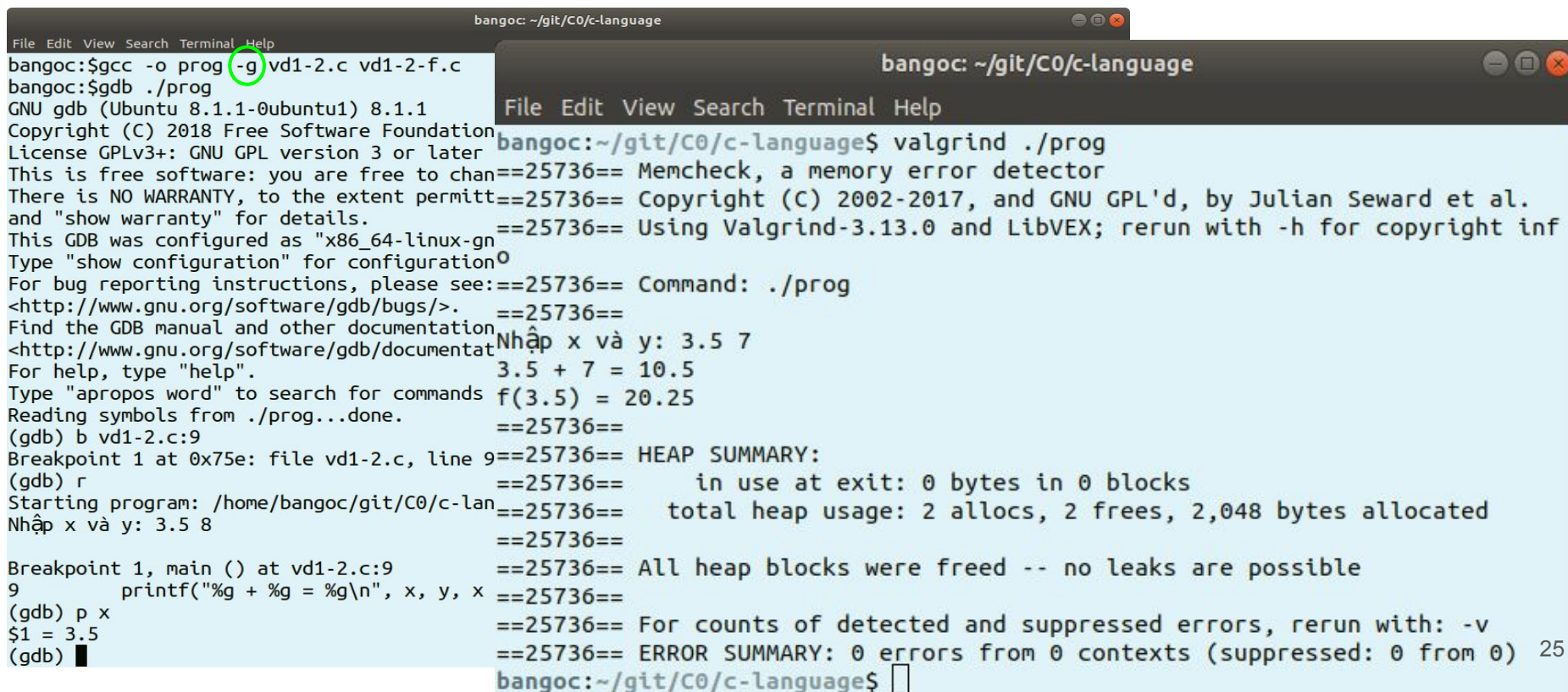
```
1 #include <stdio.h>
2 int main() {
3     printf("Xin chào!\n");
4     return 0;
5 }
6
```

Biên dịch mã nguồn C

- Sử dụng trình biên dịch hỗ trợ tốt các quy chuẩn ngôn ngữ C và có nhiều phiên bản cho các HĐH khác nhau
 - Ví dụ: GCC, Clang
 - Trình biên dịch được sử dụng và hỗ trợ chính thức trong quá trình học là GCC.
- Một số phiên bản phổ biến của GCC:
 - Windows: MinGW (được phân phối qua TDM GCC).
 - Linux: Thường có trong kho phần mềm tiêu chuẩn
 - `sudo apt-get install gcc`
 - macOS: Thường có trong kho phần mềm tiêu chuẩn
 - `brew install gcc`

Công cụ gỡ rối

- Các công cụ hỗ trợ người lập trình tìm lỗi trong chương trình: Hỗ trợ kiểm tra giá trị của biến, tìm lỗi sử dụng bộ nhớ động v.v.
 - Được sử dụng kết hợp với trình biên dịch
 - gcc: gdb, valgrind



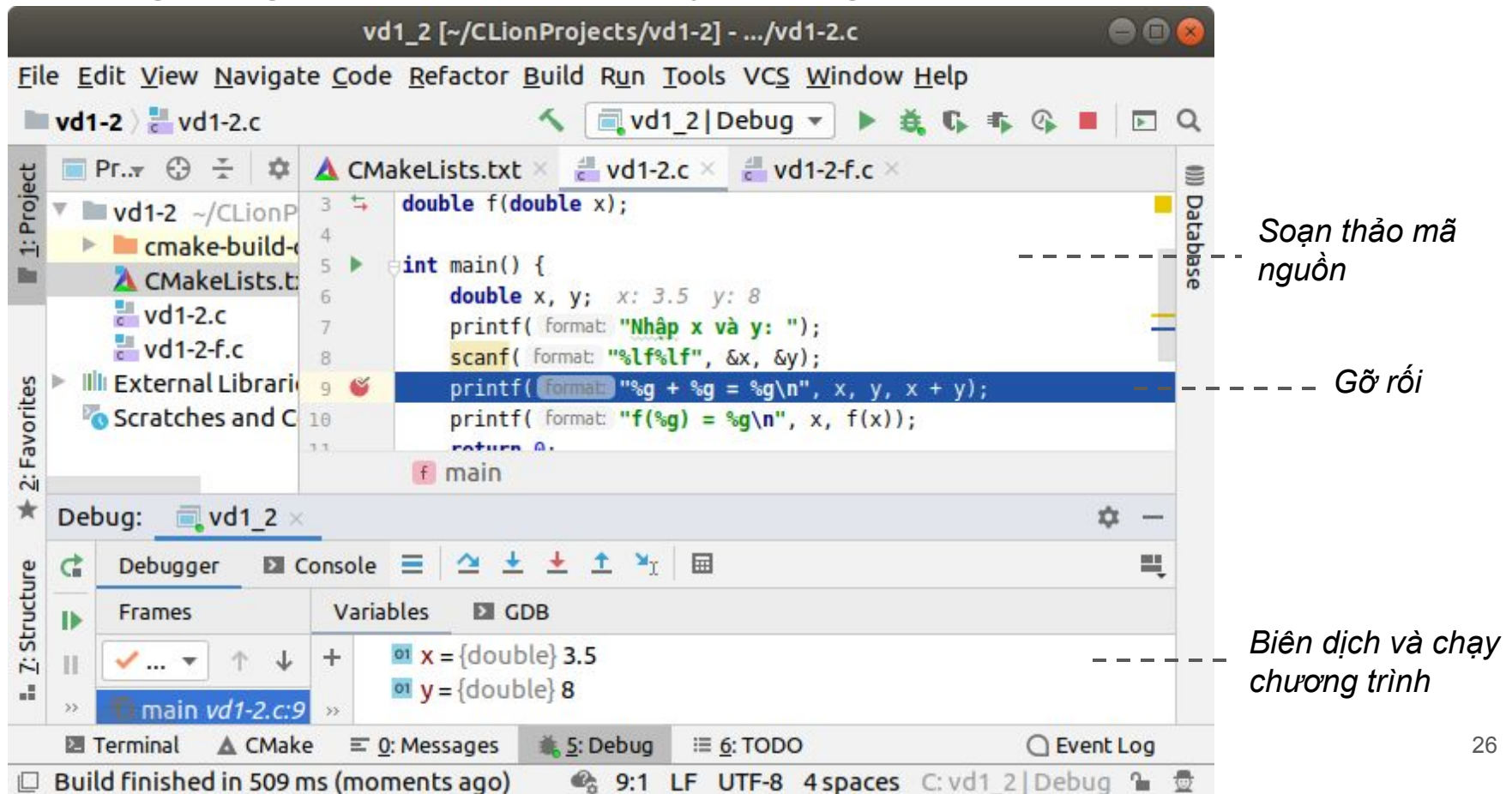
The image shows two terminal windows. The left window is titled 'bangoc: ~/git/C0/c-language' and shows the compilation of a program 'vd1-2.c' into 'prog' using 'gcc -g'. It then shows the execution of 'prog' using 'gdb'. The user sets a breakpoint at line 9 of 'vd1-2.c' and runs the program. The output shows the program calculating '3.5 + 7 = 10.5' and 'f(3.5) = 20.25'. The right window is also titled 'bangoc: ~/git/C0/c-language' and shows the execution of 'prog' using 'valgrind'. The output shows that the program ran successfully with no memory errors detected.

```
bangoc:~/git/C0/c-language$ gcc -o prog -g vd1-2.c vd1-2-f.c
bangoc:~/git/C0/c-language$ gdb ./prog
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
This is free software: you are free to change it under the terms of the GNU GPL.
There is NO WARRANTY, to the extent permitted by law.
Type "show configuration" for configuration details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands.
Reading symbols from ./prog...done.
(gdb) b vd1-2.c:9
Breakpoint 1 at 0x75e: file vd1-2.c, line 9
(gdb) r
Starting program: /home/bangoc/git/C0/c-language/./prog
Nhập x và y: 3.5 8
3.5 + 7 = 10.5
f(3.5) = 20.25
Breakpoint 1, main () at vd1-2.c:9
9      printf("%g + %g = %g\n", x, y, x + y);
(gdb) p x
$1 = 3.5
(gdb)

bangoc:~/git/C0/c-language$ valgrind ./prog
==25736== Memcheck, a memory error detector
==25736== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==25736== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==25736== Command: ./prog
==25736==
Nhập x và y: 3.5 7
3.5 + 7 = 10.5
f(3.5) = 20.25
==25736==
==25736== HEAP SUMMARY:
==25736==   in use at exit: 0 bytes in 0 blocks
==25736==   total heap usage: 2 allocs, 2 frees, 2,048 bytes allocated
==25736==
==25736== All heap blocks were freed -- no leaks are possible
==25736==
==25736== For counts of detected and suppressed errors, rerun with: -v
==25736== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
bangoc:~/git/C0/c-language$
```

Môi trường phát triển tích hợp

- Sử dụng môi trường phát triển tích hợp (Integrated Development Environment - **IDE**) có nhiều phiên bản cho các HĐH khác nhau.
 - Ví dụ Clion (miễn phí cho mục đích học tập)
 - Cung cấp đồng thời các tính năng để viết chương trình: Soạn thảo mã nguồn, gỡ rối, biên dịch và chạy chương trình.



Nội dung

1. Khái quát về Ngôn ngữ lập trình
2. Sơ lược kiến trúc von Neumann
3. Lịch sử phát triển NNLT C
4. Môi trường lập trình
5. Tiến trình biên dịch với gcc

Ví dụ 1.1. Biên dịch 1 tệp mã nguồn

```
vd1-1.c x
1 #include <stdio.h>
2 int main() {
3     printf("Chào mừng bạn đã đến với ");
4     printf("Ngôn ngữ lập trình C\n");
5     return 0;
6 }
7
```

*Chương trình này in ra
một câu chào mừng.*

*Trình biên dịch trong
dự án GCC (GNU)*



Biên dịch

*Thực hiện/chạy chương
trình (Linux, Unix)*

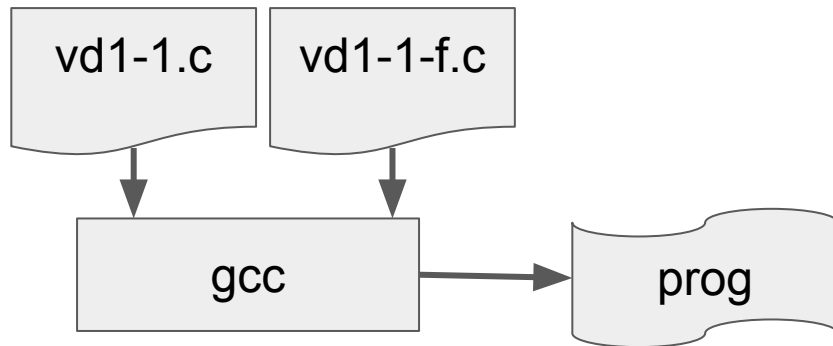
```
bangoc:$gcc -o prog vd1-1.c
bangoc:$./prog
Chào mừng bạn đã đến với Ngôn ngữ lập trình C
bangoc:$
```

Ví dụ 1.2. Biên dịch nhiều tệp

```
vd1-2.c
1 #include <stdio.h>
2
3 double f(double x);
4
5 int main() {
6     double x, y;
7     printf("Nhập x và y: ");
8     scanf("%lf%lf", &x, &y);
9     printf("%g + %g = %g\n", x, y, x + y);
10    printf("f(%g) = %g\n", x, f(x));
11    return 0;
12 }
```

```
vd1-2-f.c
1 double f(double x) {
2     return x * x + 2 * x + 1;
3 }
4
```

Chương trình này được tạo thành từ 2 tệp. Khi được thực hiện, chương trình hỏi người dùng nhập vào 2 số thực x và y, sau đó in ra tổng x + y và giá trị hàm $f(x) = x^2 + 2 * x + 1$

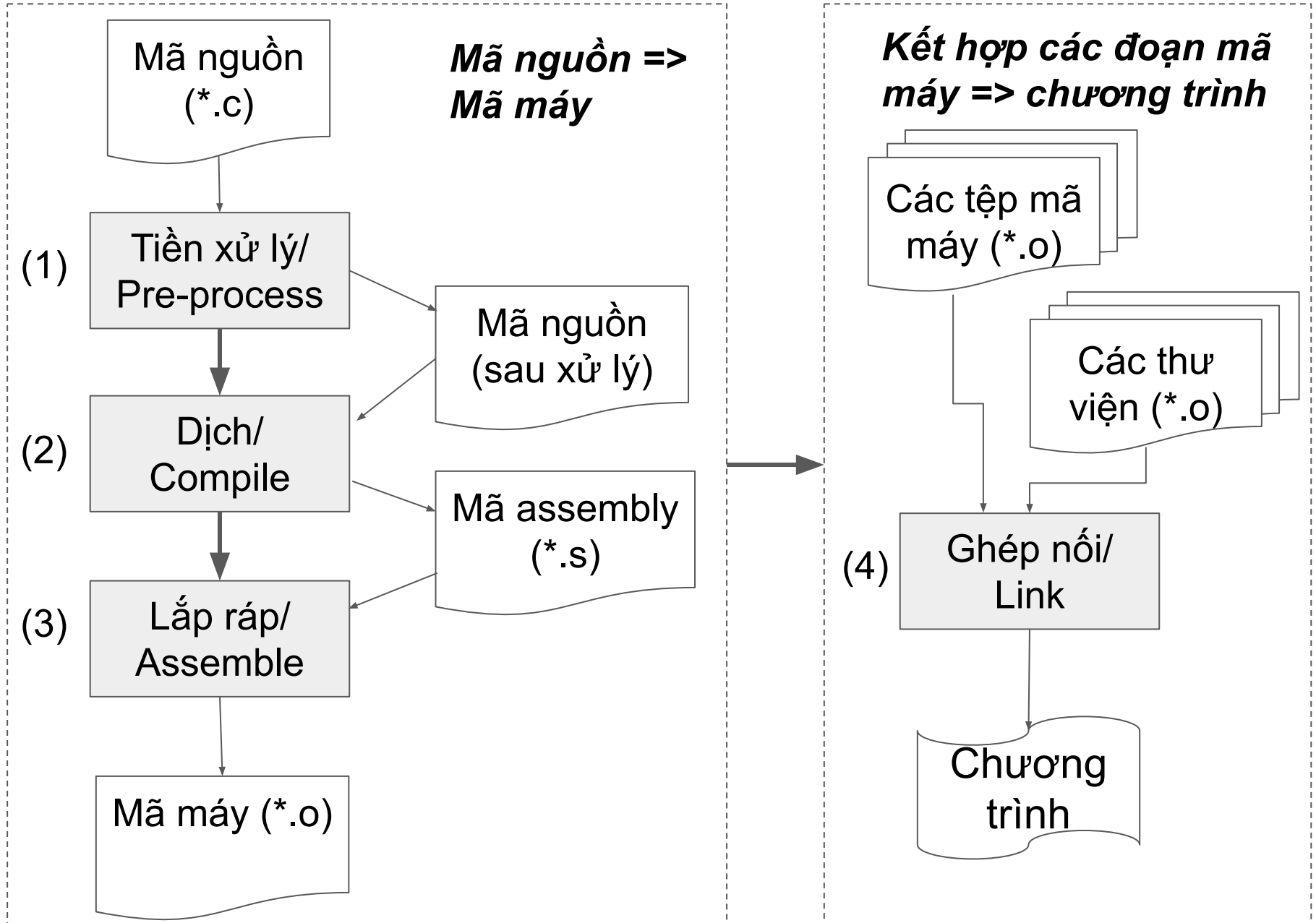


```
bangoc:$gcc -o prog vd1-2.c vd1-2-f.c
bangoc:$./prog
Nhập x và y: 3 5.5
3 + 5.5 = 8.5
f(3) = 16
bangoc:$
```

Các câu lệnh

`gcc -o prog vd1-1.c gcc -o prog vd1-2.c vd1-2-f.c`
làm những việc gì?

Tiến trình biên dịch với GCC



Ví dụ 1.3. Tiền xử lý

```
vd1-3.c      x      +  vd1-3.h      x
1  #include "vd1-3.h"
2
3  #define N 100
4
5  int a[N];
6
7  #ifdef PRINT
8  for (int i = 0; i < N; ++i) {
9      printf("a[%d] = %d\n", i, a[i]);
10 }
11 #endif
```

Các lệnh tiền xử lý thường được bắt đầu với #:

#include - Chèn nội dung tệp vào mã nguồn

#define - Định nghĩa Macro

#ifdef ... #endif - Cấu trúc rẽ nhánh

...

Các tham số gcc:

-E Chỉ tiền xử lý mã nguồn, không dịch, lắp ráp hay ghép nối.

-o <tệp> Lưu kết quả biên dịch vào tệp

-D Định nghĩa Macro

```
1  static inline mmax(int x, int y) {
2      return x > y? x: y;
3  }
4
```

```
bangoc: ~/git/C0/c-language
File Edit View Search Terminal Help
bangoc:$gcc -o vd1-3.i -E vd1-3.c
bangoc:$cat vd1-3.i
...
static inline mmax(int x, int y) {
    return x > y? x: y;
}
...
int a[100];
bangoc:$gcc -o vd1-3.i -E -DPRINT vd1-3.c
bangoc:$cat vd1-3.i
...
static inline mmax(int x, int y) {
    return x > y? x: y;
}
...
int a[100];
for (int i = 0; i < 100; ++i) {
    printf("a[%d] = %d\n", i, a[i]);
}
bangoc:$
```

Ví dụ 1.4.a. Dịch thành mã Assembly

```
1 #include <stdio.h>
2
3 #define SECRET 1984
4
5 int main() {
6     int x;
7     printf("Đoán số: ");
8     scanf("%d", &x);
9     if (x == SECRET) {
10         printf("Đúng\n");
11     } else {
12         printf("Sai\n");
13     }
14     return 0;
15 }
```

Bạn thử đọc và đoán xem chương trình này làm gì?

gcc -o vd1-4.s -S vd1-4.c Hoặc gcc -S vd1-4.c

Các tham số biên dịch:

-S Tiền xử lý và dịch thành mã Assembly, không lắp ráp hay ghép nối.

-o <tệp> Lưu kết quả biên dịch vào tệp

```
vd1-4.s
1 .file "vd1-4.c"
2 .text
3 .section .rodata
4 .LC0:
5     .string "\304\220o\30
6 .LC1:
7     .string "%d"
8 .LC2:
9     .string "\304\220\303
10 .text
11 .globl main
12 .type main, @function
13 main:
14 .LFB0:
15     .cfi_startproc
16     pushq %rbp
17     .cfi_def_cfa_offset 1
18     .cfi_offset 6, -16
19     movq %rsp, %rbp
20     .cfi_def_cfa_register
21     subq $16, %rsp
22     movq %fs:40, %rax
23     movq %rax, -8(%rbp)
24     xorl %eax, %eax
25     leaq .LC0(%rip), %rd
26     movl $0, %eax
27     call printf@PLT
28     leaq -12(%rbp), %rax
29     movq %rax, %rsi
30     leaq .LC1(%rip), %rd
31     movl $0, %eax
32     call __isoc99_scanf@
33     movl -12(%rbp), %eax
34     cmpl $1984, %eax
35     jne .L2
36     leaq .LC2(%rip), %rd
37     call puts@PLT
38 .L2:
39     movl $0, %eax
40     movq -8(%rbp), %rdx
41     xorq %fs:40, %rdx
42     je .L4
43     call __stack_chk_fai
44 .L4:
45     leave
46     .cfi_def_cfa 7, 8
47     ret
48     .cfi_endproc
49 .LFE0:
50     .size main, .-main
51     .ident "GCC: (Ubuntu
52     .section .note.GNU-s
53
```


Ví dụ 1.4.b. Dịch thành mã máy (lắp ráp)

```
1 #include <stdio.h>
2
3 #define SECRET 1984
4
5 int main() {
6     int x;
7     printf("Đoán số: ");
8     scanf("%d", &x);
9     if (x == SECRET) {
10         printf("Đúng\n");
11     } else {
12         printf("Sai\n");
13     }
14     return 0;
15 }
```

[illegible]

`gcc -o vd1-4.o -c vd1-4.c` Hoặc

```
gcc -c vd1-4.c
```

Các tham số biên dịch:

- c Tiền xử lý, dịch thành mã Assembly và lắp ráp thành mã máy, không ghép nối thành chương trình.

-o <tệp> Lưu kết quả biên dịch vào tệp

Tệp vd1-4.o chứa mã máy, chúng ta không đọc được như tệp văn bản.

Sử dụng objdump để xem mã máy:

```
objdump --disassemble --reloc vd1-4.o
```

```

bangoc: ~/git/C0/c-language
File Edit View Search Terminal Help
0000000000000000 <main>:
0: 55                                push    %rbp
1: 48 89 e5                          mov     %rsp,%rbp
4: 48 83 ec 10                        sub     $0x10,%rsp
8: 64 48 8b 04 25 28 00              mov     %fs:0x28,%rax
f: 00 00
11: 48 89 45 f8                       mov     %rax,-0x8(%rbp)
15: 31 c0                             xor     %eax,%eax
17: 48 8d 3d 00 00 00 00              lea     0x0(%rip),%rdi    # 1e <main+0x1e>
                                1a: R_X86_64_PC32      .rodata-0x4
1e: b8 00 00 00 00                    mov     $0x0,%eax
23: e8 00 00 00 00                    callq   28 <main+0x28>
                                24: R_X86_64_PLT32      printf-0x4
28: 48 8d 45 f4                       lea     -0xc(%rbp),%rax
2c: 48 89 c6                          mov     %rax,%rsi
2f: 48 8d 3d 00 00 00 00              lea     0x0(%rip),%rdi    # 36 <main+0x36>
                                32: R_X86_64_PC32      .rodata+0xa
36: b8 00 00 00 00                    mov     $0x0,%eax
3b: e8 00 00 00 00                    callq   40 <main+0x40>

```

Ví dụ 1.4.c. Ghép nối

```
vd1-4.c
1 #include <stdio.h>
2
3 #define SECRET 1984
4
5 int main() {
6     int x;
7     printf("Đoán số: ");
8     scanf("%d", &x);
9     if (x == SECRET) {
10         printf("Đúng\n");
11     } else {
12         printf("Sai\n");
13     }
14     return 0;
15 }
```

Chương trình này hỏi người dùng đoán 1 số. Nếu người dùng nhập số 1984 thì được coi là đúng. Nếu ngược lại thì được coi là sai.

gcc -o prog vd1-4.o Hoặc

gcc -o prog vd1-4.c

Các tham số biên dịch:

-o <tệp> Lưu kết quả biên dịch vào tệp **prog** là tên chương trình thực thi thu được.

Sử dụng **objdump** để xem mã máy:

objdump --disassemble --reloc prog

```
bangoc: ~/git/C0/c-language
File Edit View Search Terminal Help
bangoc:$gcc -o prog vd1-4.o
bangoc:$./prog
Đoán số: 1984
Đúng
bangoc:$objdump --disassemble --reloc prog

prog:      file format elf64-x86-64

Disassembly of section .init:

000000000000005e0 <_init>:
5e0:  48 83 ec 08          sub    $0x8,%rsp
5e4:  48 8b 05 fd 09 20 00  mov    0x2009fd(%rip),%rax
      # 200fe8 <__gmon_start__>
5eb:  48 85 c0             test   %rax,%rax
5ee:  74 02               je     5f2 <_init+0x12>
5f0:  ff d0               callq  *%rax
5f2:  48 83 c4 08          add    $0x8,%rsp
5f6:  c3                 retq

Disassembly of section .plt:

00000000000000600 <.plt>:
600:  ff 35 a2 09 20 00    pushq 0x2009a2(%rip)
      # 200fa8 <_GLOBAL_OFFSET_TABLE_+0x8>
606:  ff 25 a4 09 20 00    jmpq   *0x2009a4(%rip)
      # 200fb0 <_GLOBAL_OFFSET_TABLE_+0x10>
60c:  0f 1f 40 00          nopl   0x0(%rax)
```

Triển khai các thư viện tiêu chuẩn, hàm printf

```
00000000000000630 <printf@plt>:
630:  ff 25 92 09 20 00    jmpq   *0x200992(%rip)      # 200fc8 <printf@GLIBC_2.2.5>
636:  68 02 00 00 00      pushq  $0x2
63b:  e9 c0 ff ff ff      jmpq   600 <.plt>
```

