

Kỹ nghệ phần mềm

Software Engineering

Đại học Kinh doanh và Công nghệ Hà Nội
Khoa CNTT

GV: Đào Thị Phụng

Email: phuongdt102@gmail.com

Page fb: facebook.com/it.hubt

Phone: 0946.866.817

Bài 8: Lập trình



Nội dung

- Ngôn ngữ lập trình
- Phong cách lập trình
- Lập trình tránh lỗi
- Lập trình hướng hiệu quả

TÀI LIỆU THAM KHẢO



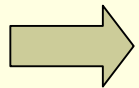
1. Nguyễn Văn Vy, Nguyễn Việt Hà. *Giáo trình kỹ nghệ phần mềm*. Nhà xuất bản Đại học Quốc gia Hà nội, 2008
2. Grady Booch, James Rumbaugh, Ivar Jacobson. *The Unified Modeling language User Guid*. Addison-Wesley, 1998.
3. M. Ould. *Managing Software Quality and Business Risk*, John Wiley and Sons, 1999.
4. Roger S.Pressman, *Software Engineering, a Practitioner's Approach*. Fifth Edition, McGraw Hill, 2001.
5. Ian Sommerville, *Software Engineering*. Sixth Edition, Addison-Wasley, 2001.
6. Nguyễn Văn Vy. *Phân tích thiết kế hệ thống thông tin hiện đại. Hướng cấu trúc và hướng đối tượng*, NXB Thống kê, 2002, Hà Nội.

Khái niệm lập trình hiệu quả



■ Sản phẩm phần mềm tốt khi

- ◆ phân tích tốt
- ◆ thiết kế tốt
- ◆ lập trình tốt
- ◆ kiểm thử chặt chẽ



■ kỹ thuật lập trình tốt

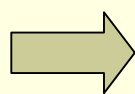
- ◆ chuyên nghiệp (tuân theo các chuẩn)
- ◆ ổn định
- ◆ hiệu quả

Khái niệm lập trình hiệu quả



Lập trình hiệu quả hơn, sản phẩm rẻ tiền hơn

- Tốc độ phát triển cao hơn
 - ◆ năng lực biểu diễn cao hơn
 - ◆ khả năng sử dụng lại cao hơn
- Dễ bảo trì hơn
 - ◆ dễ hiểu, dễ sửa đổi, thích nghi
- Chất lượng cao hơn
 - ◆ sử dụng các cấu trúc an toàn hơn



chương trình cần dễ hiểu

Tốc độ viết mã nguồn



- Tốc độ phát triển cao \neq làm ngắn chương trình nguồn
 - ◆ Tốc độ không tỷ lệ thuận với số dòng lệnh
 - ◆ Câu lệnh phức tạp làm giảm độ dễ hiểu

- Ngôn ngữ mức cao (4GL)
 - ◆ năng lực biểu diễn cao
 - ◆ tốc độ phát triển nhanh

Tiến hóa của kỹ thuật lập trình



- Lập trình tuần tự (**tuyến tính**)
- Lập trình có cấu trúc (**thủ tục**)
- Lập trình hướng chức năng
- Lập trình hướng đối tượng
- Kỹ thuật thế hệ thứ 4

Lập trình tuần tự

- không có/thiếu các lệnh có cấu trúc
(for, while, do while)
- lạm dụng các lệnh GOTO
- thiếu khả năng khai báo biến cục bộ

- ➔
- ◆ độ ghép nối cao
 - ◆ chương trình khó hiểu, khó sửa, dễ sinh lỗi

Ngôn ngữ dùng: thể hệ 1, 2: assembly, basic,...

Lập trình có cấu trúc

- sử dụng các lệnh có cấu trúc
(for, while, do while)
- hạn chế/cấm dùng GOTO
- sử dụng chương trình con, biến cục bộ

➡ dễ hiểu hơn, an toàn hơn

Ngôn ngữ dùng: thế hệ 2, 3: Fortran, Pascal, C, □

Lập trình hướng chức năng

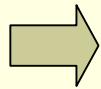


- Dựa trên nguyên tắc ghép nối dữ liệu
 - ◆ trao đổi dữ liệu bằng tham số và giá trị trả lại
 - ◆ loại bỏ hoàn toàn dữ liệu dùng chung
- Loại bỏ các hiệu ứng phụ khi sửa đổi các modul chương trình; nâng cao tính tái sử dụng
- Ví dụ: Lisp

Lập trình hướng đối tượng



- Bao gói & che dấu thông tin
- thao tác với dữ liệu qua các giao diện xác định
- kế thừa



- ◆ cục bộ hơn
- ◆ dễ tái sử dụng hơn
- ◆ thuận tiện cho các ứng dụng lớn


Ngôn ngữ hướng đối tượng: C++, Java, C#

Lập trình logic



- Tách tri thức về bài toán khỏi kỹ thuật lập trình
- Mô tả tri thức
 - ◆ các qui tắc
 - ◆ các sự thực
 - ◆ mục tiêu
- Hệ thống tự chứng minh
 - ◆ tìm đường đi đến mục tiêu
- Ví dụ: Prolog

Ví dụ. Prolog: 8 hậu



```
member(X,[X|L]).
member(X,[Y|L]):-member(X,L).
solution([]).
solution([X/Y|Other]):-
    solution(Other),
    member(Y, [1,2,3,4,5,6,7,8]),
    noattack(X/Y, Other).
noattack(_,[]).
noattack(X/Y, [X1/Y1 | Other]):-
    Y =\= Y1, Y1-Y =\= X1-X, Y1-Y =\= X-X1,
    noattack(X/Y , Other).
```

Lựa chọn ngôn ngữ



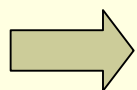
- Đặc trưng của ngôn ngữ
 - ◆ năng lực (kiểu biến, các cấu trúc)
 - ◆ tính khả chuyển
 - ◆ mức độ hỗ trợ của các công cụ
- Miền ứng dụng của ngôn ngữ
 - ◆ lập trình hệ thống
 - ◆ nghiệp vụ, kinh doanh
 - ◆ khoa học kỹ thuật
 - ◆ trí tuệ nhân tạo
- Năng lực, kinh nghiệm của nhóm phát triển
- Yêu cầu của khách hàng

Năng lực của ngôn ngữ



■ Ngôn ngữ bậc cao:

- ◆ có cấu trúc, câu lệnh phong phú
- ◆ hỗ trợ nhiều kiểu dữ liệu
- ◆ hỗ trợ con trỏ, đệ qui
- ◆ hỗ trợ hướng đối tượng
- ◆ thư viện phong phú



nên dùng ngôn ngữ bậc cao (hơn)

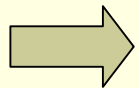
Tính khả chuyển

- Yếu tố quan trọng của ngôn ngữ, cần khi

- thay đổi phần cứng
- thay đổi OS

- ◆ Java khả chuyển

- ◆ Các ngôn ngữ thông dịch (script) khả chuyển



- sử dụng các tính năng chuẩn của ngôn ngữ
- sử dụng script khi có thể

Có công cụ hiệu quả



- Trình biên dịch hiệu quả
 - ◆ biên dịch tốc độ cao
 - ◆ khả năng tối ưu cao
 - ◆ khai thác các tập lệnh, kiến trúc phần cứng mới
- Các công cụ trợ giúp hiệu quả
 - ◆ editor, debugger, linker, make...
 - ◆ IDE (Integrated Develop Environment)
 - ◆ môi trường Unix thường không dùng IDE

Miền ứng dụng và ngôn ngữ



- Phần mềm hệ thống:
 - ◆ hiệu quả, vận năng, dễ mở rộng
 - ◆ C, C++
- Hệ thời gian thực: C, C++, Ada, Assembly
- Phần mềm nhúng: C++, Java
- Phần mềm khoa học kỹ thuật:
 - ◆ tính toán chính xác, thư viện toán học mạnh, dễ dàng song song hóa
 - ◆ Fortran vẫn phổ biến

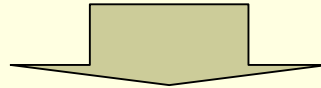
Miền ứng dụng và ngôn ngữ (t)



- Phần mềm nghiệp vụ:
 - ◆ CSDL: Oracle, DB2, SQL Server, MySQL...
 - ◆ ngôn ngữ: FoxPro, COBOL, VB, VC++
- Trí tuệ nhân tạo:
 - ◆ Lisp, Prolog, OPS5,...
- Lập trình Web/CGI:
 - ◆ Perl, ASP, PHP, Java, Java script, Python...

Phong cách lập trình

- Bao gồm các yếu tố:
 - ◆ cách đặt tên hàm và biến
 - ◆ cách xây dựng câu lệnh, cấu trúc chương trình
 - ◆ cách viết chú thích



Hướng tới phong cách làm cho mã nguồn

- ◆ dễ hiểu, dễ sửa đổi
- ◆ an toàn (ít lỗi)

Người khác có thể hiểu được, bảo trì được

Tại sao cần dễ hiểu



Phần mềm luôn cần sửa đổi

- ◆ sửa lỗi
- ◆ nâng cấp

➡ kéo dài tuổi thọ, nâng cao hiệu quả kinh tế

Nếu không dễ hiểu

- ◆ bảo trì tốn thời gian, chi phí cao
- ◆ tác giả phải bảo trì suốt vòng đời của phần mềm
- ◆ bản thân tác giả cũng không hiểu

Chú thích



Mọi điều được Chú thích trong chương trình

- Mục đích sử dụng của các biến
- Chức năng của khối lệnh, câu lệnh
 - ◆ các lệnh điều khiển
 - ◆ các lệnh phức tạp
- Chú thích các mô đun
 - ◆ mục đích, chức năng của mô đun
 - ◆ tham số, giá trị trả lại (giao diện)
 - ◆ các mô đun thuộc cấp
 - ◆ cấu trúc, thuật toán
 - ◆ nhiệm vụ của các biến cục bộ
 - ◆ tác giả, người kiểm tra, thời gian

Đặt tên



Đặt tên biến, tên hàm có nghĩa, gợi nhớ

- Sử dụng các ký hiệu, từ tiếng Anh có nghĩa
- Làm cho dễ đọc
 - ◆ dùng `DateOfBirth` hoặc `date_of_birth`
 - ◆ không viết `dateofbirth`
- Tránh đặt tên quá dài
 - ◆ không đặt tên dài với các biến cục bộ
- Thống nhất cách dùng
 - ◆ `i` cho vòng lặp, `tmp` cho các giá trị tạm thời...

Câu lệnh



- Các câu lệnh phải mô tả cấu trúc
 - ◆ tụt lề, dễ đọc, dễ hiểu
- Làm đơn giản các lệnh
 - ◆ mỗi lệnh trên một dòng
 - ◆ triển khai các biểu thức phức tạp
 - ◆ hạn chế truyền tham số là kết quả của hàm, biểu thức

```
printf("%s", strcpy(des, src));
```
- Tránh các cấu trúc phức tạp:
 - ◆ các lệnh if lồng nhau
 - ◆ điều kiện phủ định `if not`

Hàm và biến cục bộ



- Chương trình cần được chia thành nhiều mô đun (hàm)
- Không viết hàm quá dài
 - ◆ không quá 2 trang màn hình
 - ◆ tạo ra các hàm thứ cấp để giảm độ dài từng hàm
- Không dùng quá nhiều biến cục bộ
 - ◆ không thể theo dõi đồng thời hoạt động của nhiều biến

(vd. không quá 7 biến cục bộ)

Xử lý lỗi



- Có thể phát hiện lỗi trong khi thực hiện
 - ◆ lỗi chia 0
 - ◆ lỗi input/output,
- Xử lý lỗi
 - ◆ nhất quán trong xử lý: *phân loại lỗi; thống nhất định dạng thông báo,*
 - ◆ phân biệt output và thông báo lỗi
 - ◆ các hàm thư viện nên tránh việc tự xử lý, tự đưa ra thông báo lỗi

Output và thông báo (lỗi)



- Output là dữ liệu, còn được dùng để làm input cho phần mềm khác
- Thông báo (lỗi) là các thông tin nhất thời, trạng thái hệ thống, lỗi và cách khắc phục
- Cần tách output và thông báo lỗi
- OS thường cung cấp 3 luồng dữ liệu chuẩn
 - ◆ `stdin (cin)`: input chuẩn (bàn phím)
 - ◆ `stdout (cout)`: output chuẩn (màn hình)
 - ◆ `stderr (cerr)`: luồng thông báo lỗi chuẩn (không định hướng lại được)

convert.c



```
...
void main()
{
    int count = 0;
    char buf[128];
    while (NULL != scanf("%s", buf)) {
        upcase(buf);
        printf("%s", buf);
        count += strlen(buf);
    }
    fprintf(stderr, "number of chars: %d\n", count);
}
```

```
#convert < normal.txt > uppercase.txt
```

```
number of chars: 678
```

Xử lý lỗi trong hàm thư viện

- Người viết và người sử dụng thư viện là khác nhau
- Người sử dụng thường muốn có cách xử lý riêng
- Hàm thư viện trả lại trạng thái lỗi, không tự xử lý
 - ◆ trả trạng thái bằng giá trị trả lại
 - ◆ trả trạng thái bằng tham số
 - ◆ trả lại bằng ném ngoại lệ (trong các OOL)

```
int lookup(int a[], int key, int& err_code)
{
    if (not found) err_code = 0;
    else err_code =1;
}
```

Ngoại lệ



- Là cách thức xử lý lỗi tiến tiến trong các ngôn ngữ hướng đối tượng
 - ◆ môđun xử lý ném ra một *ngoại lệ* (đối tượng chứa thông tin lỗi)
 - ◆ môđun điều khiển bắt ngoại lệ (nếu có)
- Tách phần xử lý lỗi khỏi phần cài đặt thuật toán thông thường, làm cho chương trình dễ đọc hơn
- Dễ dùng hơn, an toàn hơn

Ném ngoại lệ



```
double MyDivide(double num, double denom)
{
    if (denom == 0.0) {
        throw invalid_argument("The denom cannot be 0.");
    }
    else {
        return num / denom;
    }
}
```

Bắt ngoại lệ



```
try {  
    result = MyDivide(x, y);  
}  
catch (invalid_argument& e) {  
    cerr << e.what() << endl;  
    ...    // mã xử lý với ngoại lệ  
};
```


Giao diện của mô đun



- Thống nhất định dạng
 - ◆ thứ tự truyền tham số
 - ◆ strcpy(des, src)
- Kiểm tra tính hợp lệ của dữ liệu
 - ◆ chỉ thực hiện xử lý với dữ liệu hợp lệ
- Làm đơn giản giao diện (*giảm độ ghép nối*)
 - ◆ không truyền thừa tham số
 - ◆ tìm cách kết hợp các khoản mục liên quan

Phong cách lập trình tốt



- Tuân theo các chuẩn thông dụng
- Chuẩn được chấp nhận rộng rãi hơn dễ hiểu hơn
- Chú giải đầy đủ mỗi khi không tuân theo chuẩn

"Người khác có hiểu được không?"

Kỹ thuật lập trình tránh lỗi



Kỹ thuật lập trình tốt dựa trên các yếu tố

- lập trình có cấu trúc
 - ◆ dùng các lệnh có cấu trúc
 - ◆ môđun hóa
 - ◆ hạn chế dùng các cấu trúc nguy hiểm
- đóng gói/che dấu thông tin
 - ◆ xây dựng kiểu dữ liệu trừu tượng
 - ◆ hạn chế thao tác trực tiếp lên thuộc tính

Tránh các cấu trúc nguy hiểm



■ Số thực

- ◆ các phép toán được làm tròn, kết quả không
- ◆ chính xác tuyệt đối
- ◆ so sánh (=) hai số thực là không khả thi

■ Con trỏ

- ◆ khái niệm mức thấp
- ◆ có khả năng gây lỗi nghiêm trọng
- ◆ dễ nhầm

```
double r;  
int* n = &r;
```

Tránh các cấu trúc nguy hiểm



- Cấp phát bộ nhớ động
 - ◆ quên cấp phát
 - ◆ quên giải phóng
 - ◆ chỉ nên dùng với ngôn ngữ hướng đối tượng (C++)
- Độ qui
 - ◆ khó hiểu
 - ◆ dễ nhầm điều kiện dừng

Bộ nhớ động với C++


- Tự động hóa cấp phát bộ nhớ
- Tự động hóa giải phóng

```
class Stack {  
private:  
    int* buf;  
    int  pos;  
public:  
    Stack (int size) { buf = new int[size]; }  
    int push(int);  
    int pop();  
    ~Stack() { delete[] buf; }  
}
```

hàm khởi tạo đối tượng

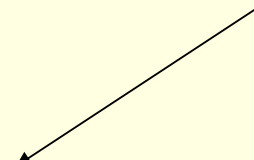
hàm hủy đối tượng

Bộ nhớ động với C++

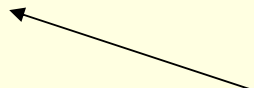


```
int use_stack()  
{  
    int m;  
    Stack s(10);  
    ...  
    s.push(m);  
    ...  
    return 0;  
}
```

đối tượng s được khởi tạo,
hàm khởi tạo được gọi tự động



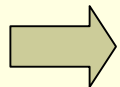
hết phạm vi sử dụng,
đối tượng s được hủy,
hàm hủy được gọi tự động



Định kiểu dữ liệu

- Nhiều lỗi lập trình do gán dữ liệu sai kiểu
- Các ngôn ngữ bậc cao cung cấp nhiều kiểu dữ liệu và cho phép *đặc tả miền dữ liệu*
- Tên kiểu có nghĩa làm cho chương trình dễ hiểu

```
typedef enum {Green, Yellow, Red} TColor;  
...  
TColor CurrentColor;  
CurrentColor = Red;  
CurrentColor = 4;
```



phát hiện các lỗi sai kiểu khi biên dịch

Lớp/kiểu dữ liệu trừu tượng

- Mức cao hơn của định kiểu dữ liệu là xây dựng lớp đối tượng
- Kiểm tra động về tính đúng đắn của dữ liệu

```
class PrimeNumber {  
private:  
    int value;  
public:  
    ...  
    int set(int val, int& err_code) {  
        // verify val  
        // if val is not a prime number  
        // set err_code = 1  
        ...  
    }  
};
```

Lớp/kiểu dữ liệu trừu tượng

...

```
PrimeNumber p;  
int n, err_code;  
err_code = 1;  
  
while (err_code) {  
    cout << "input a prime number: ";  
    cin >> n;  
    p.set(n, err_code);  
}
```

gán và kiểm tra tính hợp lệ tự động

Lập trình phòng thủ (*Defensive programming*)



- Nhiều lệnh có khả năng sinh lỗi
 - lệnh vào/ra
 - các phép toán
 - thao tác với bộ nhớ
 - truyền tham số sai kiểu
- Dự đoán khả năng xuất hiện lỗi
- Khắc phục lỗi
 - lưu trạng thái an toàn
 - quay lại trạng thái an toàn gần nhất

Lập trình phòng thủ (*Defensive programming*)



- Lệnh vào ra
 - ◆ dữ liệu không hợp lệ
 - ◆ tràn bộ đệm (kiểu ký tự)
 - ◆ lỗi thao tác file (sai tên, chưa được mở, □)
- Các phép toán
 - ◆ lỗi chia 0
 - ◆ tràn số
 - ◆ so sánh số thực (bằng nhau)

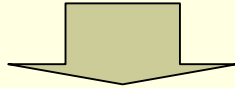
Lập trình phòng thủ (*Defensive programming*)




- Thao tác bộ nhớ
 - ◆ quên cấp phát, quên giải phóng bộ nhớ
 - ◆ thiếu bộ nhớ
 - ◆ sai địa chỉ, tràn bộ nhớ

Lập trình phòng thủ

```
FILE* fp;  
fp = fopen("data", "r");
```



```
FILE* fp;  
if (NULL == (fp = fopen("data", "r"))) {  
    fprintf(stderr, "can not open file...");  
    ...  
}
```



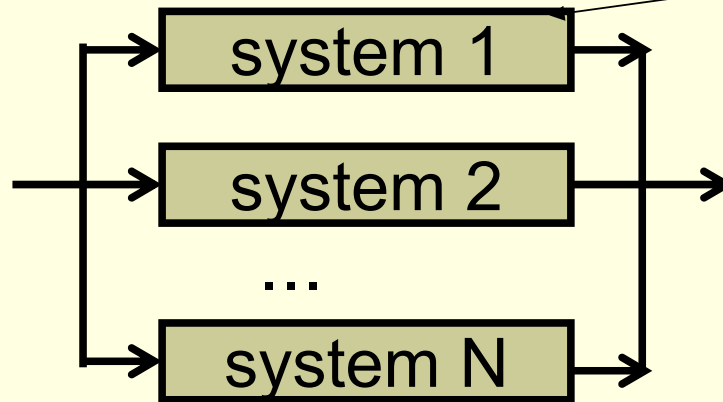
Lập trình thứ lỗi

Fault tolerance programming

- Không thể loại trừ hoàn toàn lỗi
- Cần có các hệ thống có độ tin cậy đặc biệt
- Dung thứ lỗi: chấp nhận sự xuất hiện lỗi lập trình
- Phát hiện, khắc phục lỗi
- Khởi nguyên từ thứ lỗi phần cứng
- Dựa trên nguyên tắc song song hóa chức năng

Nguyên lý thứ lỗi

- Song song hóa thiết bị



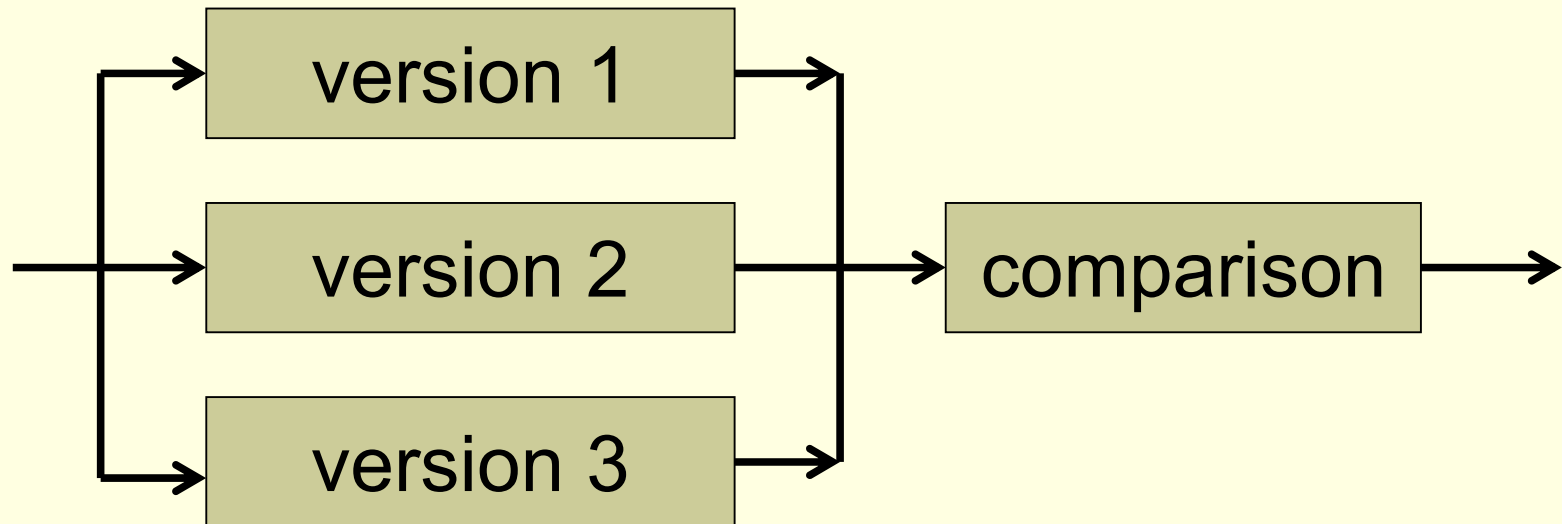
xác suất có lỗi $\alpha < 1$

xác suất cả hệ thống
ngừng hoạt động
 α^N

- Dùng thông tin *dư thừa* để khôi phục dữ liệu
 - ◆ Error Correction Code □ ECC
 - ◆ parity check, check sum

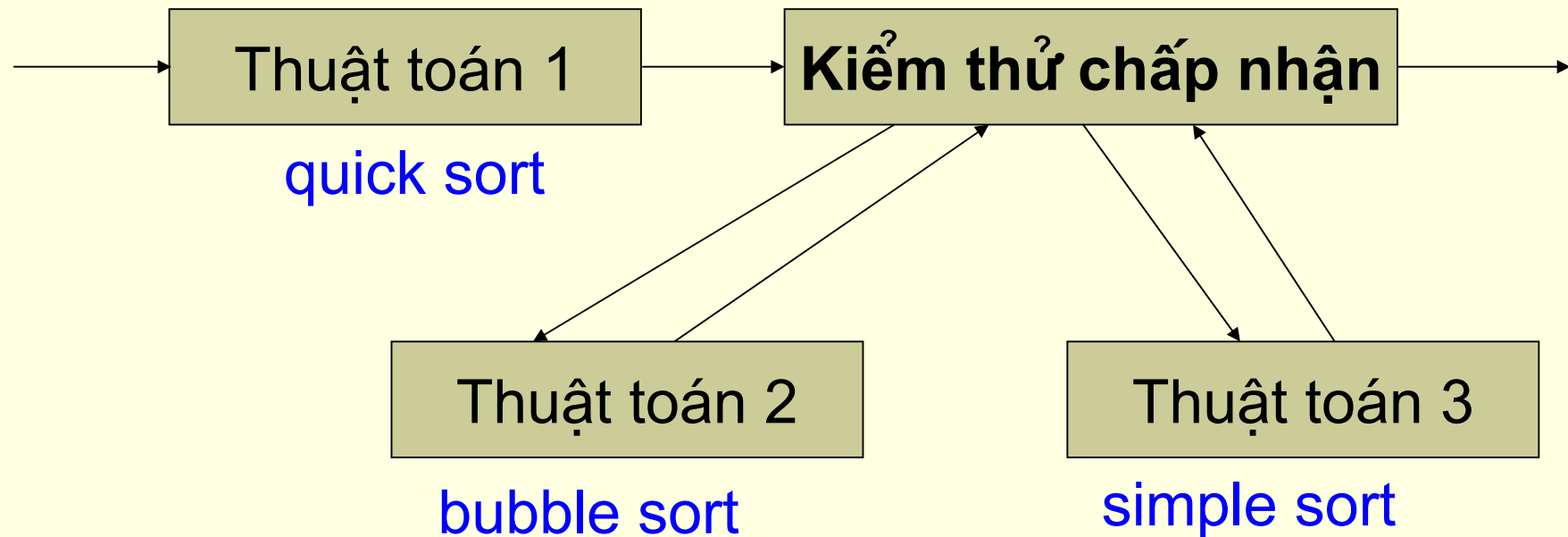
N - version programming

- sử dụng nhiều nhóm lập trình
- so sánh kết quả
- *cùng thuật toán thì có xu hướng mắc cùng lỗi*



Khối phục hồi (*Recovery block*)

- sử dụng các thuật toán khác nhau
- kiểm tra tính hợp lệ của kết quả



Thứ lỗi dữ liệu (*Data fault tolerance*)



- Phục hồi lùi
 - ◆ kiểm tra tính hợp lệ của dữ liệu mỗi khi biến đổi
 - ◆ chỉ chấp nhận các dữ liệu hợp lệ
- Phục hồi tiến
 - ◆ dùng dữ liệu *dư thừa*
 - ◆ kiểm tra và khôi phục dữ liệu

Hướng hiệu quả thực hiện



- Phần mềm ngày càng phức tạp, đa dạng
 - ◆ mô phỏng
 - ◆ ứng dụng thời gian thực
 - ◆ phần mềm nhúng
 - ◆ trò chơi
- Hiệu quả thực hiện luôn cần được xem xét
 - ◆ thuật toán hiệu quả
 - ◆ kỹ thuật lập trình hiệu quả
 - ◆ ngôn ngữ lập trình hiệu quả

Cải thiện tốc độ



- Tối ưu hóa chu trình
 - ◆ đưa phép toán bất biến ra ngoài chu trình
 - ◆ tính sẵn giá trị được sử dụng nhiều lần
- Hạn chế gọi hàm nhỏ với tần số cao
 - ◆ phí tổn cho gọi hàm nhỏ
 - ◆ dùng hàm inline, dùng macro
- Hạn chế truyền tham số trị là cấu trúc phức tạp
 - ◆ tham số được copy lên stack
 - ◆ truyền tham số biến (tham chiếu)

Phí tổn cho gọi hàm/thủ tục



- Copy tham số lên stack
- Copy giá trị trả lại từ stack
- Chuyển điều khiển chương trình
 - ◆ mất tính liên tục của dòng lệnh
 - ◆ mất tính cục bộ của cache

Macro và hàm inline



- ◆ không có kiểu
- ◆ không an toàn

`#define CUBE(x) (x*x*x)`

`CUBE(n+1)`

```
inline int cube(int x)
{
    return x*x*x;
}
```

Truyền tham số



- Tham số trị
 - ◆ copy giá trị (bộ nhớ) của tham số lên stack
 - ◆ không hiệu quả với biến kiểu cấu trúc lớn
 - ◆ không an toàn với các cấu trúc phức tạp
- Tham số biến/tham chiếu
 - ◆ copy địa chỉ của tham số lên stack
 - ◆ địa chỉ có độ lớn cố định (4 bytes)
 - ◆ nội dung có thể bị thay đổi (dùng từ khóa `const`)

Cải thiện tốc độ (tiếp)

- Tránh dùng mảng nhiều chiều
- Tránh dùng biến con trỏ
- Dùng các kiểu dữ liệu đơn giản

`double --> float`

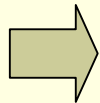
- Rút gọn các biểu thức
- Dùng các *phép toán nhanh*

`++i;`

`n<<2;`

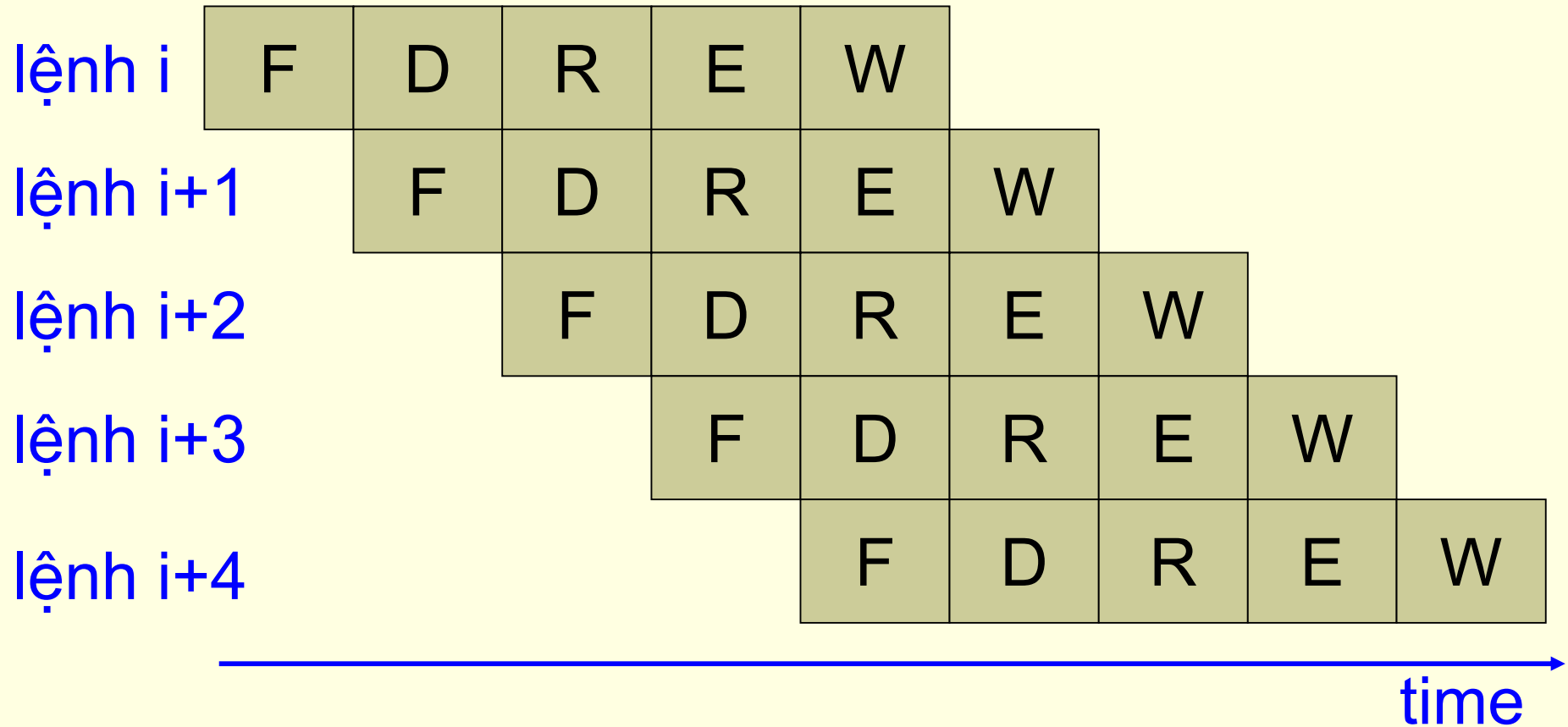
Cải thiện tốc độ (tránh lệnh rẽ nhánh)

- Tránh gọi hàm (chương trình con)
- Tránh vòng lặp
- Tránh điều kiện (if)



tận dụng hiệu quả pipeline

Pipeline



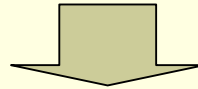
Hiệu quả vào/ra



- Tối thiểu các yêu cầu input/output
 - ◆ tốc độ thiết bị ngoại vi chậm
 - ◆ loại bỏ các inp/out không cần thiết
- ```
for (i=0; i<10000000; i++) {
 ...
 printf("i = %d\n", i);
}
```
- Input/Output theo khối
  - ◆ tối thiểu số lần gọi thư viện input/output
  - ◆ tận dụng hiệu quả bộ đệm input/output



```
Person a[N];
...
for (i=0; i<N; i++) {
 fread(&a[i], sizeof(Person), 1, fp);
 ...
}
```



```
fread(a, sizeof(Person), N, fp);
for (i=0; i<N; i++) {
 ...
}
```

# Phong cách lập trình với C/C++



- Phần mềm được cấu thành bởi nhiều tệp, các tệp thường được biên dịch riêng rẽ
  - ◆ .c: khai báo dữ liệu, hàm
  - ◆ .h: định nghĩa các kiểu dữ liệu dùng chung
- Không khai báo dữ liệu, định nghĩa hàm trong các tệp header (.h)

# Các bước xây dựng phần mềm trên C/C++



- Soạn thảo (**edit**)
- Tiền biên dịch (**pre-compile**)
- Biên dịch (**compile**)
- Kết nối (**link**)
- Thực hiện (**execute**)

# date.h



```
typedef struct {
 int year, mon, day;
} Date;
```

```
Date my_birthday;
```

```
void print_date(Date d)
{
 printf("year: %d, mon: %d, day: %d",
 d.year, d.mon, d.day);
}
```



# date.h

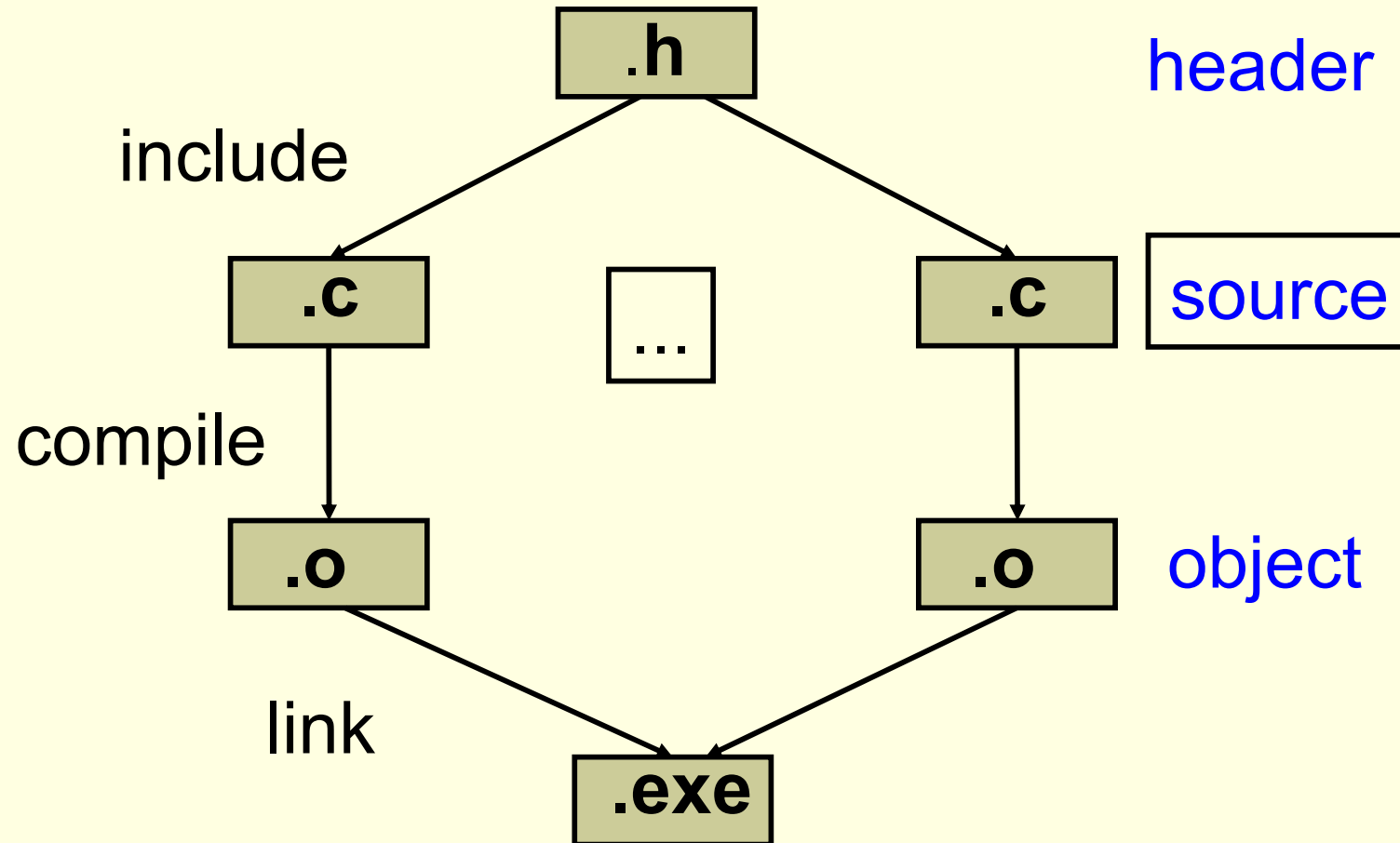
## date.h

```
typedef struct {
 int year, mon, day;
} Date;
void print_date(Date);
```

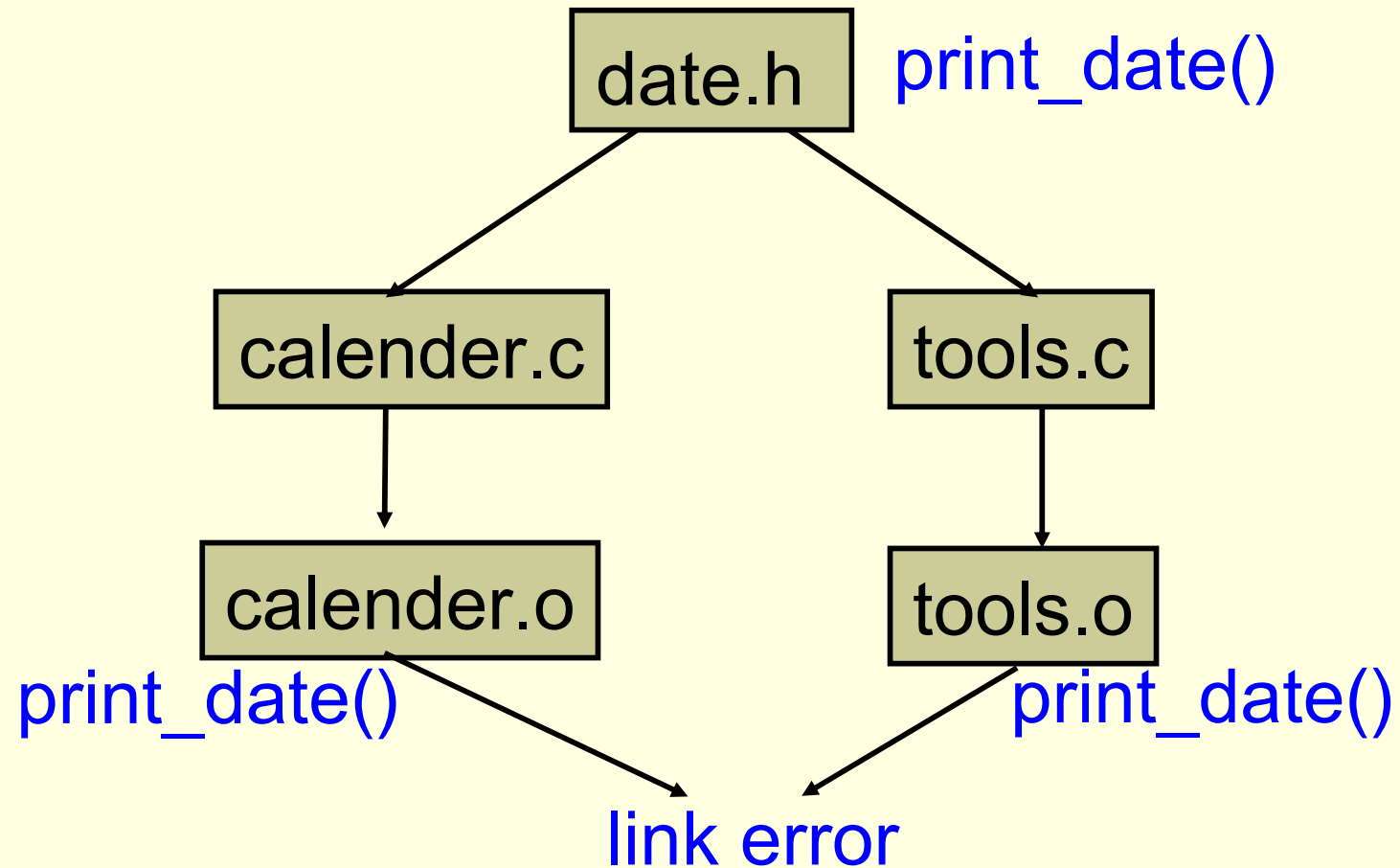
## date.c

```
#include "date.h"
void print_date(Date d)
{
 printf("year: %d, mon: %d, day: %d",
 d.year, d.mon, d.day);
}
```

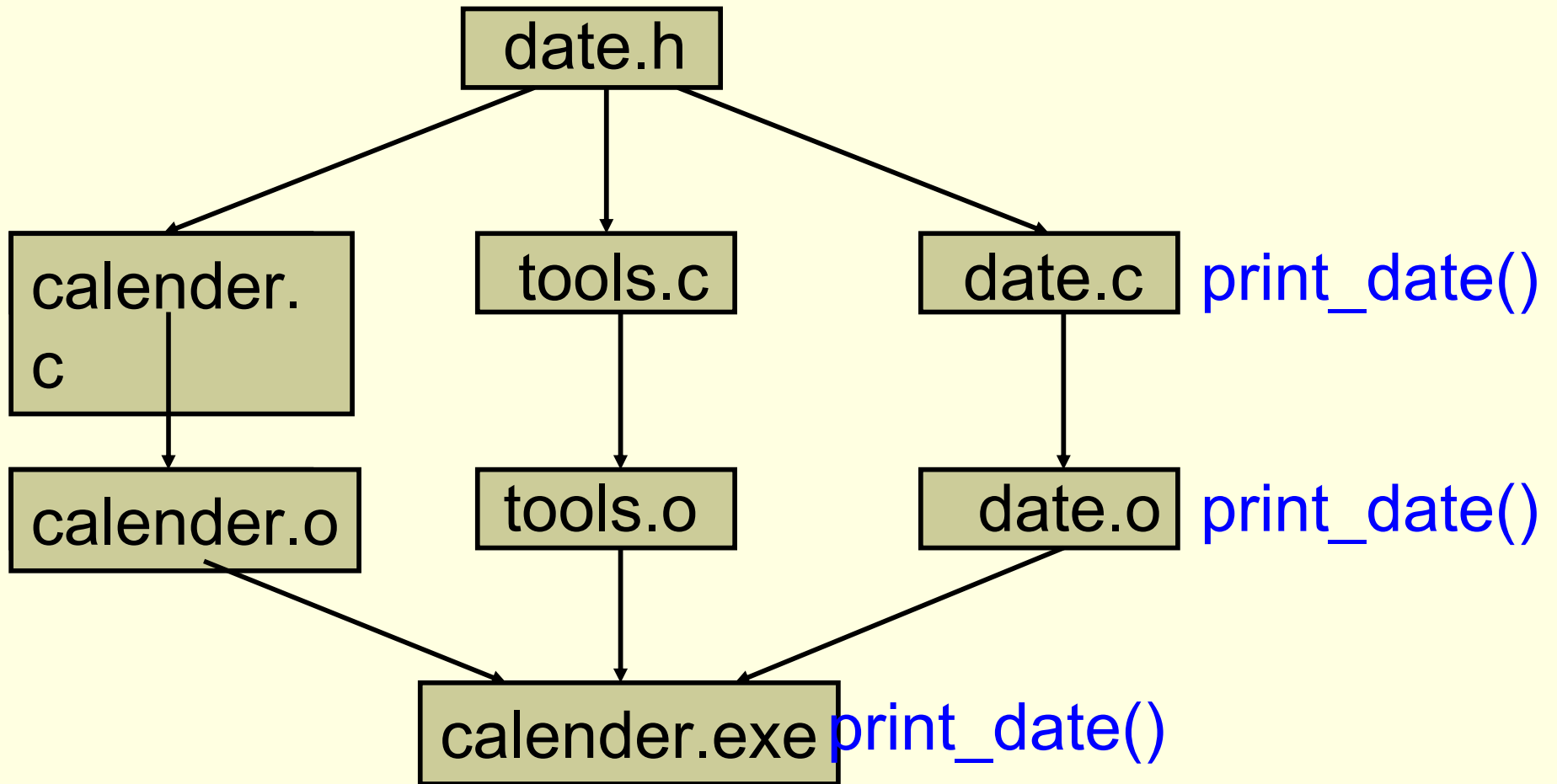
date.h



# date.h



# date.h



# Pre-compile



- Chỉ thị chương trình dịch (tiền biên dịch)
- Tạo ra các macro
- Tạo ra các phiên bản phần mềm khác nhau
- Quản lí các tệp header

`#include`

`#define`

`#ifdef`

`#ifndef`

`#endif`

# Pre-compile



```
#define Cube(x) (x*x*x)
...
void main()
{
 ...
 m = Cube(n);
 ...
}
```

# Pre-compile

Tạo các phiên bản phần mềm khác nhau

```
#define DEBUG
```

```
...
```

```
int foo()
```

```
{
```

```
#ifdef DEBUG
```

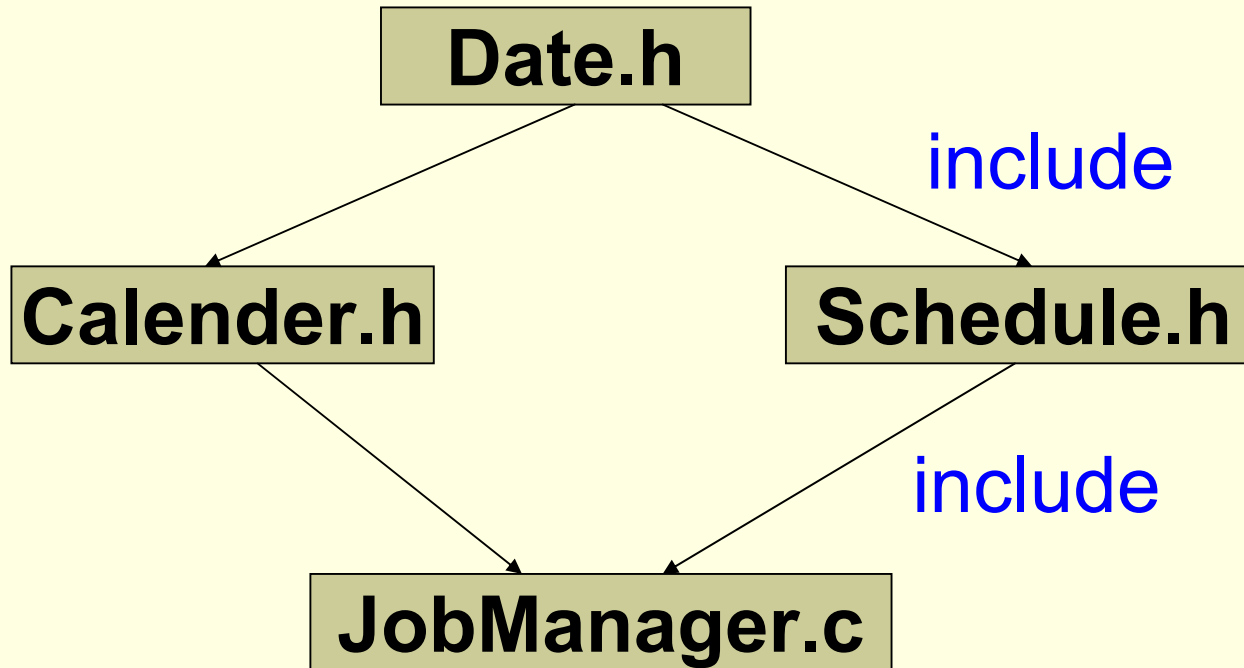
```
 fprintf(stderr, "foo() called");
```

```
#endif
```

```
...
```

```
}
```

# Header & Precompile





# Header & Precompile



date.h

```
#ifndef DATE_H
#define DATE_H

typedef struct {
 int year, mon, day;
} Date;

void print_date(Date);

#endif
```

# Header & Precompile



## calender.h

```
#ifndef CALENDER_H
#define CALENDER_H

#include "date.h"
...

#endif
```

## schedule.h

```
#ifndef SCHEDULE_H
#define SCHEDULE_H

#include "date.h"
...

#endif
```

# Header & Precompile



JobManager.c

```
#include "calender.h"
#include "schedule.h"

...
void main()
{
...
}
```

# Câu hỏi ôn tập



1. Kỹ thuật lập trình tốt thể hiện ở chỗ nào? Hệ quả của nó?
2. Nêu các kỹ thuật lập trình đã có? đặc trưng của mỗi loại là gì?
3. Tiêu chuẩn lựa chọn ngôn ngữ lập trình?
4. Thế nào là ngôn ngữ khả chuyển? Cho ví dụ?
5. Nêu các miền ứng dụng và ngôn ngữ thích hợp với nó?
6. Các yếu tố tạo ra phong cách lập trình là gì? Nó hướng tới phần mềm có đặc trưng gì?

# Câu hỏi ôn tập



7. Giải thích cách làm: *chú thích?*, *đặt tên?*, *viết câu lệnh?*, *đặt hàm và biến cục bộ?*, *xử lý lỗi và thông báo?* *Xử lý lỗi trong thư viện?* *Xử lý ngoại lệ?*
8. Tiêu chuẩn cho phong cách lập trình tốt?

# Câu hỏi và thảo luận

