# Bài 1: Modular Programming

## Mô-đun Lập Trình

Trong chương thứ hai của bài hướng dẫn, bạn sẽ khám phá những khái niệm lập trình ngôn ngữ C cao hơn.

Cho đến lúc này, các bạn cũng chỉ làm việc trên một file duy nhất tên là "main.c"
Điều này có thể tạm chấp nhận trong giai đoạn hiện tại vì chương trình của chúng ta vẫn còn khá ngắn, nhưng sắp tới chương trình của chúng ta sẽ chứa hàng chục, thậm chí hàng trăm functions, nếu bạn đặt tất cả chúng trong cùng một file thì sẽ rất dài !

Cũng chính vì lí do đó mà người ta đã sáng tạo ra cái gọi là **modular programming**. Về mặt nguyên tắc thì nghe có vẻ khá ngu ngốc: thay vì đặt tất cả các dòng code trong một file duy nhất (main.c), chúng ta sẽ chia ra thành nhiều file nhỏ hơn.


Chú ý: tôi sẽ không đặt instruction *system ("PAUSE")* vào phía cuối của *main ( )* nữa. Hãy thêm vào nếu bạn cần nó. Và tôi vẫn khuyên bạn sử dụng IDE **Code::Blocks**. Vì IDE này đã được update để không phải đặt instruction *system ("PAUSE")* ở phía cuối *main ( )* nữa.

## Prototypes

Những bài hướng dẫn trước, tôi yêu cầu bạn đặt các functions trước main. Tại sao vậy?

Tại vì thứ tự sắp xếp có một tầm quan trọng: Khi bạn đặt function trước main, máy tính sẽ đọc và nhớ nó. Khi được gọi lại trong main, máy tính sẽ biết phải kiếm lại function đó ở đâu.

Nhưng nếu bạn đặt sau main, chương trình sẽ không hoạt động vì máy tính vẫn không biết function đó là gì.

Hãy test thử, bạn sẽ thấy ngay !


Nhưng vấn đề này khá bất lợi, đúng không?

Tôi đồng ý với bạn về điểm này !

Nhưng bạn hãy yên tâm, những nhà lập trình trước cũng gặp điều tương tự và họ đã tìm cách khắc phục nó.

Nhờ vào những gì tôi sắp chỉ cho bạn sau đây, bạn sẽ có thể đặt các functions theo bất kì thứ tự nào bạn muốn trong code source.

### Prototype để báo trước một function

Chúng ta bắt đầu việc báo trước cho máy tính những function của chúng ta bằng cách viết các **prototypes.**

Tôi biết bạn đang nghĩ từ ngữ mang đậm chất "high-tech" prototypes này là thứ gì đó ghê gớm lắm nhưng thật ra nó là một thứ hoàn toàn ngu ngốc.

Hãy cùng xem đoạn code đầu tiên của function *dientichHinhChuNhat*:

**C code:**

```
double dientichHinhChuNhat (double chieuRong, double chieuDai)
{
   return chieuRong * chieuDai;
}
```

Hãy copy lại dòng đầu tiên (*double dientichHinhChuNhat...*) và chép vào phần đầu của file source của bạn (sau những dòng *#include*). Và thêm vào một dấu chấm phẩy ở cuối cùng.

Vậy là xong ! Bây giờ bạn có thể đặt function sau main nếu bạn muốn.

Và đoạn code sẽ thay đổi như sau:

**C code:**

```c
#include <stdio.h>
#include <stdlib.h>
 // Doan code sau chinh la prototype cua function dientichHinhChuNhat :

double dientichHinhChuNhat (double chieuRong, double chieuDai);
int main (int argc, char *argv[ ])
{
   printf ("Hinh chu nhat voi chieu rong 5 va chieu dai 10 co dien tich la %f\n",
dientichHinhChuNhat(5, 10));
   printf ("Hinh chu nhat voi chieu rong 2.5 va chieu dai 3.5 co dien tich la %f\n",
dientichHinhChuNhat(2.5, 3.5));
   printf ("Hinh chu nhat voi chieu rong 4.2 va chieu dai 9.7 co dien tich la %f\n",
dientichHinhChuNhat(4.2, 9.7));

   return 0;
}

// function dientichHinhChuNhat bay gio co the dat o bat ki vi tri nao trong code source

 double dientichHinhChuNhat (double chieuRong, double chieuDai)
{
   return chieuRong * chieuDai;
}
```

Và điều thay đổi ở đây là, dòng prototype được thêm vào ở phần đầu code source.
Một prototype thật ra là lời chỉ dẫn cho máy tính. Nó sẽ thông báo với máy tính có sự tồn tại của function (*dientichHinhChuNhat*) với những tham số (parameters) cần đưa vào và type giá trị sẽ xuất ra. Nhờ vậy mà máy tính có thể tự sắp xếp.

Và cũng nhờ vào dòng code này, bạn không còn đau đầu khi chọn vị trí đặt function nữa.

Hãy luôn viết prototypes của các functions có trong chương trình. Chương trình của bạn sẽ không hề bị chậm hơn khi sử dụng nhiều function đâu: và bạn nên tập một thói quen tốt kể từ bây giờ, hãy đặt prototype cho mỗi functions bạn viết.

Chắc bạn cũng thấy function main không có prototype. Và đây cũng là function duy nhất không cần prototype, bởi vì máy tính đã biết rõ nó là gì rồi (tất cả các chương trình đều dùng đến mà, nó bắt buộc phải biết thôi).


Để cho chính xác hơn, bạn cần biết thêm: dòng code prototype không cần thiết phải viết lại tên của các biến số cần cho parameter. Máy tính chỉ cần biết type của các biến số đó thôi.

Vì vậy đơn giản hơn ta có thể viết như sau:

**C code:**

```
double dientichHinhChuNhat (double , double);
```

Và với 2 cách viết đó, chương trình đều chạy tốt, nhưng lợi ích của cách viết đầu tiên là bạn có thể copy-paste nhanh chóng và chỉ thêm vào mỗi dấu chấm phẩy ở cuối.


ĐỪNG QUÊN đặt dấu chấm phẩy ở cuối một prototype. Vì nó giúp cho máy tính có thể nhận ra sự khác nhau giữa prototype và function. Nếu bạn không làm vậy, bạn sẽ mắc lỗi khi biên dịch chương trình.

## Headers

Cho đến lúc này, bạn cũng chỉ sử dụng duy nhất một file source cho project của bạn. Và tôi yêu cầu bạn gọi file source này là main.c

### Cách sử dụng nhiều files trong cùng một project

Trong thực tế, chương trình sẽ không được viết hết toàn bộ trong mỗi file main.c

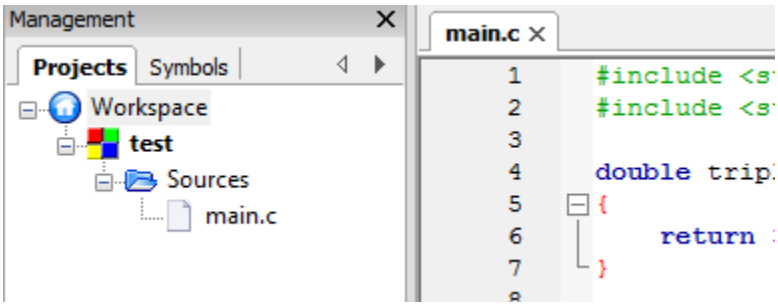
Chắc chắn là chúng ta cũng có thể làm vậy nhưng việc mò mẫm trong một file chứa 10000 dòng code thật sự không thiết thực chút nào. Chính vì vậy, thông thường, một project sẽ được tạo bởi nhiều files.


Nhưng mà project là gì vậy?

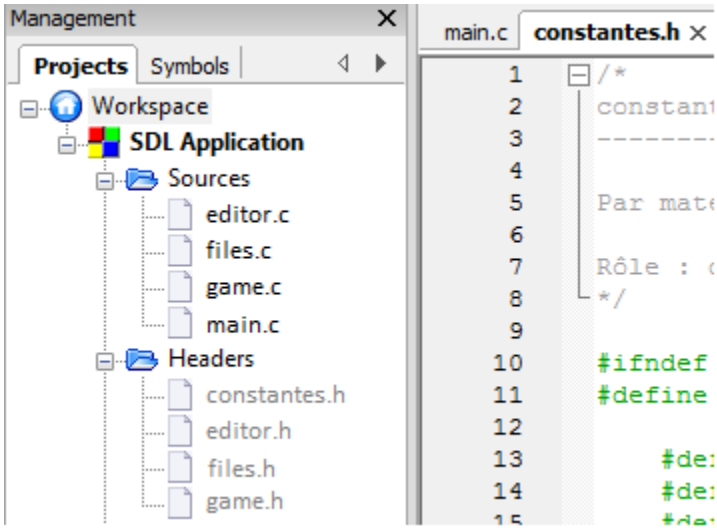
Không phải vậy chứ, bạn quên nó rồi à ?

Ok, không sao, tôi sẽ giải thích lại cho bạn, việc chúng ta thống nhất chung về khái niệm thật sự cần thiết.

Project là tập hợp những files source của chương trình.
Trong thời điểm hiện tại, chương trình của chúng ta chỉ chứa mỗi một file duy nhất. Hãy nhìn vào IDE, phía bên trái:




Bạn thấy trong hình chụp phía trên, bên trái, project này chỉ chứa duy nhất mỗi file main.c
Sau đây, tôi sẽ cho bạn xem hình ảnh một chương trình thật sự, chương trình mà bạn sẽ thực hiện trong những bài sau: trò chơi Sokoban




Bạn thấy đấy, có khá nhiều files khác nhau. Một chương trình bình thường sẽ giống như trên: bạn sẽ thấy nhiều files được liệt kê ở cột bên trái.

Bạn cũng tìm thấy file main.c: bên trong chứa function main. Và hầu như mọi chương trình tôi viết, tôi đều để function main trong file main.c (điều này không bắt buộc, mỗi người có cách sắp xếp khác nhau, nhưng theo tôi tốt nhất bạn nên thực hiện giống tôi ở điểm này).

Nhưng tại sao phải tạo ra nhiều files như vậy? Vậy thì tôi có thể tạo tối đa bao nhiêu file cho mỗi project ?

Điều đó tùy thuộc vào bạn.

Bình thường, người ta thường hay sắp xếp những function có cùng chủ đề vào chung với nhau. Trong hình vẽ trên, trong file editor.c tôi tập hợp những functions liên quan đến việc thay đổi cấp độ của trò chơi, trong file game.c, tôi tập hợp những functions liên quan đến trò chơi,...

## Các files .h và .c

Bạn thấy trong hình vẽ trên, có 2 loại file khác nhau:

- những **file .h**: gọi là **file headers**. Những file này chứa prototype của các functions.
- những **file .c**: là những **file source**. Những file này chứa nội dung của các functions.

Bình thường, người ta rất ít khi để những prototypes trong các file .c giống như vừa rồi chúng ta đã làm trong file main.c (chỉ trừ khi chương trình đó quá nhỏ).

Mỗi file .c tương ứng với một file .h trong đó chứa những prototype của những functions. Xem lại hình trên một lần nữa:

- Có file editor.c (chứa C code của các functions) và file editor.h (chứa prototype các functions đó)
- Tương tự như vậy ta có các file game.c và file game.h
- ...

Làm thế nào máy tính có thể biết được các prototypes nằm ở một file khác ngoài file .c ?

Ta thêm file .h vào chương trình nhờ vào một chỉ thị tiền xử lý (preprocessor directive).
Tập trung nhé, bạn cần chuẩn bị tinh thần để hiểu biết thêm khá nhiều thứ đấy !

Làm sao thêm vào một file header ?...
Bạn biết cách mà, bạn đã làm rất nhiều lần rồi nhớ không?

Chúng ta hãy xem ví dụ đoạn đầu của file jeu.c tôi viết:

**C code:**

```
#include <stdio.h>
#include <stdlib.h>
#include "game.h"
void play (SDL_Surface* screen)
{
//....
```

Và bây giờ bạn đã biết cách thêm vào là sử dụng các chỉ thị tiền xử lý (preprocessor directive) *#include.*

Bây giờ, chú ý những dòng đầu tiên của đoạn code trên:

**C code:**

```
#include <stdio.h>
#include <stdlib.h>
#include "game.h" // them vao file game.h
```

Tôi thêm vào 3 files .h: stdio, stdlib và game.

**Có một sự khác biệt ở đây**: Những file mà bạn để chung trong folder project phải được viết trong ngoặc kép "..." ("game.h") và những file thư viện (đã được cài đặt trước, bình thường nằm trong folder IDE của bạn) phải được viết trong ngoặc nhọn <...> (<stdio.h>).

Tóm lại, thông thường ta sử dụng:

- Ngoặc nhọn < > để thêm vào một file thư viện tìm thấy trong folder của IDE
- Ngoặc kép " " để thêm vào một file tìm thấy trong folder project của bạn ( bên cạnh những file .c)

Lệnh *#include* yêu cầu thêm vào nội dung của file .h vào file .c. Giống như bạn yêu cầu *"Hãy thêm vào đây nội dung của file game.h*"

Vậy nội dung của file game.h là gì?

Chúng ta sẽ tìm thấy trong đó những prototype của các functions nằm trong file game.c !

**C code:**

```
/*
game.h
-----
Noi dung : prototypes of functions in game.
*/

   void play (SDL_Surface* screen);
   void placingPlay (int card[ ][NB_BLOCK_HEIGHT], SDL_Rect *post, int direction);
   void placingFund (int *firstCase, int *secondCase);
```

Và đó là cách một project thật sự hoạt động !

Vậy lợi ích của việc đặt các prototypes vào file .h là gì ?

Lí do cũng khá đơn giản.

Khi code source bạn viết yêu cầu gọi một function, máy tính của bạn bắt buộc phải biết trước function đó là gì, nó cần bao nhiêu tham số (parameter),... Vì thế ta cần đến prototype: giống như một bảng hướng dẫn sử dụng trước khi dùng function cho máy tính.

Câu hỏi trên tương ứng với câu hỏi về thứ tự chạy chương trình: nếu bạn đặt các prototypes trong những file .h (headers) *#include* ở phần đầu những file .c , máy tính của bạn sẽ hiểu được cách sử dụng tất cả các function kể từ giai đoạn bắt đầu chạy chương trình.

Và điều đó giúp bạn ít bận tâm hơn về thứ tự đặt các function trong các files .c. Hiện giờ, bạn chỉ viết một số chương trình chứa khoảng hai hoặc ba functions, bạn vẫn chưa thấy rõ ích lợi của những prototypes nhưng về sau, khi bạn đã có thể viết nhiều functions hơn rồi, nếu bạn không đặt các prototypes trong những file .h, bạn sẽ thường xuyên gặp lỗi trong việc dịch chương trình.

Nếu bạn gọi một function (được viết trong file functions.c) từ file main.c thì bạn cần phải thêm các prototypes của functions.c trong file main.c. Bằng cách tạo một #include "functions.h" ở đầu main.c

Bạn cần nhớ: cứ mỗi lần bạn gọi một function X trong một file, bạn cần phải thêm các prototypes của function này vào file đó. Điều này sẽ giúp trình biên dịch kiểm tra lại xem bạn có gọi đúng cách hay không.

Vậy làm cách nào tôi có thể thêm vào project những file .h và .c ?

Điều này phụ thuộc vào IDE bạn sử dụng nhưng về tổng quát thì qui trình này tương tự nhau: File/New/ Empty file

Việc này sẽ tạo một file trống. File này vẫn chưa có dạng .h hay .c, vì thế bạn cần lưu lại để thông báo điều đó. Cứ lưu lại (mặc dù đó là một file trống !).

Máy tính sẽ hỏi bạn tên của file muốn lưu lại là gì và lúc này bạn có thể lựa chọn giữa .h và .c :

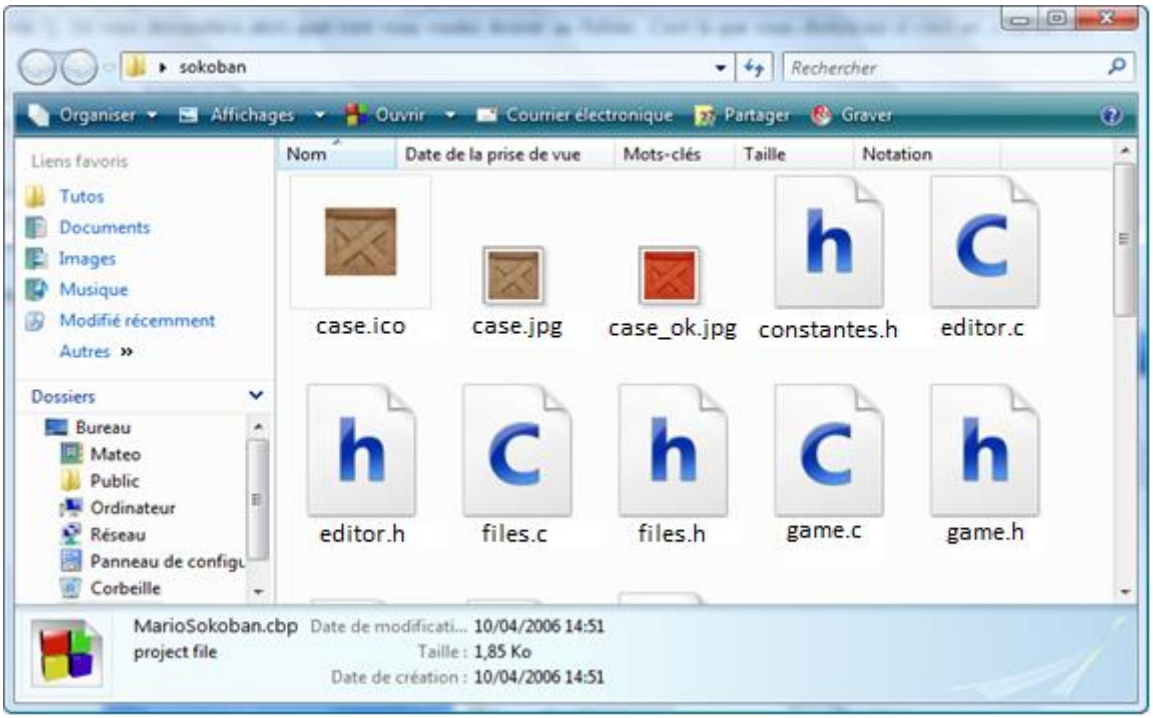
- Nếu bạn đặt tên là [tênfile].c thì nó có dạng .c
- Nếu bạn đặt tên là [tênfile].h thì nó có dạng .h

Đơn giản là như vậy 

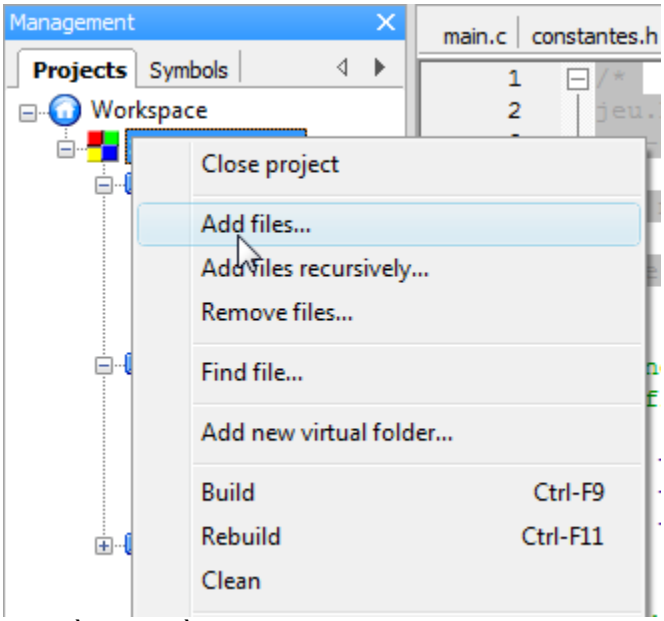
Lưu lại file trong folder chứa những files khác dùng cho project (folder chứa file main.c).

Thông thường, **tất cả những file .h và .c sẽ được lưu lại trong cùng một folder**.

Folder project về sau sẽ tương tự như hình chụp sau, bạn thấy những file .c và .h được đặt chung với nhau:




File bạn vừa tạo đã được lưu lại nhưng nó vẫn chưa được thêm vào project !
Để thêm một file vào project, nhấn chuột phải vào cột bên trái của IDE (nơi bạn tìm thấy danh sách những file dùng cho project) và chọn Add files (Xem hình dưới).




Một cửa sổ hiện ra và yêu cầu bạn cần thêm vào project những files nào. Chọn file bạn vừa tạo. Và bây giờ file này đã nằm trong project và xuất hiện trong danh sách bên trái.

# Cách thêm vào những thư viện chuẩn (standard library)

Bạn đã từng khá thắc mắc vấn đề này đúng không ?
Nếu như ta thêm vào những file stdio.h và stdlib.h, vậy thì những file này nằm đâu đó và chúng ta có thể tìm thấy chúng phải không?

Chính xác là vậy đó !
Thông thường nó đã được cài đặt chung với IDE của bạn. IDE tôi sử dụng là Code::Blocks, những files đó nằm ở đây:

**C:\Program Files\CodeBlocks\MinGW\include**

Và bình thường chúng nằm trong folder include.
Và cũng trong folder đó bạn sẽ tìm thấy khá nhiều files khác. Chúng là những headers (.h) của các thư viện chuẩn (standard libraries), những thư viện này đều đã được viết sẵn (trong Windows, Mac, Linux...), và tại đây bạn cũng tìm thấy file stdio.h và stdlib.h. Mở chúng ra mà xem nếu bạn muốn nhưng có thể bạn sẽ thấy hơi choáng đấy, chúng rất phức tạp, có quá nhiều thứ bạn vẫn chưa biết. Nếu bạn chú ý, thì chúng chứa đầy những prototypes của các functions standard, lấy ví dụ như *printf*.

Ok, bây giờ tôi đã biết cách tìm thấy những prototypes của các functions standard. Nhưng làm cách nào tôi có thể xem những function đó viết như thế nào? Những file .c này nằm ở đâu ?

Bạn không thể có được chúng đâu vì chúng đã được dịch và chứa trong folder lib (viết tắt của library có nghĩa là thư viện). Trong máy tính của tôi, chúng nằm trong folder:

**C:\Program Files\CodeBlocks\MinGW\lib**

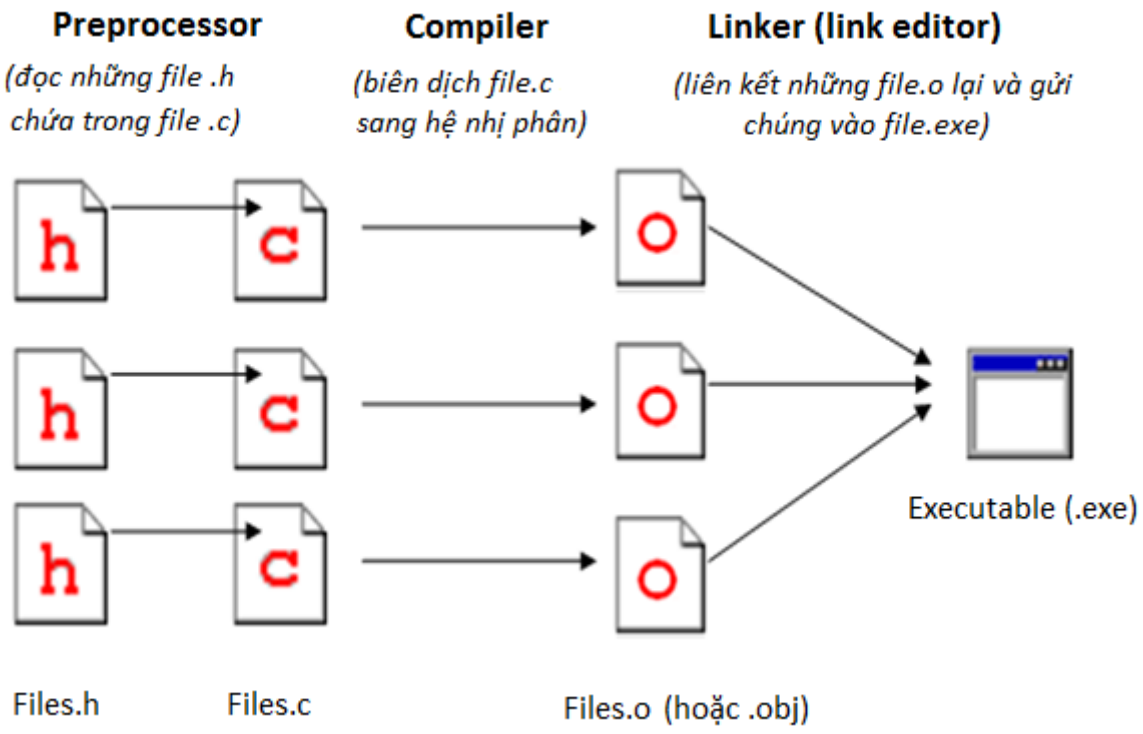
Những file thư viện đã được dịch có đuôi là .a nếu sử dụng Code::Blocks (được biên dịch bởi compiler mingw) và có đuôi là .lib nếu dùng Visual C++(dịch bởi compiler Visual). Đừng cố gắng mở ra để đọc chúng, vì chúng không dành cho người bình thường như chúng ta.

Tóm tắt lại, trong những file .c, bạn cần phải thêm những file .h của các thư viện chuẩn để có thể dùng những function standard như *printf*. Và nhờ đó máy tính của bạn có được bản hướng dẫn sử dụng trước khi dùng và có thể kiểm tra xem bạn có gọi function đúng cách hay không, ví dụ bạn quên một vài parameters dùng cho function X.

# Phân tích quá trình Compilation (separate compilation)

Bạn đã biết một project được cấu thành bởi nhiều files source, chúng ta sẽ tìm hiểu sâu hơn về cách hoạt động của quá trình biên dịch (compilation).

Các bạn đã được xem qua trước đây một biểu đồ khá đơn giản về compilation, để hiểu chi tiết hơn các bạn xem hình vẽ phía dưới, biểu đồ mới này các bạn nên hiểu rõ và phải học thuộc lòng đấy !




*Hình vẽ minh họa quá trình Compilation*

Tôi sẽ phân tích biểu đồ trên cho bạn hiểu rõ

- **Tiền xử lý (preprocessor)**: là một chương trình **khởi động trước khi compilation**. Nhiệm vụ của nó là thực hiện những instructions đặc biệt mà chúng ta đưa vào trong những chỉ thị tiền xử lý, **là những dòng bắt đầu bằng dấu #.**

  Trong thời điểm hiện tại, những chỉ thị tiền xử lý duy nhất mà bạn biết đó là *#include*, cho phép thêm file khác vào file hiện tại. Những chương trình tiền xử lý còn làm được nhiều điều khác mà ta học ở những bài sau nhưng *#include* vẫn là cái quan trọng nhất cần biết.

  Chương trình tiền xử lý sẽ thay thế những dòng *#include* bằng những file chỉ định. Nó sẽ đặt nội dung của những file .h ta yêu cầu vào file .c đang dùng. Và ngay lúc này, những file .c được hoàn chỉnh, nó chứa tất cả những prototypes của các functions bạn dùng ( file .c của bạn lúc này nó sẽ to hơn bình thường).

- **Compilation**: Giai đoạn này rất quan trọng, nó sẽ biến đổi nội dung trong file source của bạn thành code nhị phân mà máy tính có thể hiểu được. Trình biên dịch (Compiler) sẽ lần lượt dịch tất cả các file .c, quan trọng là những file này đã được thêm vào project của bạn (nó phải xuất hiện trong cột danh sách bên trái IDE mà tôi đã giới thiệu ở trên).

  Compiler sẽ tạo ra những file .o (hoặc .obj, điều này phụ thuộc vào loại compiler) tùy theo từng file .c. Đây là các files nhị phân tồn tại tạm thời, và thông thường chúng sẽ bị xóa đi ở cuối quá trình biên dịch, bạn có thể tùy chỉnh lại IDE để lưu lại chúng. Lợi ích của việc lưu lại là, lấy ví dụ bạn muốn dịch lại 1 trong 10 files .c có trong project thì bạn chỉ cần dịch lại mỗi file đó. Những file khác đã có những file .o đã được dịch từ trước.

- **Link editor (linker):** là một chương trình tổng hợp lại những file .o thành một file lớn cuối cùng: executable. Những file executable có đuôi .exe dưới Windows. (Có đuôi khác nếu bạn sử dụng một hệ điều hành khác).

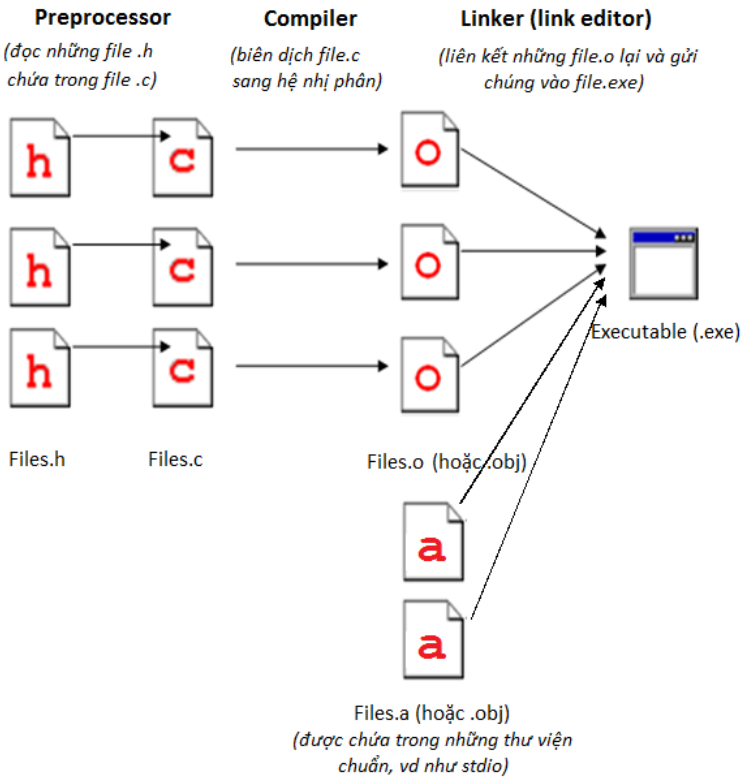
Bạn biết chúng hoạt động như thế nào rồi đó !
Tôi vẫn xin nhắc lại, biểu đồ này rất quan trọng. Nó tạo nên sự khác biệt giữa một người lập trình chuyên chép lại code mà không hiểu nội dung với một người lập trình hiểu và biết họ đang làm gì.

Phần lớn lỗi xảy ra trong quá trình compilation, nhưng đôi khi có những những lỗi xảy ra ở giai đoạn linker. Nghĩa là linker không tổng hợp được những file .o

Hình vẽ trên vẫn còn thiếu đôi chút, những file thư viện đâu ? Điều gì sẽ xảy ra nếu như ta có thêm những thư viện ?

Trong trường hợp này, giai đoạn đầu biểu đồ không thay đổi, nhưng ở giai đoạn linker thì máy tính phải làm việc nhiều hơn một tí. Nó phải tổng hợp lại các file. (tạm thời) với các thư viện mà chúng ta cần (.a .ou .lib tùy theo compiler).

Xem hình sau:




*Và đây là biểu đồ hoàn chỉnh*

Những file .o và . a (hoặc .lib) được tổng hợp lại thành file exécutable.

Và đó là cách mà chúng ta có được một chương trình hoàn chỉnh 100%, chứa tất cả những instructions cần thiết cho máy tính, kể cả các instruction yêu cầu hiển thị một tin nhắn ra màn hình, ví dụ như printf, chứa trong file .a cũng được tổng hợp vào trong file executable.

Trong chương 3, các bạn sẽ được học cách sử dụng những thư viện đồ họa (*Graphics library).*Nó cũng có đuôi .a và chứa những instruction yêu cầu máy tính hiển thị một cửa sổ chẳng hạn.

# Phạm vi sử dụng (scope) của functions và biến số

Trước khi kết thúc bài hướng dẫn, chúng ta sẽ tìm hiểu khái niệm về phạm vi sử dụng của functions và biến số. Chúng ta sẽ xem khi nào những biến số và function có thể sử dụng, có nghĩa là chúng ta có thể gọi được chúng.

## Các biến số riêng của functions

Khi bạn khai báo một biến số trong một function, nó sẽ bị xóa khi function kết thúc:

**C code:**

```
int triple (int soHang)
{
    int ketqua= 0; // Biến số ketqua được lưu lại trong bộ nhớ
    ketqua= 3 * soHang;
    return ketqua;
} // Function kết thúc, biến số ketqua bị xóa khỏi bộ nhớ
```

Một biến số được khai báo trong function chỉ tồn tại khi function đó đang được sử dụng. Điều này có nghĩa như thế nào? Chúng ta không thể dùng nó cho một function khác!

**C code:**

```
int triple (int soHang);
int main (int argc, char *argv[ ])
{
    printf ("triple cua 15 la %d\n", triple (15));
    printf ("triple cua 15 la %d\n", ketqua); // Error - Loi
    return 0;
}
int triple (int soHang)
{
   int ketqua= 0;
   ketqua = 3 * soHang;
   return ketqua;
}
```

Trong main, tôi thử sử dụng biến số *ketqua*. Biến số *ketqua* này được khai báo trong function *triple*, nó không sử dụng được trong function *main* !

**Nắm vững**: một biến số **khai báo trong function nào** thì chỉ có thể **dùng bên trong function đó**. Người ta gọi đó là *biến cục bộ* (local variable).

# Các global variables (Biến toàn cục): cần tránh sử dụng

***Global variable có thể được sử dụng trong tất cả các files***

Chúng ta có thể khai báo những biến số dùng chung cho tất cả các functions chứa trong tất cả các file của project. Tôi sẽ chỉ cho các bạn cách tạo ra nó, nhưng cần tránh sử dụng. Việc sử dụng nó có thể giúp bạn đơn giản hóa code source lúc ban đầu nhưng về sau khi bạn có một số lượng lớn các biến số, nó sẽ dễ dàng khiến bạn nhầm lẫn và gây ra lỗi không đáng có.

Để có thể khai báo một biến số « global » sử dụng chung cho tất cả, chúng ta chỉ cần khai báo ở ngoài những functions. Bạn khai báo chúng ở đoạn đầu của file, sau những dòng *#include*.

**C code:**

```
#include <stdio.h>
#include <stdlib.h>

int ketqua = 0; // khai báo một biến số global

void triple (int soHang); //prototype cua function

int main (int argc, char *argv[ ])
{
    triple (15); // ta goi function triple, no se thay doi gia tri cua bien so ketqua
    printf ("triple cua 15 la %d\n", ketqua); // Hien thi gia tri cua ketqua
    return 0;
}
void triple (int soHang)
{
   ketqua = 3 * soHang;
}
```

Trong ví dụ này, function *triple* không trả về giá trị nào (*void*). Function *triple* thay đổi giá trị biến số global *ketqua* và function *main* có thể lưu lại giá trị đó.

Biến số *ketqua* có thể sẽ được sử dụng cho tất cả các file trong project, nó có thể được gọi lại trong TẤT CẢ các functions có trong chương trình.
Dạng biến số này cần tránh sử dụng. Cách tốt nhất là sử dụng *return* mỗi khi muốn function trả về một giá trị.

***Global variable sử dụng riêng cho một file***

Biến số global mà chúng ta vừa thấy sử dụng được cho tất cả các file trong project.

Chúng ta có thể tạo ra những biến số dùng riêng cho file chứa nó. Biến số này có thể được sử dụng cho các functions xuất hiện trong file đó, không dùng được chung cho tất cả các functions có trong chương trình.

Để tạo một biến số như vậy, ta chỉ đơn giản thêm vào từ khóa static ở phía trước:
**C code:**

```
static int ketqua = 0;
```

## Static Variable trong một function

**Chú ý**: có đôi chút khó hiểu ở đây. Trong một function, nếu bạn thêm từ khóa static trước dòng khai báo biến số, biến số đó sẽ không bị xóa đi khi function kết thúc, giá trị của biến số đó vẫn được giữ lại. Và lần gọi function sau, **biến số sẽ giữ lại giá trị đó**. Có sự khác biệt với những biến số global.

Ví dụ:
**C code:**

```
int triple (int soHang)
{
    static int ketqua = 0; // Bien so ketqua duoc tao ra lan dau khi function duoc goi
    ketqua = 3 * soHang;
    return ketqua;
} // Bien so ketqua khong bi xoa di khi function ket thuc
```

Vậy điều này có ý nghĩa như thế nào ?

Người ta có thể gọi lại biến số trong những lần sau và biến số *ketqua* vẫn giữ nguyên giá trị.

Xem thêm ví dụ sau để hiểu rõ hơn:

**C code:**

```c
int increase( );

int main (int argc, char *argv[ ])
{
   printf ("%d\n", increase ( ));
   printf ("%d\n", increase ( ));
   printf ("%d\n", increase ( ));
   printf ("%d\n", increase ( ));

   return 0;
}

int increase ( )
{
   static int soHang = 0;
   soHang++;
   return soHang;
}
```

**Console**

```
1
2
3
4
```

Khi ta gọi function *increase*, biến số *soHang* được tạo ra với giá trị 0. Nó được tăng lên 1, và khi function kết thúc nó không bị xóa đi.

Khi function này được gọi lại lần nữa, dòng khai báo biến số bị bỏ qua. Máy tính sẽ sử dụng tiếp tục với biến số *soHang* được tạo ra trước đó.

Giá trị biến số *soHang* trước đó là 1, bây giờ thành 2, rồi thành 3, thành 4...

## Các local functions dùng riêng cho một file

Để kết thúc phần này, tôi sẽ chỉ bạn về phạm vi sử dụng của các functions.
Bình thường, khi bạn tạo một function, nó sẽ được dùng chung cho toàn bộ chương trình. Nghĩa là nó cũng có thể được sử dụng cho bất kì file .c nào khác.

Nhưng nếu bạn cần tạo một function chỉ dùng riêng cho mỗi file chứa nó. Bạn chỉ cần thêm vào từ khóa *static* trước function đó:
**C code:**

```
static int triple (int soHang)
{
    //instructions
}
```

Hãy nghĩ đến việc cập nhật prototype cho function này nhé.
**C code:**

```
static int triple (int soHang);
```

Bây giờ, function static *triple* chỉ có thể gọi bởi một function khác nằm chung trong file chứa nó. Nếu bạn thử gọi function triple bởi một function khác chứa trong file khác, sẽ không hoạt động.

Tóm tắt lại những phạm vi sử dụng có thể có của các biến số:

- Một biến số khai báo trong một function sẽ bị xóa đi khi function kết thúc, **nó chỉ được sử dụng riêng cho function này**.
- Một biến số khai báo trong một function với từ khóa static ở phía trước sẽ không bị xóa khi function kết thúc, **nó sẽ lưu lại giá trị và cập nhật dọc theo chương trình.**
- Một biến số khai báo bên ngoài các functions là một biến số global, **có thể sử dụng cho tất cả các functions của tất cả các file source có trong project.**
- Một biến số global với từ khóa static ở phía trước là biến số global **chỉ được sử dụng riêng cho file chứa nó**, không dùng được bởi các function viết ở các file khác.

Tương tự, đây là các phạm vi sử dụng có thể có của các function:

- **Một function mặc định có thể sử dụng chung cho tất cả các files trong project**, nó có thể gọi ra từ bất cứ vị trí nào trong các file khác.
- Nếu ta muốn một function **dùng riêng cho mỗi file chứa nó**, bắt buộc phải thêm vào từ khóa *static* ở phía trước.