

CHƯƠNG II - [NGÔN NGỮ LẬP TRÌNH C] NHỮNG KỸ THUẬT NÂNG CAO

1) Modular Programming (Mô-đun Lập trình)

- Prototypes.
- Headers.
- Separate Compilation.
- Phạm Vi Sử Dụng (scope) của functions và biến số.

2) Pointer (Con trỏ)

- Vấn đề nan giải.
- Địa chỉ trong bộ nhớ.
- Cách sử dụng pointer (con trỏ).
- Cách sử dụng con trỏ trong một function.

3) Arrays (Mảng)

- Các Arrays trong bộ nhớ.
- Cách tạo một array.
- Liệt kê các giá trị trong array.
- Tạo một function để liệt kê các giá trị trong array.
- Bài tập thực hành.

4) Chuỗi ký tự

- Biến kiểu Char.
- String hay còn gọi là mảng ký tự.
- Các thao tác với chuỗi ký tự.

5) Preprocessor – Tiền xử lý

- #include.
- #define.
- Macro.
- Condition.

6) Tạo ra những biến kiểu riêng của bạn

- Cấu trúc (struct).
- Mảng cấu trúc.
- Typedef.

7) Những thao tác làm việc với tập tin (file)

- Mở và đóng tập tin.
- Ghi dữ liệu vào tập tin.
- Đọc dữ liệu trong tập tin.
- Di chuyển tập tin.
- Đổi tên và xóa tập tin.

8) Cấp phát động

- Kích thước của biến.
- Cấp phát bộ nhớ động.
- Giải phóng bộ nhớ.

9) Test Program: Người Treo Cổ

- Một số chỉ dẫn.
- Giải pháp 1:
- Giải pháp 2:
- Ý tưởng cải tiến.

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 lý 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 kiểm 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:

```
#include <stdio.h>
#include <stdlib.h>
// Đoạn code sau chính là prototype của function dientichHinhChuNhat :

double dientichHinhChuNhat (double chieuRong, double chieuDai);
int main (int argc, char *argv[ ])
{
    printf ("Hình chu nhật với chiều rộng 5 và chiều dài 10 có diện tích là %f\n",
    dientichHinhChuNhat(5, 10));
    printf ("Hình chu nhật với chiều rộng 2.5 và chiều dài 3.5 có diện tích là %f\n",
    dientichHinhChuNhat(2.5, 3.5));
    printf ("Hình chu nhật với chiều rộng 4.2 và chiều dài 9.7 có diện tích là %f\n",
    dientichHinhChuNhat(4.2, 9.7));

    return 0;
}

// function dientichHinhChuNhat bây giờ có thể đặt ở bất kỳ vị trí nào 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.



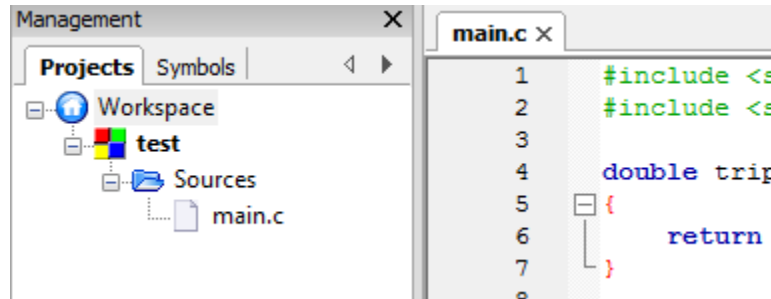
Nhưng mà project là gì vậy?

Không phải vậy chứ, bạn quên nó rồi à? 😊

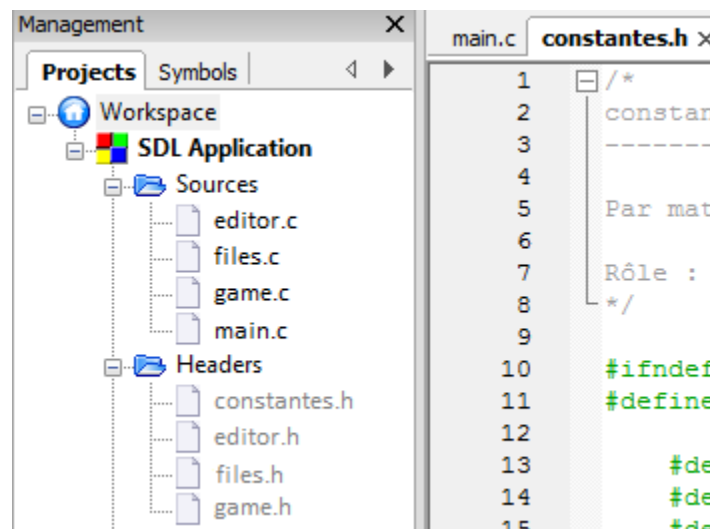
Ok, không sao, tôi sẽ giải thích lại cho bạn, việc chúng ta thống nhất chung về khái niệm thật sự cần thiết. 😊

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" // thêm vào 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 :

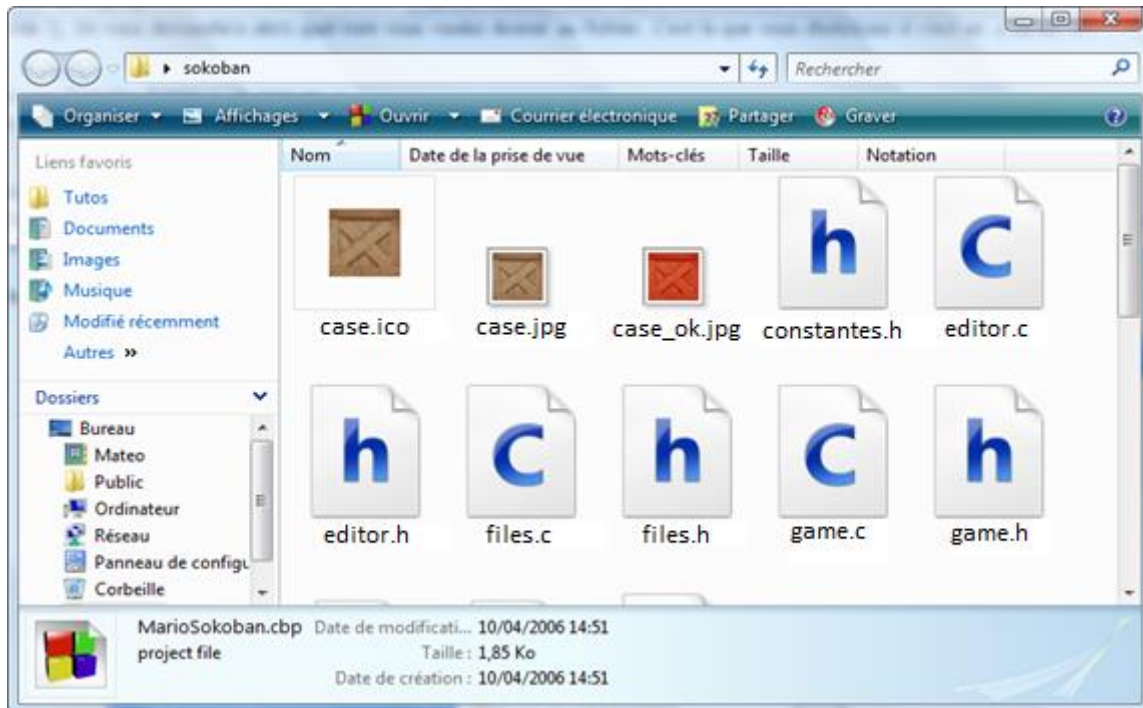
- Nếu bạn đặt tên là [tênfile].c thì nó có dạng .c
- Nếu bạn đặt tên là [tênfile].h thì nó có dạng .h

Đơn giản là như vậy 😊

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

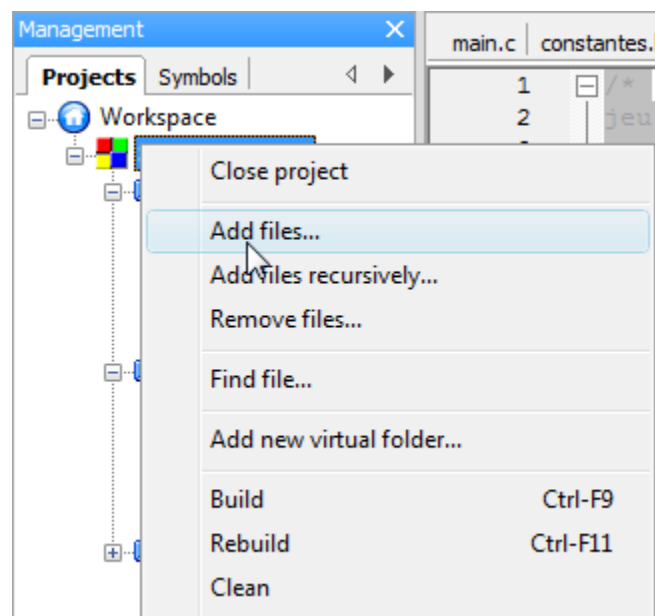
Thông thường, **tất cả những file .h và .c sẽ được lưu lại trong cùng một folder.**

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

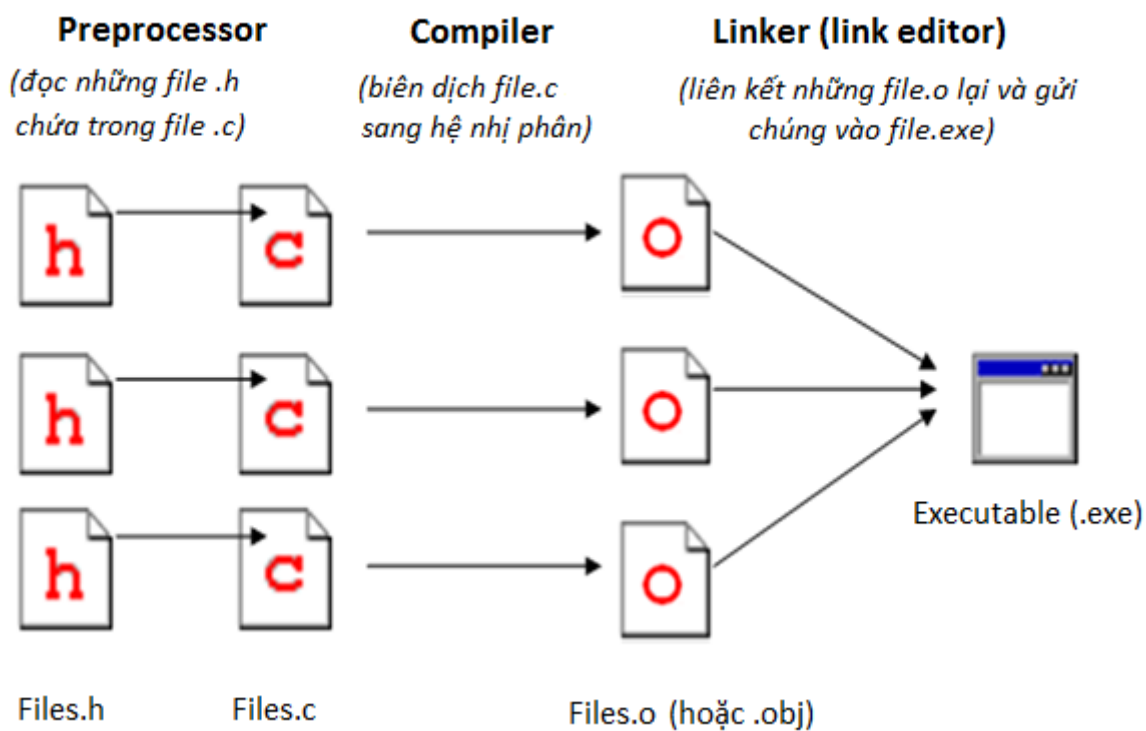
Những file thư viện đã được dịch có đuôi là `.a` nếu sử dụng Code::Blocks (được biên dịch bởi compiler mingw) và có đuôi là `.lib` nếu dùng Visual C++(dịch bởi compiler Visual). Đừng cố gắng mở ra để đọc chúng, vì chúng không dành cho người bình thường như chúng ta. 😊

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. 😊

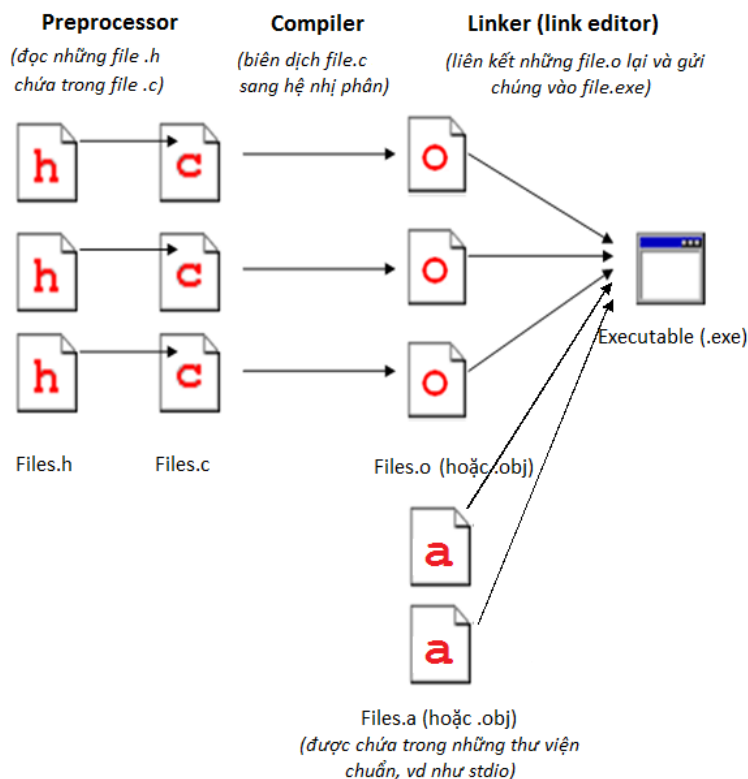
Phần lớn lỗi xảy ra trong quá trình compilation, nhưng đôi khi có những lỗi xảy ra ở giai đoạn linker. Nghĩa là linker không tổng hợp được những file .o



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 executable.

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 của 15 là %d\n", triple (15));
    printf ("triple của 15 là %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 của function

int main (int argc, char *argv[ ])
{
    triple (15); // ta gọi function triple, nó sẽ thay đổi giá trị của biến so ketqua
    printf ("triple của 15 là %d\n", ketqua); // Hiện thị giá trị của 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; // Biến so ketqua duoc tao ra lan dau khi function duoc goi
    ketqua = 3 * soHang;
    return ketqua;
} // Biến 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:

```
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.

Bài 2: Pointer

Con trỏ

Đã đến lúc chúng ta tìm hiểu về con trỏ. Hãy ra hít một hơi thật sâu trước khi bắt đầu vì tôi biết bài học này chắc chắn sẽ không khiến bạn thấy thú vị. 😊 Nhưng con trỏ là một khái niệm được sử dụng rất thường xuyên trong C. Nói về tầm quan trọng, chúng ta không thể nào lập trình trên ngôn ngữ C mà không dùng đến con trỏ, và bạn cũng đã từng dùng nó mà không biết. 😊

Phần lớn những người bắt đầu học C thường xuyên vấp ngã trong phần kiến thức về con trỏ. Và tôi hi vọng bài học này sẽ giúp các bạn không nằm trong số đó. Hãy tập trung gấp đôi bình thường và bỏ thêm thời gian để hiểu rõ từng biểu đồ, ví dụ có trong bài học này. 🧐

Một vấn đề nan giải

Đây là một trong những vấn đề lớn liên quan đến con trỏ, các bạn mới bắt đầu thường bị nhầm lẫn, cảm thấy khó khăn trong việc nắm vững cách hoạt động và sử dụng.

"Con trỏ rất cần thiết, và chúng ta sẽ thường xuyên dùng đến nó, hãy tin tôi !" 🧐

Tôi sẽ cho bạn xem một ví dụ mà các bạn không thể nào giải quyết được nếu không sử dụng đến con trỏ. Đây cũng là tiêu điểm của bài học này, tôi sẽ hướng dẫn cách giải quyết ở cuối bài học.

Đây là vấn đề: Tôi muốn viết một function trả về hai giá trị. Việc này là **không thể** vì mỗi function chỉ có thể trả về duy nhất một giá trị.

C code:

```
int function ( )  
{  
    return giatri;  
}
```

Nếu ta khai báo function với type *int*, thì ta sẽ nhận được một số dạng *int* (nhờ vào instruction *return*).

Chúng ta cũng đã học cách viết một function không trả về bất cứ giá trị nào với từ khóa *void*:

C code:

```
void function( )  
{  
  
}
```

Nhưng để nhận được hai giá trị trả về cùng lúc thật sự là việc không thể. Chúng ta không thể sử dụng hai *return* cùng lúc.

Giả sử tôi muốn viết một function, trong parameter tôi sẽ cho nó một giá trị tính bằng phút, tôi muốn nó chuyển thành giờ và phút tương ứng:

1. Nếu ta đưa vào giá trị 45, function sẽ trả về 0 giờ và 45 phút.
2. Nếu ta đưa vào giá trị 60, function sẽ trả về 1 giờ và 0 phút.
3. Nếu ta đưa vào giá trị 90, function sẽ trả về 1 giờ và 30 phút .

Nhìn có vẻ khá đơn giản, nhưng hãy cùng test đoạn code sau:

C code:

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

/* Tôi đặt các prototypes trên cùng. Vì đây là một chương trình khá nhỏ nên tôi không
đặt chúng trong một file.h, nhưng trong một chương trình thật sự, tôi sẽ đặt chúng trong
một file.h như đã hướng dẫn ở bài trước*/
void chuyenDoi(int gio, int phut);
int main (int argc, char *argv[ ])
{
    int gio = 0, phut = 90;
    /*Chúng ta có biến số "phut" giá trị 90. Sau khi kết thúc function, tôi muốn biến số
    "gio" nhận giá trị 1 và biến số "phut" nhận giá trị 30 */
    chuyenDoi(gio, phut);
    printf ("%d giờ và %d phút", gio, phut);
    return 0;
}
void chuyenDoi(int gio, int phut)
{
    gio= phut/ 60; // 90 / 60 = 1
    phut= phut% 60; // 90 % 60 = 30
}
```

Và đây là kết quả:

Console

0 gio va 90 phut

Ặc... chương trình đã không hoạt động. Vì sao vậy? 😞

Khi bạn gửi giá trị của một biến số vào vị trí parameter của một function, một bản sao của biến số này được tạo ra. Nói cách khác, biến số "gio" trong function *chuyenDoi* không phải là biến số "gio" trong function *main*! Nó chỉ là bản sao!

Function *chuyenDoi* đã thực hiện nhiệm vụ của nó. Trong function *chuyenDoi*, những biến số "gio" và "phut" nhận giá trị chính xác: 1 và 30. 😊

Nhưng sau đó, function kết thúc khi dấu ngoặc } đóng lại. Như ta đã học ở bài học trước, tất cả những biến số tạo ra trong một function sẽ bị xóa đi khi function đó kết thúc. Và ở đây, biến số *gio* và *phut* đã bị xóa đi. Sau đó chương trình tiếp tục phần tiếp theo của *main*, và ở đó biến số *gio* và *phut* của *main* giá trị vẫn là 0 và 90. Đó là lí do bạn thất bại!



Cần ghi thêm ở đây, function tạo ra một bản sao cho biến số ta gửi vào nó, nên bạn không cần phải gọi tên biến số đó chính xác giống như cách bạn gọi ở *main*.

Để rõ ràng hơn, bạn xem đoạn code sau:

```
void chuyenDoi (int g, int p)
(g thay cho gio và p thay cho phút)
```

Và tiếp theo, hãy thử tìm nhiều cách khác sửa đổi chương trình trên, như trả về một giá trị sau khi kết thúc function (sử dụng *return* và thay đổi *type* function thành *int*), bạn chỉ nhận được một trong hai giá trị bạn cần. Bạn không thể nào nhận được cùng lúc hai giá trị. Và bạn tuyệt đối không được sử dụng biến số *global*, lí do tôi đã giải thích ở bài trước.

Và đó là vấn đề khó khăn đặt ra 😞, vậy con trỏ sẽ giải quyết vấn đề trên như thế nào?

Địa chỉ trong bộ nhớ Nhắc lại kiến thức

Bạn còn nhớ bài học về những biến số không?

Dù có hay không, tôi vẫn khuyến khích bạn xem lại phần đầu của bài học, phần "Công việc của bộ nhớ". 😊

Trong đó có một biểu đồ khá quan trọng mà tôi cần nhắc lại trước khi dạy bạn những kiến thức mới:

Địa chỉ	Giá trị
0	145
1	3.8028322
2	0.827551
3	3901930
...	...
3 448 765 900 126	940.5118

Cách sắp xếp trong bộ nhớ (RAM)

Đó là cách trình bày một RAM trong máy tính của bạn.

Hãy đọc kĩ từng dòng trong biểu đồ. Dòng đầu tiên tương ứng với "0" đầu tiên của bộ nhớ (RAM). Mỗi ô tương ứng với một số, là địa chỉ của nó (address)! Bộ nhớ chứa một số lượng lớn địa chỉ, bắt đầu từ địa chỉ 0 đến một số nào đó (một số vô cùng lớn, số lượng địa chỉ phụ thuộc vào dung lượng bộ nhớ được lắp đặt trong từng máy tính).

Mỗi địa chỉ có thể chứa một số. Một và chỉ một. Ta không thể nào chứa 2 số trong cùng một địa chỉ.

Bộ nhớ của bạn tạo ra chỉ để chứa những con số. Nó không thể chứa chữ cái cũng như đoạn văn. Để giải quyết vấn đề này, người ta tạo ra những bảng mã chứa trong đó số và chữ cái tương ứng.

Ví dụ "Số 89 tương ứng với chữ cái Y". Vấn đề này sẽ được giải thích rõ hơn ở bài học sau. Bây giờ, chúng ta chỉ tập trung vào cách hoạt động của bộ nhớ. 😊

Địa chỉ và giá trị

Khi bạn tạo ra một biến số *tuoi* type *int*, lấy ví dụ:

C code:

```
int tuoi = 10;
```

... chương trình của bạn sẽ yêu cầu hệ điều hành (ví dụ là Windows) quyền sử dụng một ít bộ nhớ. Hệ điều hành sẽ trả lời bằng cách đưa ra địa chỉ bộ nhớ được phép chứa con số bạn cần. Đây cũng là một trong những nhiệm vụ chính của hệ điều hành:

Khi chúng ta yêu cầu mượn bộ nhớ cho chương trình. Máy tính giống như ông chủ, nó điều hành từng chương trình và kiểm tra xem chúng có quyền sử dụng bộ nhớ tại vị trí được cấp hay không.

Và đây là một trong những nguyên nhân khiến máy tính bạn bị đơ: Nếu chương trình đột nhiên hoạt động trên một vùng bộ nhớ không cho phép. Hệ điều hành (OS) sẽ từ chối và dừng ngay chương trình, giống như nói với bạn "Mày nghĩ ai là chủ ở đây?" 😏 Người dùng, sẽ nhìn thấy một cửa sổ hiện lên thông báo dạng "Chương trình bị dừng lại do thực hiện một công việc không được phép". 😞

Quay trở lại với biến số *tuoi*. Giá trị 10 được đưa vào một vị trí nào đó trong bộ nhớ, lấy ví dụ nó được đưa vào địa chỉ 4655. Và điều xảy ra ở đây là (nhiệm vụ của compiler), từ *tuoi* trong chương trình sẽ thay thế bằng địa chỉ 4655 khi được chạy.

Việc đó giống như, mỗi khi bạn điền vào *tuoi* trong code source, chúng sẽ được chuyển thành 4655, và máy tính sẽ biết được cần đến địa chỉ nào trong bộ nhớ để lấy giá trị. Và ngay sau đó, máy tính xem giá trị được chứa trong địa chỉ 4655 và trả lời chúng ta "biến số *tuoi* có giá trị là 10"! 😊

Và để lấy giá trị một biến số, đơn giản chỉ cần đánh tên của biến số đó vào code source. Nếu ta muốn hiển thị tuổi, ta có thể sử dụng function *printf*:

C code:

```
printf ("Bien so tuoi co gia tri la : %d", tuoi);
```

Không có điều gì mới với dòng code trên đúng ko.

Khuyến mãi thêm!

Bạn đã biết cách hiển thị giá trị của một biến số, nhưng bạn có biết chúng ta cũng có thể hiển thị địa chỉ của biến số đó? 😊

...Đương nhiên là bạn chưa biết rồi! 😊

Để hiển thị địa chỉ của một biến số, chúng ta cần sử dụng kí hiệu **%p** (p ở đây viết tắt của từ pointer) trong *printf*. Mặt khác, chúng ta phải đưa vào *printf* địa chỉ của biến số đó và để làm việc này, bạn cần phải đặt kí hiệu **&** trước biến số đó (*tuoi*), giống như cách tôi hướng dẫn bạn sử dụng *scanf*, xem code sau:

C code:

```
printf ("Địa chỉ của biến số tuoi là %p", &tuoi);
```

Kết quả

Console

```
Địa chỉ của biến số tuoi là 0023FF74
```

Đó là địa chỉ của biến số *tuoi* trong thời điểm chương trình hoạt động. Vâng, 0023FF74 là một số, nó đơn giản chỉ được viết trên hệ hexadecimal (thập lục phân), thay vì hệ decimal (thập phân) mà chúng ta thường sử dụng. Nếu bạn thay kí hiệu **%p** thành **%d**, bạn sẽ nhận được một số thập phân mà bạn biết.

Nếu bạn chạy chương trình này trên máy tính của bạn, địa chỉ sẽ khác hoàn toàn. 😊 Tất cả phụ thuộc vào phần cứng có trong bộ nhớ, chương trình bạn đang dùng,... Hoàn toàn không có khả năng báo trước địa chỉ nào của biến số sẽ được cấp. Nếu bạn thử chạy chương trình liên tục nhiều lần, địa chỉ có thể sẽ không đổi trong thời điểm đó. Nhưng nếu bạn khởi động lại máy tính, chương trình chắc chắn sẽ hiển thị một giá trị khác.

Vậy chúng ta sẽ làm gì với tất cả những thứ đó?

Tôi cần bạn nắm vững những điều sau:

- *tuoi*: tượng trưng cho **giá trị** của biến số.
- *&tuoi*: tượng trưng cho **địa chỉ** của biến số.

Với *tuoi*, máy tính sẽ đọc và gửi lại giá trị của biến số.

Với *&tuoi*, máy tính sẽ nói với chúng ta ở địa chỉ nào sẽ tìm thấy biến số.

Cách sử dụng pointers (con trỏ)

Đến bây giờ, bạn chỉ có thể tạo biến số để chứa các số hạng. Và sau đây chúng ta sẽ học cách tạo ra những biến số chứa địa chỉ của chúng, những biến số này gọi là con trỏ.



Nhưng... Địa chỉ cũng là một số đúng không? Như vậy số này cần phải chứa trong một biến số khác và cứ như thế nó sẽ lặp lại mãi sao?

Chính xác. Nhưng các con số này sẽ có một kí hiệu đặc biệt để nhận biết: địa chỉ của một biến số khác trong bộ nhớ.

Cách tạo một con trỏ

Để tạo một biến số dạng con trỏ, ta cần phải thêm kí tự * trước tên của biến số.

C code:

```
int *pointer;
```



Bạn cần biết rằng chúng ta cũng có thể viết:

```
int* pointer;
```

vẫn hoạt động tương tự như trên. Nhưng cách viết đầu được khuyến khích hơn. Bởi vì trong trường hợp bạn cần khai báo cùng lúc nhiều con trỏ trong cùng một dòng, bạn bắt buộc phải đặt * trước mỗi tên con trỏ:

```
int *pointer1, *pointer2, *pointer3;
```

Giống như điều tôi dạy bạn khi khai báo biến số, bạn cần cho nó giá trị ngay khi khởi tạo, rất quan trọng, bằng cách cho nó giá trị 0 (lấy ví dụ với biến số). Và đối với con trỏ, điều này còn quan trọng hơn nữa! Để khởi tạo con trỏ, có nghĩa là cho nó một giá trị mặc định, người ta không dùng giá trị 0 mà dùng từ khóa *NULL* (phải được viết hoa):

C code:

```
int *pointer = NULL;
```

Bạn đã khởi tạo một con trỏ giá trị *NULL*. Như vậy, bạn chắc rằng con trỏ của bạn không chứa địa chỉ nào.

Việc này diễn ra như thế nào? Đoạn mã trên sẽ đặt trước một chỗ trong bộ nhớ, giống như cách bạn tạo ra một biến số thông thường. Nhưng điều thay đổi ở đây là giá trị của con trỏ chỉ dùng để chứa địa chỉ của một biến số khác.

Vậy thử xem voi địa chỉ của *tuoi* thì sao?

Và đây là cách chỉ ra địa chỉ của một biến số (*tuoi*) dựa trên giá trị của nó (bằng cách sử dụng kí tự &), nào bắt đầu thôi!

C code:

```
int tuoi = 10;  
int *pointerTuoi = &tuoi;
```

Dòng thứ nhất : "Tạo một biến số type int có giá trị là 10".

Dòng thứ hai : "Tạo một con trỏ có giá trị là địa chỉ của biến số *tuoi*".

Dòng thứ hai thực hiện cùng lúc hai việc. Nếu bạn thấy phức tạp nên không muốn gộp hai việc với nhau, tôi sẽ tách biệt chúng bằng cách chia thành hai giai đoạn, xem đoạn code sau :

C code:

```
int tuoi = 10;  
int *pointerTuoi; // 1) có nghĩa là "Tôi tạo một con trỏ pointerTuoi"  
pointerTuoi = &tuoi; // 2) có nghĩa là "con trỏ pointerTuoi chứa địa chỉ của biến số tuoi"
```

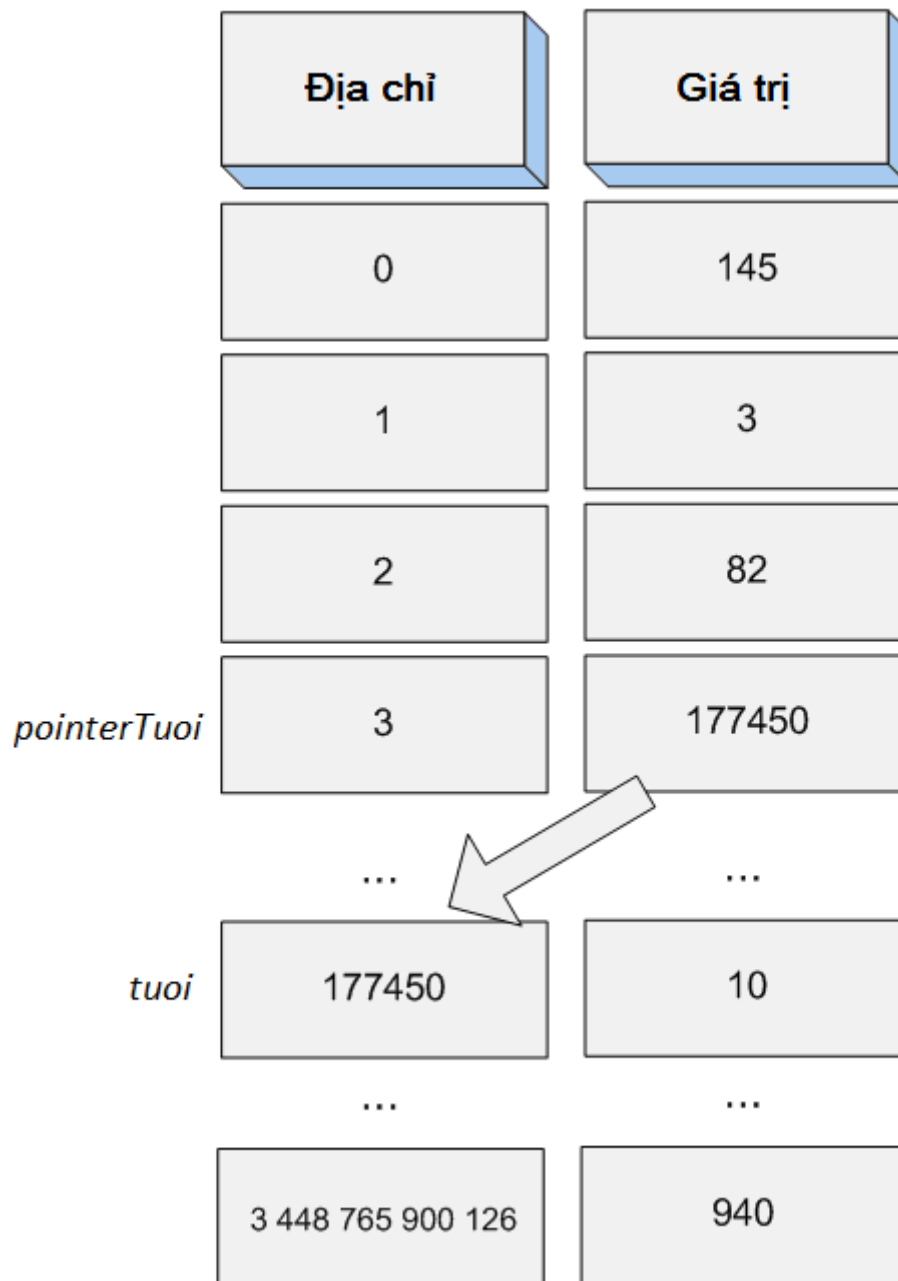
Bạn cần nhớ rằng không có type "pointer" như type int hay double. Người ta không ghi như sau:
`pointer pointerTuoi;`

Thay vì vậy, chúng ta sẽ sử dụng kí tự *****, sau *int*. Tại sao lại như vậy? Thật ra, chúng ta cần phải chỉ rõ type của biến số mà con trỏ sẽ chứa địa chỉ của nó.

Ở trên, *pointerTuoi* sẽ chứa địa chỉ của biến số *tuoi* (type là *int*), vậy con trỏ phải có type *int**. Nếu biến số *tuoi* có type là *double*, ta phải viết *double *pointerTuoi*.

Giá trị của con trỏ *pointerTuoi* chỉ ra địa chỉ của biến số *tuoi*.

Hình sau tóm tắt lại những gì đã diễn ra trong bộ nhớ:



Trong biểu đồ trên, biến số *tuoi* được đặt vào ô địa chỉ 177450 (và bạn thấy tại đó giá trị tương ứng là 10), và con trỏ *pointerTuoi* được đặt vào ô địa chỉ 3 (tất cả địa chỉ đều được chọn ngẫu nhiên, và các địa chỉ trong biểu đồ cũng do tôi tự viết ra 🤪).

Khi con trỏ được tạo ra, hệ điều hành sẽ dành một ô trong bộ nhớ giống như cách nó tạo ra với biến số *tuoi*. Khác nhau là giá trị của *pointerTuoi* là địa chỉ của biến số *tuoi*.

Chúng ta bắt đầu tiến vào thế giới huyền diệu của những con trỏ. Thế giới bí mật của những chương trình viết trên ngôn ngữ C (C++)



Ok, nhưng... nó dùng để làm gì ?

Hiển nhiên, nó không giúp máy tính bạn biến đổi thành máy làm ra café. Chỉ là, ta có một con trỏ *pointerTuoi* chứa địa chỉ của biến số *tuoi*. Hãy dùng *printf* xem thử nó chứa gì trong đó:

C code:

```
int tuoi = 10;
int *pointerTuoi = &tuoi;
printf ("%d", pointerTuoi);
```

Console

177450

uhm, thật sự điều này không có gì ngạc nhiên lắm. Người ta yêu cầu giá trị chứa trong *pointerTuoi* và đó là địa chỉ của biến số *tuoi* (177450).

Vậy làm sao có được **giá trị của biến số** mà *pointerTuoi* chỉ vào?

Chúng ta phải đặt kí tự ***** trước tên của con trỏ:

C code:

```
int tuoi = 10;
int *pointerTuoi = &tuoi;

printf ("%d", *pointerTuoi);
```

Console

10

Đó, bạn làm được rồi đấy! 🎉 Bằng cách đặt kí tự ***** trước tên con trỏ, ta nhận được giá trị của biến số *tuoi*. 😊

Nếu chúng ta sử dụng kí tự **&** trước tên của con trỏ, chúng ta sẽ nhận được địa chỉ để tìm thấy con trỏ (trong trường hợp này là 3).



Vậy tôi đạt được điều gì ở đây? Nãy giờ tôi chỉ thấy các vấn đề càng lúc càng rắc rối thêm. Tôi thấy không cần thiết hiển thị giá trị của biến số *tuoi* bằng con trỏ! Trước đây chúng ta vẫn có thể hiển thị giá trị của biến số mà đâu cần đến con trỏ.

Đây là một câu hỏi chính đáng (mà bạn bắt buộc phải đặt ra cho bản thân). Sau tất cả những điều rắc rối bạn vừa được học, hiển nhiên là bạn muốn biết tác dụng của nó, nhưng tại thời điểm này, tôi khó có thể giải thích hết, qua các bài học sau, từng chút một, các bạn sẽ thấy nó không đơn giản được tạo ra chỉ để làm mọi thứ phức tạp thêm. 😊

Xin bạn hãy bỏ qua một bên cái cảm giác khó chịu tôi tạo ra cho bạn ("Tất cả mọi thứ ở trên chỉ để làm những việc này thôi sao?" 😞).

Nếu bạn hiểu được nguyên lý hoạt động, thì chắc chắn, những thắc mắc sẽ được sáng tỏ trong các bài học sau. 😊

Những điều cần nắm vững

Đây là những điều mà bạn cần hiểu và nắm vững trước khi tiếp tục bài học:

- Đối với một biến số, lấy ví dụ biến số *tuoi*:
 - *tuoi* có nghĩa là: "Tôi muốn giá trị của biến số *tuoi*",
 - *&tuoi* có nghĩa là: "Tôi muốn địa chỉ để tìm thấy biến số *tuoi*";
- Đối với một con trỏ, lấy ví dụ con trỏ *pointerTuoi*:
 - *pointerTuoi* có nghĩa là: "Tôi muốn giá trị của con trỏ *pointerTuoi*" (giá trị này là địa chỉ của một biến),
 - **pointerTuoi* có nghĩa là: "Tôi muốn giá trị của biến số mà con trỏ *pointerTuoi* chỉ vào".

Để có thể hiểu được 4 điểm chính trên. Bạn cần test nhiều lần để hiểu cách nó hoạt động. Biểu đồ sau đây giúp bạn có thể hiểu rõ hơn:

Địa chỉ	Giá trị
0	145
1	3
2	82
3	177450 <i>pointerTuoi</i>
...	...
177450 <i>&tuoi</i>	10 <i>tuoi</i> <i>*pointerTuoi</i>
...	...
3 448 765 900 126	940



Chú ý không nhầm lẫn các ý nghĩa của kí tự * Khi bạn khai báo một con trỏ, * có tác dụng chỉ ra bạn muốn tạo ra một con trỏ:
`int *pointerTuoi;`
Còn trong printf:
`printf ("%d", *pointerTuoi);`
điều này không phải "tôi muốn tạo một con trỏ" mà là "tôi muốn giá trị của biến số mà con trỏ chỉ vào".

Tất cả những điều trên là cơ bản. Bạn phải học thuộc lòng và tất nhiên phải hiểu rõ. Đừng ngại đọc đi đọc lại nhiều lần. Đừng xấu hổ nếu không hiểu ngay được bài học khi chỉ đọc qua lần đầu tiên, có nhiều vấn đề chúng ta cần nhiều ngày để có thể hiểu rõ và đôi khi cần nhiều tháng để có thể sử dụng thành thạo. 😊

Nếu bạn có cảm giác không theo kịp, thì hãy nghĩ đến những bậc thầy trong việc lập trình: không ai trong số họ có thể hiểu rõ hoàn toàn hoạt động của con trỏ trong lần đầu tiên. Nếu có một người như vậy tồn tại, bạn hãy giới thiệu với tôi nhé. 🏠

Cách sử dụng con trỏ trong một function

Điều khá thú vị ở con trỏ là chúng ta có thể sử dụng chúng trong các function để có thể thay đổi trực tiếp giá trị của biến số trong bộ nhớ chứ không phải một bản sao như bạn đã thấy ở đoạn đầu bài học.

Vậy nó hoạt động như thế nào? Có rất nhiều cách thức để sử dụng. Đây là ví dụ đầu tiên:

C code:

```
void triplePointer(int *pointerSoHang);

int main (int argc, char *argv[ ])
{
    int soHang = 5;

    triplePointer(&soHang); // Ta gửi địa chỉ của soHang vào function
    printf ("%d", soHang); /* Ta hiển thị biến so soHang. Và function đã trực tiếp thay đổi giá trị
của biến số vì nó biết địa chỉ của biến số này */
    return 0;
}

void triplePointer(int *pointerSoHang)
{
    *pointerSoHang *= 3; // Ta x3 giá trị của số hàng được đưa vào
}
```

Console

15

Function *triplePointer* nhận vào parameter giá trị *type int ** (đó là một con trỏ chỉ vào một biến số *type int*).

Và đây là những gì diễn ra theo thứ tự, bắt đầu bởi function *main*:

1. Một biến số *soHang* được tạo ra trong *main*. Khởi tạo với giá trị 5.
2. Ta gọi function *triplePointer*. Ta gửi vào parameter địa chỉ của biến số.
3. Function *triplePointer* nhận địa chỉ là giá trị của *pointerSoHang*. Và trong function *triplePointer*, ta có một con trỏ *pointerSoHang* chứa địa chỉ của biến số *soHang*
4. lúc này, ta có một con trỏ chỉ lên biến số *soHang*, ta đã có thể thay đổi trực tiếp giá trị của biến số *soHang* trong bộ nhớ! Chỉ cần dùng **pointerSoHang* để điều chỉnh giá trị của biến số *soHang*! Ở ví dụ trên, người ta chỉ đơn giản thực hiện: nhân 3 lần giá trị của biến số *soHang*.
5. kết thúc bằng *return* trong function *main*, lúc này *soHang* đã có giá trị 15 vì function *triplePointer* đã trực tiếp thay đổi giá trị của nó.

Tất nhiên, tôi có thể thực hiện *return* để trả về giá trị như cách chúng ta đã học trong bài học về function. Nhưng điều thú vị ở đây là, bằng cách sử dụng con trỏ, chúng ta có thể thay đổi giá trị của nhiều biến số trong bộ nhớ (có nghĩa là "chúng ta có thể trả về nhiều giá trị"). Không còn giới hạn một giá trị duy nhất được trả về nữa !



Vậy *return* còn giá trị sử dụng gì khi người ta đã có thể dùng con trỏ để thay đổi giá trị ?

Điều này phụ thuộc vào bạn và chương trình bạn viết. Chúng ta cần hiểu là cách dùng *return* để trả về giá trị là một cách viết khá đẹp và được sử dụng thường xuyên trong C.

Và thường xuyên nhất, người ta dùng *return* để thông báo lỗi của chương trình: ví dụ, function trả về 1 (*true*) nếu tất cả diễn ra bình thường, và 0 (*false*) nếu có lỗi trong chương trình.

Một cách khác để sử dụng con trỏ trong function

Trong những code source mà chúng ta vừa thấy, không có con trỏ trong function *main*. Duy nhất chỉ biến số *soHang*.

Con trỏ duy nhất được sử dụng nằm trong function *triplePointer* (có type *int **)

Bạn cần biết rằng có cách viết khác cho đoạn code vừa rồi bằng cách thêm vào con trỏ trong function *main*:

C code:

```
void triplePointer(int *pointerSoHang);

int main (int argc, char *argv[ ])
{
    int soHang = 5;
    int *pointer = &soHang; // con trỏ nhận địa chỉ của biến soHang

    triplePointer (pointer); // Ta đưa con trỏ (địa chỉ của soHang) vào function
    printf ("%d", *pointer); // Ta hiển thị giá trị của soHang với *pointer

    return 0;
}

void triplePointer(int *pointerSoHang)
{
    *pointerSoHang *= 3; // Ta x3 giá trị của soHang
}
```

Hãy so sánh đoạn code source này với đoạn code source trước đó. Có một số thay đổi nhưng chúng sẽ cho ta cùng một kết quả:

Console

15

Điều cần xét đến là cách đưa địa chỉ của biến số *soHang* vào function, cách sử dụng địa chỉ của biến số *soHang*. Điều khác biệt xảy ra ở đây là cách tạo con trỏ trong function *main*.

VD trong *printf*, tôi muốn hiển thị giá trị của biến số *soHang* bằng cách viết **pointer*. Bạn cần biết rằng tôi vẫn thể viết *soHang*: kết quả sẽ giống nhau vì **pointer* và *soHang* đều có chung một giá trị trong bộ nhớ

Trong chương trình "Lớn hơn hay nhỏ hơn", chúng ta đã sử dụng con trỏ bất chấp việc biết nó là gì, trong việc sử dụng function *scanf*.

Thật ra, function này có tác dụng đọc những thông tin mà người dùng nhập vào bàn phím và gửi lại kết quả.

Để *scanf* có thể thay đổi trực tiếp giá trị của một biến số bằng cách nhập từ bàn phím, ta cần địa chỉ của biến số đó:

C code:

```
int soHang = 0;  
scanf ("%d", &soHang);
```

function làm việc với con trỏ của biến số *soHang* và có thể thay đổi trực tiếp giá trị của *soHang*.

Và như chúng ta biết, chúng ta có thể làm như sau:

C code:

```
int soHang = 0;  
int *pointer = &soHang;  
scanf ("%d", pointer);
```

Chú ý là ta không đặt kí tự **&** trước pointer trong function *scanf*. Tại đây, pointer bản thân nó đã là địa chỉ của biến số *soHang*, không cần thiết phải thêm **&** vào nữa !

Nếu bạn làm điều đó, bạn sẽ đưa cho *scanf* địa chỉ của pointer: nhưng thứ chúng ta cần là địa chỉ của *soHang*. 😊

Giải quyết vấn đề nan giải ở đầu bài?

Đã đến lúc chúng ta xem lại tâm điểm của bài học. 😊 Nếu bạn hiểu bài học này, bạn đã có thể tự giải quyết vấn đề đặt ra. Hãy thử đi! trước khi xem kết quả tôi đưa bạn: 😊

C code:

```
#include <stdio.h>
#include <stdlib.h>
void chuyenDoi(int *pointerGio, int *pointerPhut);

int main (int argc, char *argv[ ])
{
    int gio = 0, phut = 90;
    // Ta đưa vào địa chỉ của gio và phut
    chuyenDoi(&gio, &phut);
    // Lúc này, giá trị của chúng đã được thay đổi !
    printf ("%d gio và %d phut", gio, phut);
    return 0;
}

void chuyenDoi(int *pointerGio, int *pointerPhut)
{
    /*Note: đừng quên đặt dấu * ở phía trước tên của con trỏ! Bằng cách này bạn có thể thay đổi
    giá trị của biến số chứ không phải địa chỉ của nó! Hạn là bạn không muốn chia địa chỉ của nó
    đúng không? */
    *pointerGio = *pointerPhut / 60;
    *pointerPhut = *pointerPhut % 60;
}
```

Console:

1 gio và 30 phut

Không có gì khiến bạn ngạc nhiên trong đoạn code source này. Và như mọi khi, để tránh những nhầm lẫn không đáng có, tôi sẽ giải thích những gì đã diễn ra để chắc chắn rằng các bạn theo kịp tôi, vì đây là một bài học quan trọng, bạn cần cố gắng rất nhiều để hiểu, và tôi cũng cố gắng hết sức để giải thích rõ ràng giúp các bạn hiểu: 😊

1. Biến số *gio* và *phut* được khởi tạo trong function *main*.
2. Ta gửi vào function *chuyenDoi* địa chỉ của *gio* và *phut*.
3. Function *chuyenDoi* nhận địa chỉ bằng cách đưa vào các con trỏ *pointerGio* và *pointerPhut*. Bạn cần biết rằng, cách gọi tên con trỏ không quan trọng. Tôi có thể gọi là *g* và *p*, hoặc cũng có thể là *gio* và *phut*. 😊
4. function *chuyenDoi* thay đổi trực tiếp các giá trị của *gio* và *phut* trong bộ nhớ vì nó đã có địa chỉ của chúng trong các con trỏ. Và điều cần biết ở đây, tuyệt đối chấp hành, là phải đặt *** trước tên của con trỏ nếu như ta muốn thay đổi giá trị của *gio* và *phut*. Nếu ta không làm việc này, ta sẽ thay đổi địa chỉ chứa trong con trỏ, và nó chẳng giúp ta được gì. 😊



Các bạn cần lưu ý là chúng ta vẫn có thể giải quyết "vấn đề" này không qua cách sử dụng con trỏ. Điều này là chắc chắn nhưng điều đó sẽ phá vỡ luật chúng ta đã đặt ra: không sử dụng những biến số *global*. Hoặc sử dụng *printf* trong function *chuyenDoi* (nhưng ta cần một *printf* trong function *main*).

Mục đích chính của chương trình là giúp bạn có được hứng thú với việc sử dụng con trỏ. Và bạn hãy cố gắng thúc đẩy những hứng thú này ngày một nhiều hơn trong các bài học tiếp theo. 😊

Bài 3 Array

Mảng

Bài học này là phần tiếp theo của bài trước, nó giúp bạn hiểu thêm về cách sử dụng các pointer. Bạn cảm thấy có chút khó khăn trong việc sử dụng các pointers? 😊

Bạn sẽ không tránh khỏi việc dùng chúng đâu! Các pointers được sử dụng thường xuyên trong C, tôi đã nói với bạn điều này! 😊

Trong bài học này, chúng ta sẽ học cách tạo những biến số type "array" hay còn gọi là mảng. Các mảng được sử dụng thường xuyên trong C vì nó tiện lợi trong việc sắp xếp một chuỗi các giá trị.

Bài học được bắt đầu bằng vài lời giải thích về cách hoạt động của các arrays trong bộ nhớ. Tôi nhận thấy việc mở đầu với các kiến thức trong phần bộ nhớ vô cùng quan trọng: Nó giúp bạn hiểu được phương thức hoạt động. Một lập trình viên hiểu được điều họ làm, điều này sẽ đảm bảo chương trình viết ra chạy ổn định hơn, bạn nghĩ sao? 😊

Các arrays trong bộ nhớ

“Arrays là một dãy các biến số cùng type, chứa trong một vùng bộ nhớ liên tục.”

Lời giải thích trên có vẻ giống trong từ điển phải không? 😊

Rõ ràng hơn, mảng có thể chứa một số lượng lớn biến số cùng type (*long, int, char, double...*).

Mỗi mảng có một kích thước xác định. Nó có thể tạo bởi 2, 3, 10, 150, 2500 cases (ô, slots), tùy theo tùy chọn của bạn.

Biểu đồ sau là một mảng kích thước 4 ô trong bộ nhớ, nó bắt đầu từ địa chỉ 1600 :

Địa chỉ	Giá trị
1600	10
1601	23
1602	505
1603	8

Khi bạn yêu cầu tạo một mảng kích thước 4 ô trong bộ nhớ, chương trình sẽ yêu cầu hệ điều hành quyền sử dụng 4 ô bộ nhớ. 4 ô này phải nằm kế nhau, có nghĩa là ô sau sẽ kế tiếp ô trước. Giống như trên, các địa chỉ nằm nối tiếp nhau: 1600, 1601, 1602, 1603 và không có "khoảng trống" nào ở giữa.

Cuối cùng, mỗi ô trong mảng chứa một số cùng type. Nếu mảng có type int, thì mỗi ô trong mảng chứa một số type int. Không thể tạo mảng cùng lúc chứa giá trị type int và double.

Tóm lại, sau đây là những điều buộc phải ghi nhớ:

- Khi một mảng (array) được tạo ra, nó sử dụng **một vùng liên tục trong bộ nhớ**: ở đó các ô bộ nhớ sẽ nằm liên tục kế nhau.
- Tất cả các ô (case) trong mảng **phải cùng type**. Một array type int chỉ chứa các số dạng int, không thể chứa các số dạng khác.

Cách tạo một mảng (array)

Bắt đầu, chúng ta sẽ xem làm thế nào để tạo một mảng chứa 4 giá trị int:

C code:

```
int array[4] ;
```

Vậy thôi, 🤔 ta chỉ cần thêm vào trong dấu ngoặc vuông [] số lượng ô mà bạn muốn chứa trong mảng. Không có giới hạn (tùy theo dung lượng bộ nhớ của máy tính). 🌐

Vậy bây giờ, làm cách nào đưa giá trị vào mỗi case trong mảng? Rất đơn giản, chỉ cần viết `array[số_thứ_tự_của_case]`



Chú ý: một mảng bắt đầu từ số 0! Mảng chứa 4 giá trị int có các ô với số thứ tự 0, 1, 2, và 3. Không có ô số 4 trong array 4 cases! Các bạn hay nhầm lẫn ở đây, nhớ kĩ.

Nếu tôi muốn thêm vào mảng các giá trị giống như trong biểu đồ trên, tôi sẽ viết:

C code:

```
int array[4] ;  
array[0] = 10;  
array[1] = 23;  
array[2] = 505;  
array[3] = 8;
```



Vậy mối quan hệ giữa mảng và pointer là gì ?

Nếu bạn chỉ viết là array thì đó chính là pointer. Đó là một pointer chỉ vào ô đầu tiên của mảng.
Test :

C code:

```
int array[4] ;  
printf ("%d", array);
```

Kết quả, ta nhận được ô địa chỉ đầu tiên của mảng:

Console:

1600

Nếu bạn ghi thứ tự của ô trong mảng vào ngoặc vuông [], bạn sẽ nhận được giá trị của ô đó:

C code:

```
int array[4];  
printf ("%d", array[0]);
```

Console:

10

Tương tự với các ô khác. Nhắc lại là nếu bạn viết array, nó sẽ là một pointer, chúng ta có thể sử dụng kí tự * để có được giá trị của ô đầu tiên:

C code:

```
int array[4];  
printf ("%d", *array);
```

Console:

10

Và có thể nhận giá trị của ô tiếp theo với *(array+1) (địa chỉ của array+1). Cả 2 dòng code sau hoạt động tương tự nhau:

C code:

```
array[1] // Cho giá trị của ô thứ 2 (ô đầu tiên viết là [0])  
*(array+ 1) // Tương tự: cho giá trị ô thứ 2
```

Nói rõ hơn, nếu bạn viết array[0], cũng giống như bạn yêu cầu giá trị tìm thấy ở địa chỉ **array + 0** (ở ví dụ là 1600). Nếu bạn viết array[1], bạn sẽ nhận được giá trị ở địa chỉ **array + 1** (1601 trong ví dụ). Và tương tự với những trường hợp còn lại. 😊

Dynamic array(mảng động)

Ngôn ngữ C tồn tại rất nhiều versions. Version trước đây, gọi là C99, cho phép tạo các dynamic array, có nghĩa là mảng với kích thước được khai báo bởi một biến số:

C code:

```
int kíchThuoc = 5;  
int array[kíchThuoc];
```

Cách viết này không được thông dụng lắm đối với các compiler, nhiều khi chương trình chạy đến ở dòng thứ 2 sẽ dừng lại. Từ đầu đến giờ, tôi hướng dẫn bạn ngôn ngữ C89 nên chúng ta sẽ tuyệt đối không dùng dòng code thứ 2

Vậy là, các bạn không được viết một biến số đặt trong ngoặc vuông để khai báo kích thước một mảng, kể cả khi biến số này là hằng số (constant)! Một mảng phải có kích thước xác định, có nghĩa là khi khai báo bạn phải ghi rõ ràng kích thước của mảng đó bằng một con số:

C code:

```
int array[5] ;
```



Vậy tại sao lại cấm tạo một mảng với kích thước phụ thuộc vào một biến số?

Tôi bảo đảm với bạn: điều này là có thể thực hiện! ngay cả trong C89. 🤖

Nhưng để thực hiện điều này, chúng ta sẽ dùng một kỹ thuật khác (chắc chắn, đảm bảo có thể chạy được trên mọi điều kiện) gọi là phân bổ động (dynamic allocation). Chúng ta sẽ được học về sau.

Liệt kê các giá trị trong mảng (array)

Bây giờ tôi muốn hiển thị giá trị mỗi ô trong mảng. Tôi có thể sử dụng số lượng printf bằng với số ô trong mảng. Nhưng tôi phải viết nhiều lần printf, điều này thật nhàm chán (hãy tưởng tượng đến trường hợp mảng chứa 8000 giá trị) 😞

Vì vậy, tốt hơn là sử dụng vòng lặp. Và vòng lặp *for* rất tiện lợi trong việc này:

C code:

```
int main (int argc, char *argv[ ])
{
    int array[4], i = 0;
    array[0] = 10;
    array[1] = 23;
    array[2] = 505;
    array[3] = 8;
    for (i = 0 ; i < 4 ; i++)
    {
        printf ("%d\n", array[i]);
    }
    return 0;
}
```

Console:

```
10
23
505
8
```

Vòng lặp sẽ chạy dọc các ô trong mảng nhờ vào biến số *i* (các nhà lập trình thường dùng *i*, đây là một biến số khá thông dụng để chạy dọc một mảng) 😊

Cách này đặc biệt thông dụng, ta để một biến số trong dấu []. Những biến số tuyệt đối cấm sử dụng trong việc tạo các mảng (để khai báo kích thước), nhưng nó được phép sử dụng để "di chuyển" trong mảng, có nghĩa là để hiển thị các giá trị! Trong ví dụ trên, tôi đặt biến số *i*, nó sẽ

tăng dần từ 0, 1, 2 rồi 3. Như vậy, nó sẽ hiển thị các giá trị của array[0], array[1], array[2] và array[3]! 😊



Cần chú ý là không nên hiển thị giá trị của array[4]! Một array 4 ô chỉ có số thứ tự là 0, 1, 2, 3. Nếu bạn thử hiển thị giá trị của array[4], nó sẽ hiển thị một số không xác định, đây sẽ là một lỗi khá đẹp. Hệ điều hành sẽ dừng chương trình lại do nó cố ý xâm nhập vào một địa chỉ không cho phép.

Khởi tạo các giá trị trong một mảng

Bạn đã biết cách di chuyển trong mảng, điều này có nghĩa bạn có thể khởi tạo mảng với giá trị 0 ở tất cả các ô bằng việc sử dụng vòng lặp!

Bạn đã đạt được trình độ cần thiết có thể thực hiện điều này 😊:

C code:

```
int main (int argc, char *argv[ ])
{
    int array[4], i = 0;
    // Khởi tạo các giá trị trong array
    for (i = 0 ; i < 4 ; i++)
    {
        array[i] = 0;
    }
    // Hiển thị các giá trị để kiểm tra
    for (i = 0 ; i < 4 ; i++)
    {
        printf ("%d\n", array[i]);
    }
    return 0;
}
```

Console:

```
0
0
0
0
```

Một cách khác để khởi tạo giá trị

Bạn cần biết là còn một cách khác để khởi tạo giá trị trong mảng khá thủ công trong C. Bằng cách viết từng giá trị trong ngoặc nhọn { }, cách nhau bởi dấu phẩy “,” :

array[4] = { giaTri1, giaTri2, giaTri3, giaTri4};

C code:

```
int main (int argc, char *argv[ ])
{
    int array[4] = {0, 0, 0, 0}, i = 0;
    for (i = 0 ; i < 4 ; i++)
    {
        printf ("%d\n", array[i]);
    }
    return 0;
}
```

Console:

```
0
0
0
0
```

Mặt khác, lợi ích của cách làm này là, bạn chỉ cần khai báo giá trị những ô đầu tiên, những ô còn lại sẽ tự động nhận giá trị 0:

Cụ thể, nếu tôi viết:

C code:

```
int array[4] = {10, 23}; // Gia tri nhap vao: 10, 23, 0, 0
```

Ô đầu tiên nhận giá trị 10, ô thứ 2 nhận giá trị 23, các ô còn lại nhận giá trị 0.

Vậy làm cách nào để khai báo tất cả các ô với giá trị 0? Bạn chỉ cần khai báo ô đầu tiên giá trị 0, sau đó các ô còn lại cũng sẽ nhận giá trị 0. 😊

C code:

```
int array[4] = {0} // Khai bao array voi tat ca cac o gia tri 0.
```

Kỹ thuật này có thể hoạt động với bất kì kích thước nào (có thể thực hiện trên array 100 ô và hơn nữa) 😊



Chú ý, thường gặp lỗi

```
int array[4] = {1};
```

Dòng code này sẽ thêm vào các giá trị sau: 1, 0, 0, 0. Nhiều bạn nghĩ rằng dòng code trên sẽ khởi tạo mảng với tất cả các giá trị là 1, nhưng không phải vậy, để làm điều này các bạn cần sử dụng vòng lặp.

Tạo một function để liệt kê các giá trị trong mảng

Bạn đã biết cách hiển thị nội dung của một mảng.

Vậy tại sao bạn không thử viết một chương trình để thực hiện điều này? Như thế bạn có thể tìm hiểu cách đưa một array vào function. 🤔

Chúng ta sẽ cần gửi hai parameter vào function: đầu tiên là array (địa chỉ của array) và thứ hai sẽ là kích thước của nó! Và function của bạn có khả năng hoạt động với bất kì array nào khác đưa vào. Chúng ta sẽ cần một biến số *kichThuocArray*.

Bạn biết rằng array có thể xem như là một pointer. Vì vậy, chúng ta sẽ đưa array vào function giống như thực hiện với pointer:

C code:

```
// Prototype của function hiển thị
void hienThi (int *array, int kichThuocArray);

int main (int argc, char *argv[ ])
{
    int array[4] = { 10, 15, 3 };

    // Chúng ta hiển thị nội dung của array
    hienThi (array, 4);

    return 0;
}

void hienThi (int *array, int kichThuocArray)
{
    int i;

    for (i = 0 ; i < kichThuocArray ; i++)
    {
        printf ("%d\n", array[i]);
    }
}
```

Console

```
10
15
3
0
```

Function này không khác nhiều so với function chúng ta được học ở bài trước. Chúng ta sẽ đưa vào một parameter pointer type int (array), sau đó là kích thước của array (rất quan trọng để biết khi nào vòng lặp dừng lại!). Nội dung của array sẽ được hiển thị qua function nhờ vào một vòng lặp.

Cần ghi thêm là còn một cách khác để đưa mảng vào function:

C code:

```
void hienThi (int array[ ], int kíchThuocArray)
```

hoạt động tương tự như trên, việc sử dụng những dấu ngoặc vuông cho phép người đọc code hiểu rõ: function cần một mảng. Có thể hạn chế được nhầm lẫn là function cần một con trỏ hoặc biến cơ bản nào đó. 😊

Tôi thường sử dụng cách viết thứ hai để đưa mảng vào function, các bạn nên thực hiện giống tôi. Và trong cách viết này chúng ta không cần khai báo kích thước trong ngoặc vuông.

Một vài bài tập thực hành!

Tôi lúc nào cũng có rất nhiều bài tập cho bạn luyện tập ! 🧐

Các bạn nên tự viết các function làm việc với array.

Tôi chỉ đưa đề bài cho các bạn thực hành, về code, các bạn sẽ tự viết lấy. Hãy đặt câu hỏi, nếu có thắc mắc. 😊

Bài tập 1

Tạo một function tongArray để tính tổng các giá trị chứa trong nó (sử dụng return để trả về giá trị). Và để giúp bạn hiểu rõ hơn, đây là prototype của function cần viết:

C code:

```
int tongArray (int array[ ], int kíchthuocArray);
```

Bài tập 2

Tạo một function trungBinhArray để tính trung bình các giá trị chứa trong nó.

Prototype:

C code:

```
double trungBinhArray (int array[ ], int kíchThuocArray);
```

Bài tập 3

Tạo một function copyArray để chép nội dung array này sang một array khác.

Prototype:

C code:

```
void copyArray (int array1[ ], int array2[ ], int kíchThuoc);
```

Bài tập 4

Viết một function `maximumArray` có nhiệm vụ so sánh tất cả các giá trị chứa bên trong array với `giaTriMax`. Nếu có giá trị lớn hơn biến số `giaTriMax` đưa vào, nó sẽ chuyển thành 0.

Prototype:

C code:

```
void maximumArray (int array[ ], int kíchThuoc, int giaTriMax);
```

VD: array {1,5,7,8,5,2,3} và max=5, sẽ chuyển thành {1,5,0,0,5,2,3}.

Bài tập 5

Bài tập này khó hơn hẳn các bài tập kia nhưng bạn hoàn toàn có khả năng thực hiện. 😊

Hãy viết một function `sapXepArray` sắp xếp lại các giá trị bên trong theo thứ tự tăng dần.

C code:

```
void sapXepArray (int array[ ], int kíchThuoc);
```

VD: array {1,5,7,8,5,2,3} sẽ chuyển thành {1,2,3,5,5,7,8}.



Hãy viết trong file `array.c` chứa tất cả các functions cần thiết và file `array.h` chứa các prototypes của các functions đó. 😊

Nào bắt đầu làm việc thôi ! 😊

Nếu bạn qua được bài học về pointer thì các bài khác các bạn đều có thể vượt qua. Tôi không nghĩ là bài học này sẽ gây khó khăn cho bạn. 😊

Bạn nên nhớ hai điều quan trọng sau:

- **Đừng bao giờ quên** một array bắt đầu bằng số thứ tự 0 (**không phải 1**)
- Khi bạn đưa một array vào function, luôn gửi kèm theo kích thước của array.

Và tôi sẽ cho bạn một tin vui, **bạn đã có đủ trình độ để có thể làm việc với các chuỗi kí tự.** 😊
Bạn có thể đưa một chuỗi kí tự vào bộ nhớ, ví dụ như việc yêu cầu họ tên người dùng.

Và trong bài học sau chúng ta sẽ học cách làm việc với các chuỗi kí tự ! 🌐

Bài 4: String

Chuỗi (mảng ký tự)

String - Chuỗi (mảng ký tự) là một thuật ngữ tin học chính xác dùng để chỉ *một dãy các ký tự*, đơn giản là như vậy! Một chuỗi ký tự được lưu trong bộ nhớ máy tính dưới dạng biến số. Nhờ vậy ta có thể lưu trữ tên của người dùng.

Và như bạn đã biết, máy tính chỉ có thể nhớ được những con số. Máy tính không hiểu chữ cái là gì. Vậy làm thế nào máy tính có thể nhớ được những dãy ký tự? 🤔

Biến kiểu char:

Trong phần này, chúng ta sẽ đặc biệt quan tâm đến biến kiểu char. Bạn có nhớ rằng biến kiểu char cho phép chứa các con số trong khoảng -128 và 127. 🤖



Liệu biến kiểu char có cho phép chứa những con số? Bạn cần biết rằng trong C người ta *rất hiếm khi* sử dụng chúng để làm điều đó. Bình thường, ngay cả đối với những con số thật sự nhỏ, người ta vẫn dùng int để lưu lại. Hẳn rằng tôi đã sử dụng nhiều bộ nhớ hơn so với char nhưng trong thời đại ngày nay, vấn đề bộ nhớ không còn đáng lo nữa. 😎

Thật ra biến kiểu char được tạo ra để chứa ... một ký tự! Chú ý là tôi nói rõ ràng « một ký tự ». Bộ nhớ máy tính chỉ có thể chứa những con số nên người ta đã tạo ra một bảng chuyển đổi giữa số và ký tự. Lấy ví dụ con số 65 sẽ được chuyển đổi thành chữ cái A. 🧠

Ngôn ngữ C cho phép chúng ta chuyển đổi dễ dàng giữa số và chữ cái tương ứng. Để nhận được một số ứng với chữ cái, người ta chỉ cần viết chúng giữa những dấu móc đơn, như sau: 'A'. Qua quá trình compilation, 'A' sẽ được thay thế bằng con số tương ứng.

Test thử nào:

C code:

```
int main (int argc, char *argv[ ])
{
    char letter = 'A';
    printf ("%ld\n", letter);
    return 0;
}
```

Console:

65

Chúng ta thấy ngay rằng chữ A viết hoa đã được thay bằng số 65. Tương tự như vậy B thay bằng 66, C bằng 67... Test thử với những chữ cái viết thường, giá trị của những chữ cái sẽ thay đổi. Và chữ 'a' không giống như 'A', máy tính phân biệt chữ cái viết hoa và viết thường.

Hầu hết các chữ cái thông thường được code giữa 0 và 127. Bảng chuyển đổi giữa số và chữ cái có tên là ASCII (cách đọc "át-xơ-ki").

Trang web **AsciiTable.com** là địa chỉ khá nổi tiếng để tìm thấy bảng chuyển đổi này nhưng nó không phải là duy nhất, chúng ta có thể tìm thấy nó trên Wikipédia và một số trang web khác.

Cách hiển thị một ký tự:

Function printf, sẽ tiếp tục làm chúng ta ngạc nhiên, nó có khả năng hiển thị một chữ cái. Để làm điều đó, chúng ta cần phải sử dụng kí hiệu %c (c viết tắt của character):

C code:

```
int main (int argc, char *argv[ ])
{
    char letter = 'A';
    printf ("%c\n", letter);
    return 0;
}
```

Console:

A

Chúc mừng bạn đã biết cách hiển thị một chữ cái!

Chúng ta cũng có thể yêu cầu người dùng nhập vào một chữ cái bằng cách sử dụng %c trong scanf:

C code:

```
int main (int argc, char *argv[ ])
{
    char letter = 0;
    scanf ("%c", &letter);
    printf ("%c\n", letter);
    return 0;
}
```

Nếu tôi nhập vào đây chữ cái B, máy tính sẽ hiển thị:

Console:

```
B  
B
```

(chữ B thứ nhất là do tôi nhập từ bàn phím và chữ B thứ 2 do printf hiển thị)

Và đó là những gì bạn cần biết về biến kiểu char.



Cần nhớ:

- Type char cho phép nhập vào những giá trị từ -128 đến 127, unsigned char từ 0 đến 255.
- Có những bảng chuyển đổi cho phép máy tính chuyển từ chữ cái thành số và ngược lại.
- Chúng ta có thể sử dụng type char để chứa MỘT chữ cái.
- 'A' được chương trình dịch chuyển thành một số tương ứng (thông thường là 65). Chúng ta dùng những dấu ngoặc đơn '...' để có được giá trị của một chữ cái.

String - Chuỗi ký tự là một mảng các giá trị char!

Tiêu đề ở trên nói ra tất cả những gì tôi sẽ nói với bạn ở đây.

Một chuỗi ký tự (string) chỉ là một mảng các giá trị biến kiểu char. Đơn giản là một mảng, không hơn.

Nếu tôi tạo một mảng:

C code:

```
char string[5];
```

... nếu tôi đặt vào string[0] chữ cái 'H', string[1] chữ cái 'e'... tôi sẽ tạo được một dãy ký tự , hoặc là một văn bản (text)

Đây là biểu đồ về cách bộ nhớ máy tính lưu trữ (Bạn cần chú ý kỹ phần này vì phần sau bài học sẽ phức tạp hơn rất nhiều):

Address	Value
18000	'H'
18001	'e'
18002	'l'
18003	'l'
18004	'o'

Giống như bạn thấy, đó là một bảng chứa 5 ô bộ nhớ để trình bày chữ « Hello ». Để có được giá trị số, tôi đặt chúng vào giữa những dấu ngoặc đơn, để nói với máy tính rằng đó là một số chứ không phải ký tự. Và trong thực tế, trong bộ nhớ máy tính sẽ lưu lại giá trị số tương ứng với những chữ cái được lưu.

Vâng, nhưng bạn cần lưu ý, một dãy ký tự không chỉ chứa các chữ cái! Biểu đồ trên chưa hoàn chỉnh.

Một dãy ký tự bắt buộc phải chứa một ký tự đặc biệt ở cuối cùng, gọi là « ký tự kết thúc chuỗi ». Ký tự đó được viết là `'\0'`.

❓ Tại sao ta cần phải kết thúc chuỗi bằng một `'\0'` ?

Đơn giản là vì máy tính cần biết khi nào kết thúc một chuỗi ký tự. Ký tự `'\0'` cho phép giải thích: « Dừng lại, không còn gì để đọc tiếp ở đây nữa! ».

Qua đó, để lưu từ « Hello » (gồm 5 chữ cái) vào bộ nhớ, ta cần dùng một string 6 char chứ không phải là một string 5 char!

Mỗi khi bạn tạo một chuỗi ký tự, bạn cần phải nghĩ đến trước đó vị trí dự phòng để đặt ký tự '\0', đây là điều bắt buộc!

Lỗi thường gặp trong lập trình ngôn ngữ C là việc bạn quên đi ký tự này, chính tôi cũng đã mắc lỗi này khá là nhiều lần

Để hiểu rõ hơn, đây chính là biểu đồ chính xác về từ « Hello » chứa trong bộ nhớ:

Address	Value
18000	'H'
18001	'e'
18002	'l'
18003	'l'
18004	'o'
18005	'\0'

Bạn thấy đây, chuỗi ký tự chiếm 6 ô trong bộ nhớ chứ không phải 5.

Chuỗi ký tự sẽ kết thúc bằng '\0', ký tự kết thúc chuỗi cho phép máy tính biết rằng chuỗi ký tự sẽ kết thúc ở đây.

Bạn sẽ thấy rằng ký tự '\0' sẽ là một lợi thế cho chúng ta. Nhờ nó mà bạn có thể biết được độ dài của chuỗi ký tự, vì nó nằm ở vị trí kết thúc của chuỗi. Bạn có thể đưa chuỗi ký tự vào một function mà không cần phải thêm vào một biến số chỉ độ dài của chuỗi.

Việc này chỉ có tác dụng đối với những chuỗi ký tự (có nghĩa là với type char*, hoặc char[]). Đối với những dạng mảng khác, các bạn bắt buộc phải lưu lại độ lớn của chuỗi ở đâu đó.

Khai báo và khởi tạo chuỗi ký tự

Nếu ta muốn khai báo một từ « Hello », ta có thể sử dụng phương pháp buồn chán sau:

C code:

```
char string[6]; // mảng string gom 6 char de luu tru H-e-l-l-o va \0
string[0] = 'H';
string[1] = 'e';
string[2] = 'l';
string[3] = 'l';
string[4] = 'o';
string[5] = '\0';
```

Phương pháp này hoạt động, bạn có thể kiểm tra lại bằng cách sử dụng printf.

Ah, tôi quên mất ở printf: bạn cần phải biết thêm một ký tự đặc biệt khác đó là %s (s như là một string).

Và đây là đoạn code hoàn chỉnh tạo và hiển thị từ « Hello »:

C code:

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[ ])
{
    char string[6]; // mảng string gom 6 char de luu tru H-e-l-l-o và \0
    // Khởi tạo chuỗi ký tự (Ta viết từng ký tự vào bộ nhớ)
    string[0] = 'H';
    string[1] = 'e';
    string[2] = 'l';
    string[3] = 'l';
    string[4] = 'o';
    string[5] = '\0';
    // Hiển thị chuỗi ký tự bằng printf nhớ %s
    printf ("%s", string);
    return 0;
}
```

Console:

Hello

Tất cả những dòng code trên chỉ để tạo và hiển thị mỗi từ « Hello ». Khá là mệt mỏi khi phải lặp đi lặp lại việc khai báo riêng biệt từng chữ cái trong mảng string. Để khởi tạo một chuỗi ký tự, may mắn thay ta còn một phương pháp khác đơn giản hơn:

C code:

```
int main (int argc, char *argv[ ])
{
    char string[ ] = "Hello"; // Do dài của chuỗi ký tự được máy tính tự động tính toán.
    printf ("%s", string);
    return 0;
}
```

Console:

Hello

Ở dòng đầu tiên trong main, tôi tạo một biến số dạng char[]. Tôi cũng có thể viết là char*, kết quả sẽ vẫn như nhau.

Sau đó bạn sẽ điền trong ngoặc kép từ bạn muốn lưu lại, bộ dịch của C sẽ tính toán tự động độ dài cần thiết. Có nghĩa là nó sẽ đếm số chữ cái và thêm vào ký tự '\0'. Sau đó nó sẽ điền từng chữ cái của từ « Hello » vào bộ nhớ giống như cách đầu tiên chúng ta làm ở trên.

Tóm lại, đơn giản và dễ sử dụng.

Khuyết điểm: Cách này chỉ hoạt động khi khởi tạo chuỗi, bạn sẽ không thể sử dụng tiếp ở những phần sau của chương trình, bạn không thể viết:

C code:

```
string = "Hello";
```

Sau phần khởi tạo, bạn chỉ có thể lưu một từ bằng cách viết riêng biệt từng chữ cái vào bộ nhớ.

Cách lưu trữ một chữ cái bằng scanf:

Các bạn có thể nhờ người sử dụng nhập vào một từ bằng scanf, bằng cách sử dụng tiếp ký tự %s.

Vấn đề duy nhất xảy ra, bạn sẽ không biết có bao nhiêu chữ cái người dùng sẽ nhập vào. Nếu bạn yêu cầu nhập tên, có thể người dùng chỉ nhập vào Nhu (3 chữ cái), và đôi lúc người đó sẽ nhập vào Superman (nhiều chữ cái hơn).

Để làm điều này, 36 ký của tên từ là không đủ cho bạn. Đôi khi, chúng ta cần khai báo một mảng char lớn hơn, đủ lớn để có thể chứa tên của người dùng. Chúng ta sẽ tạo một char[100] để chứa tên người dùng. Việc này khiến bạn có cảm giác chúng ta đang lãng phí bộ nhớ máy tính, nhưng vấn đề về bộ nhớ này không thật sự đáng để lưu tâm (sẽ có những chương trình làm lãng phí bộ nhớ máy tính hơn rất nhiều lần như thế này, từ từ bạn sẽ thấy).

C code:

```
int main (int argc, char *argv[ ])
{
    char ten[100];
    printf ("E ku, may ten gi vay? ");
    scanf ("%s", ten);
    printf ("Hello %s, tao rat vui vi duoc gap may!", ten);
    return 0;
}
```

Console:

```
E ku, may ten gi vay? M0N1M
Hello M0N1M, tao rat vui vi duoc gap may!"
```

Và đây là phần lớn những gì phải làm để yêu cầu người dùng nhập vào một từ.

Các thao tác sử dụng trên chuỗi ký tự:

Những chuỗi ký tự sẽ được sử dụng thường xuyên. Tất cả những câu, những từ hiển thị trên màn hình máy tính bạn thấy đều được tạo bởi các array type char trong bộ nhớ máy tính, chúng hoạt động theo cách tôi vừa hướng dẫn ở trên.

Tôi sẽ giới thiệu cho bạn một số thao tác để tùy chỉnh những chuỗi ký tự, người ta đã tạo trong thư viện string.h những function cần thiết.

Tôi sẽ không hướng dẫn bạn tất cả trong bài này, sẽ rất dài và có một số function thật sự không cần thiết.

Tôi sẽ hướng dẫn bạn những cái cần thiết đủ dùng trong thời điểm hiện tại.

Hãy nhớ khai báo thư viện string.h

Mặc dù điều này là hiển nhiên nhưng tôi muốn bạn xác định rõ: khi bạn sẽ phải sử dụng một thư viện mới (string.h), bạn cần phải khai báo ở đầu file .c mà bạn cần dùng đến:

```
#include <string.h>
```

Nếu bạn không làm việc này, máy tính sẽ không biết được những function mà tôi sắp hướng dẫn cho bạn, vì chương trình không có được những prototypes cần thiết, và việc dịch chương trình sẽ không hoàn thành được.

Tóm lại, bạn đừng quên phải khai báo thư viện này mỗi khi cần dùng đến những function thao tác trên chuỗi.

Strlen: tính độ dài một chuỗi

Strlen là một function dùng để tính toán độ dài một chuỗi ký tự (không tính ký tự ‘\0’).

Bạn chỉ cần đưa vào nó parameter: chuỗi ký tự của bạn! Function này sẽ trả về độ dài của nó.

Bây giờ bạn đã biết prototype là gì vì vậy tôi sẽ đưa bạn prototype của function tôi sắp hướng dẫn. Những người lập trình xem prototype giống như “hướng dẫn sử dụng trước khi dùng” của function (và những dòng chú thích bên cạnh sẽ không bao giờ thừa).

Prototype đây:

```
size_t strlen(const char* string);
```

⚠ size_t là một type đặc biệt, nó cho biết function sẽ trả về một số tương ứng với một kích thước nào đó. Đây không phải là một type cơ bản như int, long hay char, đây là một type được “chế” thêm. Chúng ta sẽ học cách tạo ra những type mới ở những bài học sau.

Trong thời điểm hiện tại, chúng ta tạm thời hài lòng với giá trị trả về của strlen được lưu lại trong một biến số type long (máy tính sẽ tự động chuyển **size_t** thành **long**). Để rõ ràng, chính xác, bạn cần lưu lại kết quả trong một biến số type size_t, nhưng thực tế thì một biến số type long là đủ.

Function này có parameter là một type const char. Const (có nghĩa là constant, bạn nhớ chứ?) có tác dụng ngăn không cho strlen thay đổi chuỗi ký tự của bạn. Khi bạn thấy một const, bạn biết rằng giá trị của biến số đó sẽ không thể thay đổi, chỉ có thể đọc giá trị của nó.

Test thử function strlen:

C code:

```
int main (int argc, char *argv[ ])
{
    char string[ ] = "Xinchao";
    long doDaiChuoi = 0;
    // giá trị do dài của chuỗi sẽ được lưu lại trong biến số doDaiChuoi
    doDaiChuoi = strlen (string);
    // hiển thị do dài chuỗi
    printf ("Chuoi %s có độ dài %ld kí tự", string, doDaiChuoi);
    return 0;
}
```

Console:

Chuoi Xinchao có độ dài 7 kí tự

Function strlen này được viết dễ dàng bằng cách sử dụng vòng lặp trên mảng kiểu char, và nó sẽ dừng lại khi gặp ký tự kết thúc chuỗi ‘\0’. Nó sẽ lần lượt truy cập vào các ô có chứa từng ký tự của chuỗi, số lượt truy cập sẽ tăng dần sau mỗi vòng lặp, và kết quả sẽ trả về giá trị cuối cùng của những lượt truy cập này.

Tôi sẽ viết function strlen của riêng mình. Việc này sẽ giúp bạn hiểu rõ hơn nó hoạt động như thế nào:

C code:

```
long doDaiChuoai(const char* string);

int main (int argc, char *argv[ ])
{
    char string[ ] = "Hello";
    long doDai = 0;
    doDai = doDaiChuoai(string);
    printf ("chuoi %s co do dai %ld ki tu", string, doDai);
    return 0;
}

long doDaiChuoai (const char* string)
{
    long soLuongKiTu = 0;
    char kiTuHienTai = 0;
    do
    {
        kiTuHienTai = string[soLuongKiTu];
        soLuongKiTu++;
    }
    while (kiTuHienTai != '\0'); // Vòng lặp tiếp tục nếu ki tu hiện tại không phải là \0
    soLuongKiTu--; // Do dài chuỗi giảm đi 1 vì ta không tính \0

    return soLuongKiTu;
}
```

Giải thích

1. Function *doDaiChuoai* tạo vòng lặp trên mảng string. Nó sẽ lưu lại từng ký tự trong biến *kiTuHienTai*. Khi *kiTuHienTai* tiến đến \0, vòng lặp sẽ dừng lại.
2. Tại mỗi vòng lặp, độ lớn sẽ tăng lên 1 sau mỗi lần truy cập ô chứa ký tự.
3. Khi kết thúc vòng lặp, số lượng ký tự sẽ bớt đi 1. Điều này có nghĩa là ta không tính ký tự kết thúc chuỗi '\0'.
4. Cuối cùng, kết quả sẽ được trả về *soLuongKiTu*.
5. Trò chơi kết thúc.

Strcpy: sao chép chuỗi này vào chuỗi khác

Function strcpy(có thể hiểu là « string copy ») cho phép sao chép một chuỗi ký tự này đặt vào trong một chuỗi ký tự khác.

Prototype của nó là:

```
char* strcpy(char* copyString, const char* stringCopy);
```

function này nhận 2 parameter:

copyString: là một pointer char* (mảng char). Chuỗi ký tự sẽ được chép vào trong mảng này.

stringCopy: là một pointer của một mảng char khác. Chuỗi ký tự này sẽ được dùng để chép vào copyString.

Function trả về pointer của copyString cũng không hữu dụng lắm. Thực tế, ta không cần sử dụng kết quả function này trả về. Cùng test thử nhé.

C code:

```
int main (int argc, char *argv[ ])
{
    /* Chung ta khai bao bien "string" kieu char trong do co chua 1 chuoi ky tu va mot bien "copy"
    voi kich thước 100 ky tu de bao dam co du cho trong */

    char string[ ] = "Text", copy[100] = {0};

    strcpy(copy, string); // Chung ta se sao chep nhung ky tu tu "string" sang "copy"
    // Neu khong co gi sai sot thi "copy" bay gio se giống như "string"

    printf ("string is : %s\n", string);
    printf ("copy is : %s\n", copy);
    return 0;
}
```

Console:

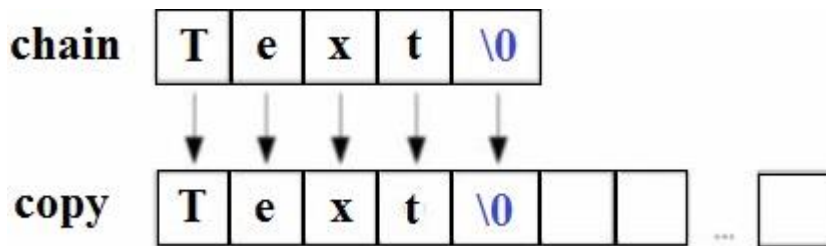
```
string is : Text
copy is : Text
```

Kết quả của string là « Text ». Điều đó là bình thường.

Nhưng, biến số copy cũng có kết quả tương tự, ban đầu biến số này không có giá trị nào, sau đó nó nhận nội dung của string. Vậy, string được sao chép lại vào trong copy.

Bạn cần chắc rằng độ lớn của chuỗi copy có đủ để nhận nội dung của string. Nếu trong ví dụ ở trên, tôi khai báo copy[4] (sẽ không đủ để chứa ký tự \0), function strcpy sẽ vượt giới hạn bộ nhớ, điều này sẽ làm chương trình bạn dừng lại. Bạn cần tránh điều này.

Biểu đồ sẽ như sau:



Mỗi ký tự trong string sẽ được điền vào copy.

Phần sau của copy còn nhiều ô nhớ không dùng đến, vì ở trên, tôi khai báo copy có độ dài là 100, nhưng trong ví dụ này, 5 là đủ để sử dụng. Lợi ích của việc tạo ra một mảng lớn hơn là để có thể sử dụng cho nhiều trường hợp khác nhau, có thể có một số chuỗi có độ dài lớn hơn trong phần sau của chương trình.

strcat: ghép nối 2 chuỗi

function này có tác dụng thêm nội dung một chuỗi phía sau một chuỗi khác. Gọi là concatenation. (sự **xâu chuỗi**)

Nếu ta có:

```
string1 = "Hello "
```

```
string2 = "M0N1M"
```

Nếu tôi nối string2 vào string1, string1 sẽ thành « Hello M0N1M ». Còn

string2 sẽ không thay đổi, string2 vẫn luôn là « M0N1M ». Chỉ mỗi string1 thay đổi.

Đó là cách strcat hoạt động, và đây là prototype của nó:

```
char* strcat(char* string1, const char* string2);
```

Như bạn thấy, string2 không thể thay đổi vì nó được định nghĩa là một constant trong prototype của function.

Function trả về pointer của string1, giống như strcpy, không có giá trị sử dụng nhiều nên ta có thể không cần quan tâm đến kết quả nó trả về.

Function thêm vào string1 nội dung của string2. Bạn có thể hiểu rõ hơn qua đoạn code sau:

C code:

```
int main (int argc, char *argv[ ])
{
    /* Chung ta se tao ra 2 mang ky tu, nho rang string1 phai du lon de chua
    duoc nhung ky tu cua string2. Neu khong chương trình se bao loi. */

    char string1[100] = "Hello ", string2[ ] = "M0N1M";

    strcat (string1, string2); // Nhung ky tu cua string2 se duoc noi tiep vao string1
    // Neu moi thu dien ra tot dep thi ket qua string1 se la "Hello M0N1M"

    printf ("string1 is : %s\n", string1);
    // string2 van khong bi thay doi :

    printf ("string2 is always : %s\n", string2);
    return 0;
}
```

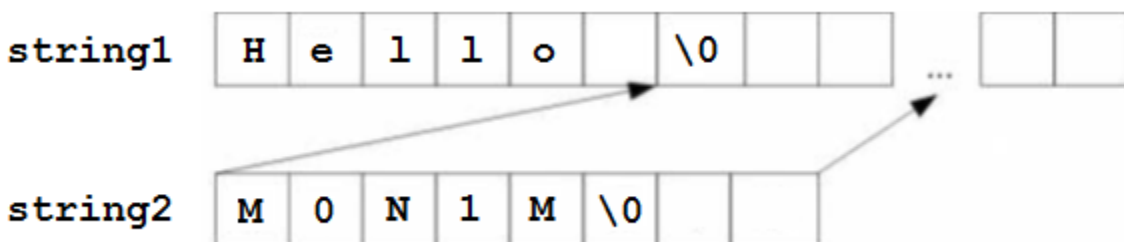
Console:

```
string1 is : Hello M0N1M
string2 is always : M0N1M
```

Cần chắc chắn là string1 phải đủ lớn để chứa thêm nội dung của string2, nếu không bạn sẽ vượt qua giới hạn bộ nhớ cho phép, điều đó sẽ khiến chương trình dừng lại.

Vì vậy tôi khai báo string1 với độ lớn là 100. Trong string2, tôi để máy tính tự tính độ lớn của chuỗi (tôi không cần phải suy nghĩ nhiều về điều này) vì chuỗi này không bị thay đổi. Ta không cần thiết phải khai báo.

Biểu đồ sẽ như sau:



Mảng của string2 sẽ được thêm vào phía sau của string1 (nó sẽ chiếm thêm một vài ô bộ nhớ)

Ký tự '\0' của chuỗi ở string1 sẽ bị xóa đi (hay được thay thế bằng M của M0N1M). Thực tế, không thể điền '\0' vào giữa chuỗi nếu không nó sẽ cắt chuỗi ra làm 2 phần! ta chỉ đặt '\0' vào vị trí cuối của chuỗi, khi chuỗi kết thúc.

Strcmp: so sánh độ dài 2 chuỗi

Function Strcmp dùng để so sánh độ dài 2 chuỗi với nhau.

Và đây là prototype:

C code:

```
int strcmp(const char* string1, const char* string2);
```

Những biến số của string1 và string2 được so sánh với nhau. Như bạn thấy, không có chuỗi nào bị thay đổi vì chúng được khai báo như những constants.

Kết quả trả về của function này cần được giữ lại.

Thực tế, strcmp trả về:

- Giá trị 0 nếu hai chuỗi giống nhau.
- Một giá trị khác 0 (lớn hơn hay nhỏ hơn 0) nếu hai chuỗi khác nhau.

Về logic, function trả về 1 nếu 2 chuỗi bằng nhau để nói là « TRUE » thì hợp lí hơn (những boolean). Nhưng function này không do tôi viết ra...

Nói rõ hơn, function sẽ so sánh giá trị của từng ký tự của mỗi chuỗi với nhau. Nếu tất cả những ký tự giống nhau, nó sẽ trả về 0.

Nếu các ký tự của string1 lớn hơn string2, nó sẽ trả về một số dương. Ngược lại, function sẽ trả về một số âm.

Trong thực tiễn, người ta thường dùng strcmp để so sánh 2 chuỗi nếu chúng giống nhau.

Đây là đoạn mã để test.

C code:

```
int main (int argc, char *argv[ ])
{
    char string1[ ] = "Text for test", string2[ ] = "Text for test";
    if (strcmp(string1, string2) == 0) // Neu 2 chuoi giống nhau
    {
        printf ("Hai chuoi giống nhau!\n");
    }
    else
    {
        printf ("Hai chuoi khác nhau!\n");
    }
    return 0;
}
```

Console:

```
Hai chuoi giống nhau!
```

Nếu hai chuỗi giống nhau, kết quả sẽ trả về 0.

Lưu ý thêm là tôi có thể lưu kết quả của strcmp lại bằng một biến số kiểu int. Tuy nhiên chúng ta không bắt buộc phải làm điều này, bạn có thể dùng trực tiếp if như tôi đã làm.

Chúng ta sẽ không nói nhiều hơn về function này. Nó khá đơn giản để sử dụng, và điều bạn cần nhớ đó là giá trị 0 có nghĩa là « giống nhau » và một giá trị khác 0 có nghĩa là « khác nhau ».

Đây chính là nguyên nhân lỗi thường xuất hiện khi sử dụng function này.

Strchr: tìm kiếm một ký tự

Chức năng của function strchr là tìm kiếm một ký tự trong chuỗi.

Prototype của function strchr đây:

```
char* strchr(const char* string, int characterSearch);
```

function nhận 2 parameters:

- *string*: chúng ta sẽ tìm kiếm ký tự trong biến này.
- *characterSearch*: ký tự bạn muốn tìm kiếm trong biến string.

Bạn thấy rằng characterSearch là biến kiểu int chứ không phải char. Điều này cũng bình thường thôi vì về cơ bản, ký tự chính là số.

Tuy nhiên, người ta thường dùng char hơn là int để chứa một ký tự trong bộ nhớ.

Function sẽ trả về pointer của chữ cái đầu tiên tìm thấy, có nghĩa là trả về địa chỉ của ký tự đó trong bộ nhớ. Kết quả sẽ trả về NULL nếu không tìm thấy ký tự bạn muốn tìm.

Trong ví dụ sau, tôi sẽ lưu pointer này trong subString:

C code:

```
int main (int argc, char *argv[ ])
{
    char string[ ] = "Text for test", *subString = NULL;
    subString = strchr(string, 'f');
    if (subString != NULL) // NULL là ko tìm thấy, vậy dòng này nghĩa là “neu ta tìm dc gì đó”
    {
        printf ("Bat dau tu ky tu dau tien duoc tim thay, string duoc in ra la : %s \n", subString);
    }
    return 0;
}
```

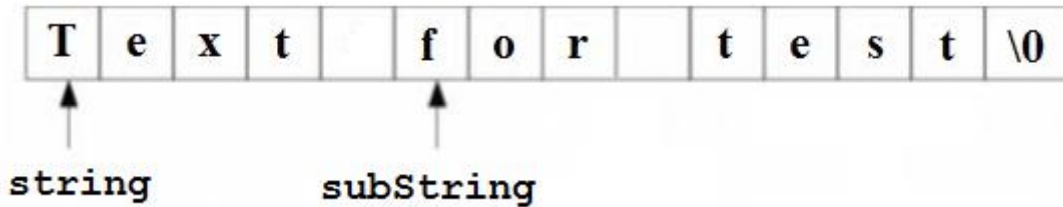
Console:

```
Bat dau tu ky tu dau tien duoc tim thay, string duoc in ra la : for test
```

Bạn hiểu điều gì đã xảy ra ở đây không? Có một sự khác biệt nhỏ ở đây .

Thực tế, subString cũng là một pointer như string. Khác biệt ở chỗ string chỉ vào chữ cái đầu tiên (T viết hoa), còn subString chỉ vào chữ 'f' đầu tiên được tìm thấy trong string.

Biểu đồ sau chỉ ra cho bạn thấy nó hoạt động như thế nào :



String bắt đầu ở ký tự đầu ('T' viết hoa) còn subString thì chỉ vào ký tự 'f'

Khi tôi viết một printf lên subString, thông thường nó sẽ hiển thị kết quả là «for test». Function printf sẽ hiển thị tất cả những chữ cái nó gặp ('f', 'o', 'r', ' ', 't', 'e', 's', 't') đến khi gặp ký tự \0 để thông báo là string kết thúc ở đây.

Biến thể:

Ngoài ra còn có một function **strchr** với cách hoạt động hoàn toàn giống như **strchr**, khác ở chỗ nó sẽ chỉ vào chữ cái cuối cùng được tìm thấy trong string thay vì chữ cái đầu tiên.

Lưu ý:

- Function strchr tìm kiếm và chỉ vào ký tự đầu tiên nó tìm thấy trong chuỗi.
- Function strrchr tìm kiếm và chỉ vào ký tự cuối cùng nó tìm thấy trong chuỗi.

VD: Nếu bạn muốn tìm ký tự 'r' trong chuỗi Program:

- Khi dùng strchr: hàm sẽ chỉ vào ký tự 'r' đầu tiên giữa ký tự 'P' và 'o'.
- Khi dùng strrchr: hàm sẽ chỉ vào ký tự 'r' cuối cùng giữa ký tự 'g' và 'a'.

Strpbrk: chữ cái đầu tiên trong danh sách.

Function này có cách hoạt động khá giống với function trước. Nó sẽ tìm một trong những chữ cái trong danh sách bạn cho dưới dạng một dãy ký tự, ngược lại với strchr chỉ có thể tìm kiếm mỗi lần duy nhất một ký tự.

Ví dụ, nếu danh sách tìm kiếm đưa vào có dạng « xfs » cho dãy ký tự « Text for test », Function sẽ cho kết quả là một pointer chỉ vào chữ cái đầu tiên tìm được của một trong những chữ cái trong danh sách. Nói rõ hơn, chữ cái đầu tiên trong « xfs » nó tìm thấy trong « Text for test » là x, nên strpbrk sẽ trả về pointer lên 'x'.

Prototype của function này là:

```
char* strpbrk (const char* string, const char* charactersSearch);
```

Test thử xem nào:

C code:

```
int main (int argc, char *argv[ ])
{
    char *subString;
    // Chúng ta tìm kiếm sự xuất hiện đầu tiên của 1 trong 3 ký tự x, f hoặc s trong "Text for test"

    subString = strpbrk("Text for test", "xfs");

    if (subString != NULL)
    {
        printf ("Sau khi tìm một trong ba ký tự: x, f, s trong "Text for test"\n");
        printf ("Bắt đầu từ ký tự đầu tiên được tìm thấy.\n");
        printf ("Biên string được in ra là: %s", subString);
    }
    return 0;
}
```

Console:

```
Sau khi tìm một trong ba ký tự: x, f, s trong "Text for test"
Bắt đầu từ một trong những ký tự đầu tiên được tìm thấy.
Biên string được in ra là: xt for test
```

Trong ví dụ này, tôi đưa trực tiếp các giá trị vào thẳng function (trong ngoặc kép). Chúng ta không bị bắt buộc phải tạo một biến số, việc viết trực tiếp như thế này khá tiện lợi.

Các bạn phải lưu ý những cách viết sau:

Nếu các bạn sử dụng ngoặc kép “ ” có nghĩa là chuỗi (mảng ký tự).

Nếu các bạn sử dụng móc đơn ‘ ’ có nghĩa là ký tự

Strstr: tìm kiếm chuỗi ký tự trong một chuỗi ký tự khác.

Function này sẽ giúp bạn **tìm kiếm chuỗi ký tự đầu tiên** tìm thấy trong một chuỗi ký tự khác.

Prototype của nó là:

```
char* strstr(const char* string, const char* stringSearch);
```

Prototype của function này khá giống với **strpbrk**, nhưng chú ý đừng nhầm lẫn, **strpbrk** tìm kiếm một chữ cái trong danh sách, còn **strstr** tìm kiếm nguyên cả dãy ký tự

Ví dụ:


C code:

```
int main (int argc, char *argv[ ])
{
    char *subString;
    // Chúng ta sẽ tìm kiếm chuỗi "test" trong "Text for test" :
    subString = strstr("Text for test", "test");
    if (subString != NULL)
    {
        printf ("Chuỗi ký tự đầu tiên mà bạn muốn tìm trong chuỗi Text for test là: %s\n", subString);
    }
    return 0;
}
```

Chuỗi ký tự đầu tiên mà bạn muốn tìm trong chuỗi Text for test là : test

Function strstr tìm kiếm dãy ký tự « test » trong “Test de test”.

Và trả lại kết quả, giống như những function khác, một pointer tại vị trí nó tìm thấy. Kết quả là NULL nếu nó không tìm thấy gì.

 Hãy nhớ kiểm tra trường hợp function tìm kiếm không trả về kết quả NULL. Nếu bạn không làm điều này trước, khi bạn muốn hiển thị một dãy ký tự có pointer là NULL, chương trình sẽ bị lỗi. Hệ điều hành sẽ dừng chương trình của bạn ngay tức khắc vì nó cố gắng xâm nhập vào một địa chỉ NULL mà nó không được phép.

Trong vd trên, tôi hài lòng với kết quả mà function này đã trả về. Trong thực tiễn, việc này thật sự **không cần thiết**. Bạn chỉ cần viết một đoạn code `if (result != NULL)` để biết rằng việc tìm kiếm tìm được một thứ gì đó. Trường hợp không tìm thấy gì, hiển thị thông báo “Không tìm được kết quả cần tìm”.

Tất cả đều tùy theo chương trình bạn viết, nhưng trong mọi trường hợp, những function này là cơ bản để bạn thực hiện các thao tác tìm kiếm liên quan tới văn bản.

sprintf: viết trong một chuỗi

Khác với các function trước đó, function này được tìm thấy trong **stdio.h**.

Tên của function này khiến bạn nhớ đến thứ gì quen thuộc.

Function này khá giống với function printf mà bạn đã được biết, nhưng thay vì in ra màn hình, sprintf sẽ viết vào một dãy ký tự! Bạn để ý có thể thấy tên của function này bắt đầu bằng “s” của từ “string”

Đây là một function rất tiện lợi để tạo ra một chuỗi, sau đây là một ví dụ nhỏ:

C code:

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[ ])
{
    char string[100];
    long age = 18;
    // Chương trình sẽ viết "You are 18 years old !"
    sprintf (string, "You are %ld years old !", age);
    // Chúng ta sẽ in chuỗi ký tự này ra màn hình để đảm bảo chương trình hiển thị đúng
    printf ("%s", string);
    return 0;
}
```

Console:

```
You are 18 years old !
```

Nó được sử dụng tương tự như printf, chỉ khác ở điểm bạn phải thêm vào một parameter ngay vị trí đầu tiên, đó là một pointer chỉ đến chuỗi bạn cần viết vào.

Trong ví dụ của tôi, tôi viết trong chuỗi “You are %ld years old !”, tại đó %ld sẽ được thay thế bằng giá trị của biến số age. Tất cả các nguyên tắc của printf đều có ở đây, bạn có thể sử dụng %s để điền vào một chuỗi ký tự khác vào đấy.

Như thường lệ, hãy kiểm tra trước việc chuỗi của bạn có đủ lớn để nhận những ký tự mà sprintf sẽ gửi vào.

Nếu không, chúuuuuuu.... Bùm

Phải nói là các thao tác trên chuỗi ký tự trong C cần được thi hành một cách rất tỉ mỉ.

Bạn cần biết rằng tôi cũng không thể biết được tất cả những function có trong string.h. Tôi cũng không yêu cầu bạn phải học thuộc lòng. Nhưng bạn cần phải biết cách dãy ký tự hoạt động với \0 và những điều ở trên.

Tổng kết !!!

- Máy tính không thể làm việc với các ký tự, nó chỉ biết đến những con số. Chúng ta có những con số để giải quyết vấn đề liên quan đến những ký tự trong bảng chữ cái của mình bằng bảng mã tên là ASCII.
- Mỗi biến kiểu char chỉ có thể chứa một và chỉ một ký tự duy nhất. Chúng thường được lưu lại tại một ô địa chỉ bộ nhớ ngẫu nhiên trên máy tính, máy tính sẽ tự động sắp xếp và biên dịch những biến này.
- Để có thể tạo ra một từ hoặc một cụm từ chúng ta phải tạo ra một chuỗi. Trong trường hợp này, chúng ta sẽ sử dụng mảng ký tự.
- Tất cả các chuỗi đều được kết thúc bởi một ký tự đặc biệt, đó là ký tự kết thúc chuỗi '\0'
- Có rất nhiều functions dùng để xử lý chuỗi đã được viết sẵn trong thư viện string.h. Vì vậy đừng quên khai báo thư viện này ở đầu chương trình trước khi bạn muốn thao tác với chuỗi ký tự nhé.

Bạn cần nhớ rằng ngôn ngữ C có bậc “trung đối thấp”, có nghĩa là các thao tác của bạn rất gần với cách hoạt động của máy tính.

Lợi ích khác là hiện giờ bạn đã biết phương thức hoạt động của máy tính trên chuỗi ký tự. Những kiến thức tôi dạy bạn ở đây sẽ phát huy tác dụng trong tương lai, tôi có thể chắc đảm bảo với bạn.

Ngược lại, việc lập trình trên Java hay trên Basic không cần thiết phải hiểu cách thức hoạt động của máy tính, bạn hiện giờ đã bắt đầu hiểu làm thế nào máy tính hoạt động, điều này theo tôi là rất quan trọng.

Nghe có vẻ hay đấy nhưng có một điều đáng ngại là chương này hơi phức tạp. Bạn phải dự đoán trước độ lớn của array, nghĩ đến lưu lại \0... Các thao tác trên dãy ký tự không dễ dàng nắm vững bởi người mới bắt đầu, phải cần thêm một ít thực hành để có thể đạt được.

Nói về phương diện thực hành một cách chính xác thì tôi có công việc dành cho bạn.

Tôi khuyến khích bạn cố gắng thực hành thật nhiều. Còn điều gì tốt hơn bằng cách làm việc trên những chuỗi ký tự? Nếu chưa đủ phê, hãy cùng lúc làm việc trên chuỗi ký tự, mảng và pointer...

Và đây là việc tôi muốn bạn làm: Bạn vừa học một số functions trong thư viện string.h, nhưng bạn đã có đủ khả năng để tự viết lại các function cho riêng mình.

(?)Nhưng có cần thiết ko? Nếu các function đã được viết rồi, tại sao phải phí công viết lại?

Thật sự thì không cần thiết tí nào, và trong tương lai chắc chắn bạn sẽ sử dụng các function trong string.h chứ không phải những function của riêng bạn. Nhưng việc này sẽ giúp bạn luyện tập, tôi thấy rằng đây là một bài tập khá hay. Tôi đã chỉ cho bạn cách hoạt động của function strlen, điều đó có thể giúp bạn thực hiện các function khác.

Nhưng, đừng cố gắng viết lại những function như sprintf, đây là một function với độ phức tạp tương đối cao. Hãy tạm chấp nhận với các function có trong string.h.

Nếu bạn bị kẹt ở đâu đó, đừng ngại đặt câu hỏi trên các diễn đàn.

Nào, làm việc thôi!

Bài 5: Preprocessor

Tiền xử lý

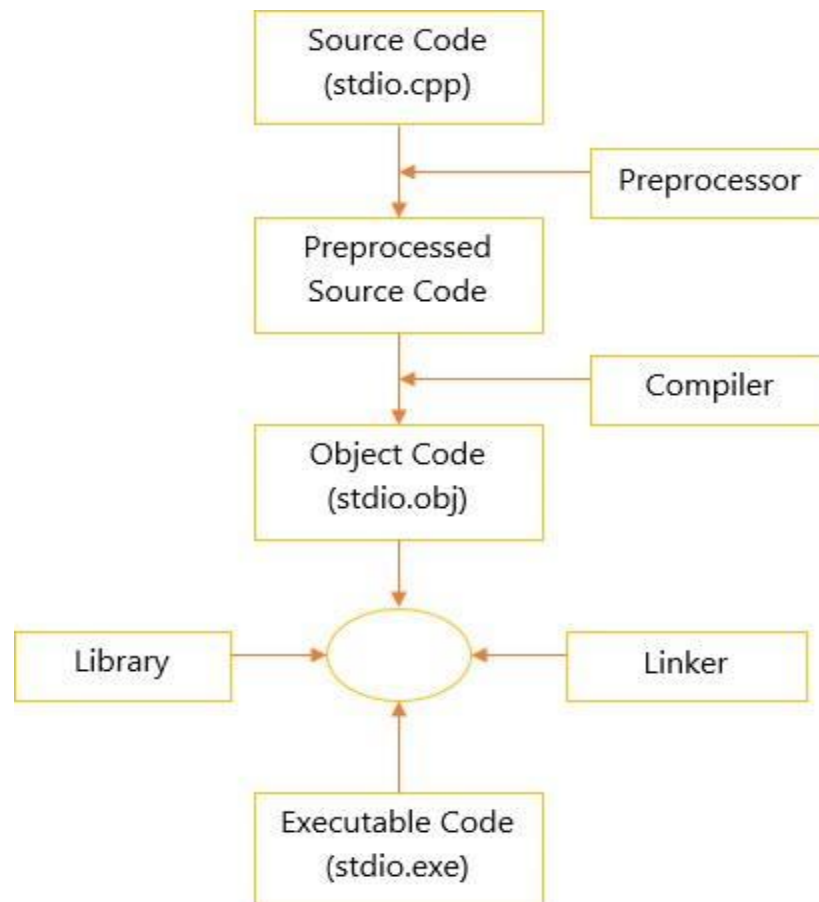
Sau những bài học về con trỏ (pointer), mảng (array) và chuỗi kí tự (string), chúng ta sẽ tạm nghỉ ngơi một chút.

Tôi nghĩ đến thời điểm này các bạn đã có được không ít những kiến thức lập trình từ các bài học trước và có lẽ chúng ta nên ngừng lại một chút cho dễ thở.

Nhưng điều này không có nghĩa là sẽ không có điều gì mới mẻ để học trong thời gian này. Nhìn chung trong bài này, bạn sẽ được học một số thứ đơn giản, cũng như được nhắc lại một vài kiến thức.

Bài này sẽ viết về Tiền xử lý (Preprocessor), đây là những chương trình được chạy trước khi biên dịch (Compilation).

Tôi có một sơ đồ thể hiện quá trình biên dịch (compile) của máy tính để tạo ra một chương trình. Chúng ta cùng nhau tham khảo để hiểu rõ hơn nhé:



Sơ đồ quá trình biên dịch

Có một điều cần biết là: nội dung của bài này sẽ giúp ích cho bạn rất nhiều. Và những kiến thức này khá đơn giản để hiểu...

Chỉ thị #include

Giống như tôi từng giải thích trong chương đầu tiên, chúng ta tìm thấy trong mã nguồn những dòng hơi đặc biệt gọi là **chỉ thị tiền xử lý** (preprocessor directives).

Những preprocessor directives có đặc tính sau: chúng luôn **bắt đầu bằng ký tự #**. Khá dễ dàng để nhận biết. Và dòng directive đầu tiên (cũng là duy nhất) mà chúng ta từng thấy cho đến thời điểm này đó là **#include**.

Dòng directive này cho phép **thêm nội dung một file khác vào file đang viết**.

Chúng ta đặc biệt dùng #include để thêm vào file.c các nội dung từ những file.h của các thư viện (stdio.h, stdlib.h, string.h, math.h) và cũng có thể là từ những file.h của riêng bạn.

Để thêm nội dung những file.h có trong thư mục cài đặt IDE của bạn, bạn cần sử dụng những ngoặc nhọn < > :

Code C:

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

Để thêm nội dung những file.h có trong thư mục chứa project của bạn, bạn cần sử dụng những dấu ngoặc kép:

Code C:

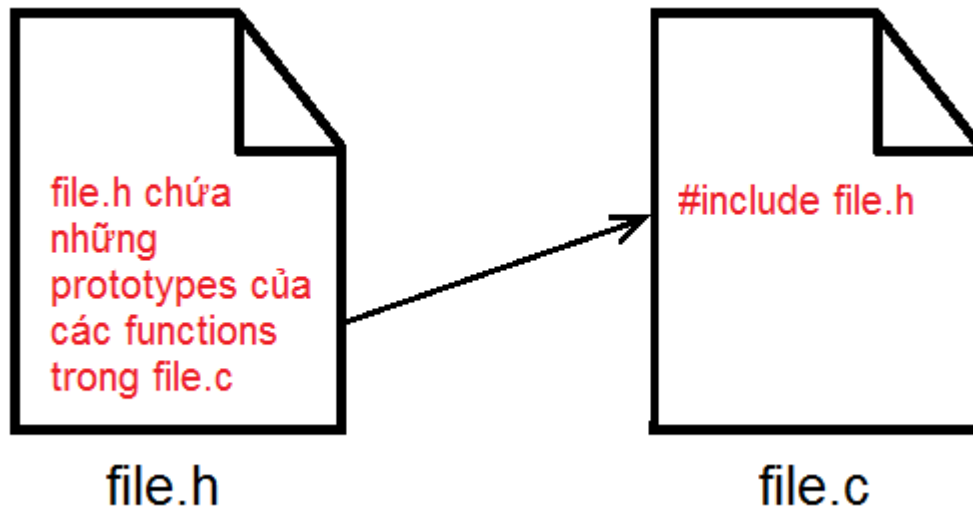
```
#include "myfile.h"
```

Cụ thể hơn, những chương trình tiền xử lý sẽ bắt đầu trước khi compile. Nó sẽ quét các tập tin của bạn để tìm ra những chỉ thị tiền xử lý trước, là tất cả những dòng bắt đầu bằng #.

Khi nó gặp directive #include, nó sẽ đặt nội dung của file được chỉ định vào vị trí #include.

Giả sử tôi có một « file.c » chứa code của các function và « file.h » chứa các prototypes của các function trong file.c

Đơn giản hơn bạn có thể xem biểu đồ sau :



Tất cả nội dung của file.h sẽ được đặt vào trong file.c, ngay tại vị trí đặt directive #include file.h

Dưới đây là những gì ta có trong file.c:

Code C:

```
#include "file.h"
long myFunction(int cai_nay, double cai_kia)
{
    /* Source code of function */
}
void otherFunction(long value)
{
    /* Source code of function */
}
```

Và những gì có trong file.h :

C code:

```
long myFunction(int cai_nay, double cai_kia);
void otherFunction(long value);
```

Khi chương trình tiền xử lý chạy đến đây, trước khi biên dịch file.c, nó sẽ đặt file.h vào trong file.c.

Cuối cùng, mã nguồn của file.c trước khi biên dịch sẽ giống như sau:

C Code:

```
long myFunction(int cai_nay, double cai_kia);
void otherFunction(long value);

long myFunction(int cai_nay, double cai_kia)
{
/* Source code of function */
}
void otherFunction(long value)
{
/* Source code of function */
}
```

Nội dung của file.h đã được đặt tại vị trí của dòng **#include**.

Thật sự không có gì quá khó hiểu đúng không? Tôi nghĩ rằng đã có khá nhiều bạn biết cách thức hoạt động của nó như thế nào.

Chức năng của các **#include** không có gì khác ngoài hành động chèn file này vào file khác, và các bạn phải biết rằng việc hiểu vấn đề này rất quan trọng.



Thay vì đặt tất cả các prototype vào trong các file.c, chúng ta nên chọn cách đặt những prototypes trong các file.h, việc này gần như được xem là một nguyên tắc cơ bản khi lập trình. Ta có thể ưu tiên để những prototypes này lên đầu file.c (thường gặp trong những file chương trình nhỏ), nhưng để tiện cho việc sắp xếp, tôi khuyên bạn hãy đặt những prototype này trong những file.h.

Khi chương trình bạn viết ngày càng phức tạp và có quá nhiều file.c, chỉ cần sử dụng **#include** để giúp chúng sử dụng chung 1 file.h, bạn sẽ cảm thấy mình thật may mắn khi không phải tốn quá nhiều thời gian để copy & paste lại những prototype có cùng 1 chức năng.

Chỉ thị #define

Chúng ta sẽ tìm hiểu một preprocessor directives mới: **#define**.
Lệnh directive này cho phép khai báo một hằng số (constant) của tiền xử lý (preprocessor). Nó cho phép đưa giá trị số vào một từ.

Lấy ví dụ:

Code C:

```
#define MANG_SONG_NHAN_VAT_BAN_DAU 3
```

Bạn sẽ viết theo thứ tự:

- #define
- Từ khóa sẽ được nhận giá trị
- Giá trị

Cần chú ý một chút về cách viết: tên của constant này thường được viết hoa toàn bộ, chủ yếu là để phân biệt với những constant mà ta được học trước đó:

Code C:

```
const long MANG_SONG_NHAN_VAT_BAN_DAU = 3;
```

Những constant sẽ chiếm một chỗ trong bộ nhớ, ngay cả khi giá trị không đổi, số « 3 » sẽ được lưu trữ đâu đó trong bộ nhớ. Không giống như trường hợp của các constant của preprocessor

Vậy nó hoạt động như thế nào? Đó là, #define sẽ thay thế tất cả những từ khóa trong mã nguồn bởi giá trị tương ứng. Tương tự như chức năng « find/replace » trong word bạn thường sử dụng.

Và dòng:

Code C:

```
#define MANG_SONG_NHAN_VAT_BAN_DAU 3
```

Sẽ thay thế tất cả từ MANG_SONG_NHAN_VAT_BAN_DAU có trong file bằng giá trị 3
Và đây là một ví dụ:

Một file.c trước khi qua preprocessor :

Code C:

```
#define MANG_SONG_NHAN_VAT_BAN_DAU 3
int main (int argc, char *argv[ ])
{
    long mang_song = MANG_SONG_NHAN_VAT_BAN_DAU;
    /* Code ... */
```

Sau khi qua preprocessor:

Code C:

```
int main (int argc, char *argv[ ])
{
    long mang_song = 3;
    /* Code ... */
}
```

Trước khi biên dịch, tất cả những #define sẽ được thay thế bằng các giá trị tương ứng. Compiler sẽ thấy nội dung trong file sau khi qua tiền xử lý, trong đó tất cả những chỗ cần thay thế đều đã được thực hiện

❓ Vậy đến thời điểm này, đâu là lợi ích của việc sử dụng những constant ?

Ok, như tôi đã giải thích với bạn là nó không chiếm thêm bộ nhớ máy tính. Điều này có nghĩa là, đến giai đoạn compilation thì trong đó chỉ còn những con số trong mã nguồn của bạn.

Một lợi ích khác khi sử dụng #define là việc thay thế sẽ được thực hiện trong tất cả các file chứa dòng #define. Khác với khi bạn khai báo một constant trong một function, nó chỉ có hiệu lực trong function đó, sau đó nó sẽ bị xóa đi. Nhưng đối với #define, nó sẽ tác động lên tất cả các function có trong file đó, và điều này thật sự rất tiện lợi cho những người lập trình.

Bạn muốn một ví dụ cụ thể về việc sử dụng #define?

Đây là một ví dụ mà bạn có thể ứng dụng ngay. Khi bạn mở một cửa sổ chương trình bằng C, hẳn là bạn muốn xác định những constant của tiền xử lý để chỉ ra kích thước của cửa sổ:

C Code:

```
#define CHIEUDAI_CUASO 800
#define CHIEURONG_CUASO 600
```

Ưu điểm của việc này là nếu về sau bạn có ý định thay đổi kích thước của các cửa sổ (chẳng hạn như vì lý do chúng quá nhỏ), bạn chỉ cần thay đổi lại những dòng #define và sau đó biên dịch lại.

Viết thêm rằng: những dòng #define thường được đặt trong những file.h, bên cạnh các prototype (Bạn có thể xem những file.h của các thư viện như stdio.h, bạn sẽ thấy ở đó luôn có những dòng #define!).

Những #define giống như « cách thức đơn giản », giúp bạn thay đổi kích thước cửa sổ bằng việc thay đổi những #define, thay vì phải rà soát lại toàn bộ nội dung file tìm nơi mà bạn mở cửa sổ để thay đổi lại kích thước. Việc này sẽ tiết kiệm rất nhiều thời gian cho bạn trong việc lập trình.

Tóm tắt lại, những constants của preprocessor cho phép « chỉnh sửa » chương trình của bạn trước việc compilation. Giống như một dạng mini-configuration.

Define cho kích thước array

Ta thường dùng các define để xác định kích thước các table. Đây là ví dụ :

Code C:

```
#define KICH_THUOC_TOI_DA 1000
int main (int argc, char *argv[ ])
{
    char line1[KICH_THUOC_TOI_DA], line2[KICH_THUOC_TOI_DA];
    // ...
}
```

❓ Nhưng... tôi nghĩ rằng chúng ta không thể đặt biến số (KICH_THUOC_TOI_DA) giữa những dấu ngoặc [] khi khởi tạo table (line1[], line2[]) ?

Đúng vậy, nhưng < KICH_THUOC_TOI_DA > **KHÔNG** phải là một biến số.
Giống như tôi đã nói, preprocessor sẽ thay đổi nội dung của file trước khi compilation, như sau:

C Code:

```
int main (int argc, char *argv[ ])
{
    char line1[1000], line2[1000];
    // ...
}
```

...và đoạn code này là chính xác.

Bằng cách xác định giá trị của KICH_THUOC_TOI_DA, các bạn có thể sử dụng nó để tạo những mảng có kích thước khác nhau. Nếu trường hợp sau này mảng đang sử dụng không đủ lớn, bạn chỉ cần thay đổi giá trị ở dòng #define, sau đó compile lại, và mảng dạng char của bạn đã sẽ nhận kích thước mới mà bạn vừa thay đổi.

Tính toán trong define

Chúng ta có thể thực hiện một vài phép tính trong define.

Ví dụ, đoạn code sau sẽ tạo constant CHIEUDAI_CUASO và CHIEURONG_CUASO, sau đó là SOLUONG_PIXELS chứa số lượng pixel hiển thị trong cửa sổ.
(việc tính toán khá đơn giản: chiều rộng*chiều dài):

C Code:

```
#define CHIEUDAI_CUASO 800
#define CHIEUCAO_CUASO 600
#define SOLUONG_PIXELS (CHIEUDAI_CUASO * CHIEUCAO_CUASO)
```

Giá trị của SOLUONG_PIXELS được thay đổi trước khi compile bằng giá trị của (CHIEUDAI_CUASO * CHIEUCAO_CUASO), có nghĩa là (800*600), 480000.

Hãy luôn đặt phép tính của bạn trong những dấu ngoặc đơn như cách tôi đã làm.

Bạn có thể thực hiện tất cả các phép toán cơ bản mà bạn biết: phép cộng (+), phép trừ (-), phép nhân (*), phép chia (/) và modulo (%).

Những constant được xác định trước

Ngoài việc bạn có thể tự tạo ra một số constant của riêng bạn, có tồn tại một số constant khác đã được xác định trước trong preprocessor

Ngay lúc này đây, khi tôi đang viết những dòng hướng dẫn này cho bạn, bạn cần biết rằng có một số constant tôi chưa từng sử dụng đến, tôi sẽ giới thiệu cho bạn một số constant có ích mà tôi biết.

Mỗi constant sẽ bắt đầu và kết thúc bởi hai dấu gạch dưới _

- `__LINE__` : vị trí của dòng code hiện tại
- `__FILE__` : tên của file hiện tại
- `__DATE__` : ngày tháng hiện tại
- `__TIME__` : thời gian hiện tại khi compile

Tôi nghĩ rằng những constant này có ích trong việc khắc phục lại các lỗi chương trình, lấy ví dụ :

C Code:

```
printf ("Co loi o dong %ld trong tap tin %s\n", __LINE__, __FILE__);  
printf ("Tap tin nay duoc bien dich vao ngay: %s luc: %s\n", __DATE__, __TIME__);
```

Console:

```
Co loi o dong 9 trong tap tin main.c  
Tap tin nay duoc bien dich vao ngay: 25 April 2015 luc: 15:36:10
```

Cách xác định đơn giản

Ta cũng có thể đơn giản viết:

Code C:

```
#define CONSTANT
```

...mà không cần biết giá trị của nó

Điều này có tác dụng báo cho preprocessor biết rằng từ CONSTANT đã được xác định, nó không có giá trị, nhưng nó « tồn tại ».

🔗 **Thật sự tôi không hứng thú lắm về nó, tác dụng của nó là gì vậy?**

Có lẽ bạn vẫn chưa thấy rõ tác dụng của nó, tôi sẽ chỉ ra cho bạn ở phần sau đây.

Macros

Chúng ta đã thấy rằng với các #define chúng ta có thể yêu cầu preprocessor thay thế một từ khóa bằng một giá trị.

Lấy ví dụ:

Code C:

```
#define SOLUONG 9
```

... có nghĩa rằng tất cả những từ « SOLUONG » trong code của bạn sẽ được thay thế bằng giá trị 9. Chúng ta thấy rằng nó cũng tương tự như chức năng find / replace trong Microsoft Word, trong trường hợp này thì nó được thực hiện bởi preprocessor trước khi compilation.

Tôi có một thông tin mới cho bạn!

Thực tế #define có tác dụng mạnh hơn rất nhiều, nó cho phép thay thế cả... một đoạn code! Khi ta sử dụng #define để find / replace một từ bằng một đoạn code, chúng ta gọi đó là macro.

Macro không dùng tham số (parameters)

Đây là một ví dụ của macro đơn giản:

Code C:

```
#define CUCKOO( ) printf ("Cuckoo");
```

Điều thay đổi ở đây, là những dấu ngoặc () được thêm vào sau từ khóa (ở đây là từ CUCKOO). Chúng ta sẽ xem ngay đây tác dụng của nó.

Test macro trong code source:

C Code:

```
#define CUCKOO( ) printf ("Cuckoo") ;  
int main (int argc, char *argv[ ] )  
{  
    CUCKOO()  
    return 0 ;  
}
```

Console:

Cuckoo

Tôi đồng ý với bạn, rằng đây không phải cách truyền thống mà chúng ta đã được học ở các bài trước.


Điều bạn cần phải biết đó là, macro sẽ được trực tiếp thay thế lại trong code source trước khi compile.

Đoạn code trên sẽ tương tự như đoạn code sau đây trong thời điểm compilation.

C Code:

```
int main (int argc, char *argv[ ] )  
{  
    printf ("Cuckoo");  
    return 0;  
}
```

Nếu bạn đã hiểu 2 đoạn code ở trên, bạn hầu như đã nắm bắt được nền tảng của các macro.

 Nhưng... chúng ta chỉ có thể đặt duy nhất một đoạn code trong mỗi macro thôi sao?

Không, thật may mắn là chúng ta có thể đặt nhiều dòng code trong mỗi macro. Chỉ cần đặt một kí tự \ trước mỗi dòng mới, tương tự như sau:

C Code:

```
#define GIOI_THIEU_BAN_THAN( ) printf ("Xin chao, toi ten la Minh\n"); \  
                                printf ("Toi song tai thanh pho HCM\n"); \  
                                printf ("Toi thích nghe nhạc\n");  
int main (int argc, char *argv[ ] )  
{  
    GIOI_THIEU_BAN_THAN()  
    return 0;  
}
```

Console:

Xin chao, toi ten la Minh
Toi song tai thanh pho HCM
Toi thích nghe nhạc

⚠ Cần lưu ý là trong main, khi gọi một macro chúng ta không đặt dấu chấm phẩy « ; » ở cuối dòng. Thực tế, đây là một dòng cho preprocessor, chúng ta không cần thiết phải kết thúc bằng dấu chấm phẩy.

Macro với nhiều tham số (parameters)

Vừa rồi chúng ta đã biết làm cách nào để tạo ra một macro không dùng parameter, có nghĩa là không ghi gì vào trong parameter của nó. Tác dụng chính của dạng macro này, là có thể « rút ngắn » một đoạn code dài có khả năng lặp lại nhiều lần trong code source của bạn.

Trong tình huống như vậy, những macro này càng hiệu quả hơn khi chúng ta đặt vào chúng những parameter. Nó hoạt động tương tự như các function.

C Code:

```
#define NGUOI_TRUONG_THANH(tuoi) if (tuoi >= 18) \
printf ("Ban la nguoi truong thanh\n");
int main (int argc, char *argv[ ])
{
    NGUOI_TRUONG_THANH(22)
    return 0;
}
```

Console:

Ban la nguoi truong thanh

⚠ Cần viết thêm rằng tôi có thể thêm vào đó một else để hiển thị « Ban van chua truong thanh ». Hãy làm thử để tập luyện, không có chút khó khăn nào cả và đừng quên đặt thêm dấu antislash \ trước mỗi dòng mới

Tôi nghĩ bạn có thể hiểu ngay cách hoạt động của đoạn macro này:

C Code:

```
#define NGUOI_TRUONG_THANH(tuoi) if (tuoi >= 18) \
printf ("Ban la nguoi truong thanh\n");
```

Tôi đặt vào trong dấu ngoặc () tên của biến số ta gọi là "tuoi". Trong tất cả đoạn code được thay thế, "tuoi" sẽ được thay thế bằng một số khi ta gọi lại macro (trong trường hợp này là 22).

Và đoạn code trên sẽ tương tự như sau ngay thời điểm biên dịch chương trình:

Code C:

```
int main (int argc, char *argv[ ])
{
    if (22 >= 18)
    printf ("Ban la nguoi truong thanh\n");
    return 0;
}
```

Đoạn code source trên đã được thay đổi lại, giá trị của biến số "tuoi" đã được đưa thẳng vào.
Và chúng ta cũng có thể tạo một macro chứa nhiều parameters:

C Code:

```
#define NGUOI_TRUONG_THANH(tuoi, ten) if (tuoi >= 18) \  
printf ("Ban la nguoi truong thanh %s\n", ten);  
int main (int argc, char *argv[ ])   
{  
    NGUOI_TRUONG_THANH(22, "Minh")  
    return 0;  
}
```

Đó là phần lớn những gì ta có thể biết về macro.



Bình thường chúng ta không cần thiết phải sử dụng thường xuyên các macro. Nhưng ở một số trường hợp, trong các thư viện tương đối phức tạp như wxWidgets hay QT (thư viện dùng để tạo các cửa sổ bạn sẽ được học sau này) sử dụng rất nhiều các macro. Tôi thấy việc hướng dẫn bạn kể từ lúc này sẽ giúp bạn thích ứng dễ dàng hơn trong tương lai.

Điều kiện - Conditions

Hãy nắm rõ: bạn có thể tạo ra những điều kiện trong preprocessor
Đây là ví dụ về cách chúng hoạt động:

C Code:

```
#if condition  
/* Nhưng mã nguồn (source code) sẽ được biên dịch nếu điều kiện đưa ra là đúng */  
#elif condition2  
/* Nếu điều kiện trên không đúng, nhưng mã nguồn (source code) sẽ được biên dịch nếu điều kiện 2 đúng */  
#endif
```

Từ khóa #if cho phép đưa vào một điều kiện cho preprocessor. #elif có nghĩa tương tự như là else if.

Điều kiện sẽ dừng lại khi bạn đặt vào nó dòng #endif. Bạn cần ghi nhớ rằng không có các ngoặc {} trong các preprocessor.

Lợi ích của chúng là cho phép tạo các điều kiện để biên dịch chương trình.

Trong trường hợp điều kiện thỏa mãn, đoạn code phía sau sẽ được biên dịch. Nếu không nó sẽ bị xóa đi trong thời điểm biên dịch chương trình. Và nó sẽ không xuất hiện khi chương trình hoàn tất.

#ifdef và #ifndef

Bây giờ chúng ta sẽ thấy lợi ích của việc #define một constant không xác định trước giá trị, như tôi có hướng dẫn bạn trước đó:

C Code:

```
#define CONSTANT
```

Ta có thể sử dụng #ifdef để tạo ra điều kiện "Nếu hằng số đã được xác định."

#ifndef, để tạo ra điều kiện: "Nếu hằng số chưa được xác định."

Bạn hãy xem đoạn code sau:

C code:

Code C:

```
#define WINDOWS

#ifdef WINDOWS
/* Mã nguồn cho Windows */
#endif

#ifdef LINUX
/* Mã nguồn cho Linux */
#endif

#ifdef MAC
/* Mã nguồn cho Mac */
#endif
```

Đây là cách để viết các chương trình có thể hoạt động trên nhiều hệ điều hành khác nhau, ví dụ trong Windows, bạn chỉ việc đặt vào #define WINDOWS, sau đó biên dịch.

Nếu bạn viết chương trình hoạt động trên Linux (tất nhiên là trong mã nguồn của bạn phải có phần mã nguồn dành riêng cho Linux), bạn chỉ cần thêm vào: #define LINUX, biên dịch lại, và lần này máy tính sẽ biên dịch phần mã nguồn cho Linux, các phần khác sẽ được bỏ qua.

#ifndef để tránh các "inclusions vô hạn"

#ifndef thường được dùng trong các file h để tránh các "inclusions vô hạn".

❓ Thế nào là một inclusion vô hạn ?

Hãy tưởng tượng, khá đơn giản.

Tôi có một file A.h và một file B.h .

File A.h chứa một dòng include file B.h. Nội dung của file B sẽ được đưa vào trong file A.

Nhưng, hãy nghĩ đến trường hợp củ chuỗi sau, giả sử file **B.h** lại include ngược lại nội dung của **A.h**? Trường hợp này trong lập trình thường hay xảy ra.

File thứ nhất cần file thứ hai để chạy, file thứ 2 lại cần file thứ nhất để chạy.

Hãy bỏ 10 giây để suy nghĩ, bạn sẽ nhanh chóng nhận thấy chuyện gì sẽ xảy ra:

1. Máy tính đọc **A.h** và thấy cần nội dung của **B.h**
2. Nó nhảy vào **B.h** để đọc nội dung và nhận ra, ở đây cũng cần nội dung của **A.h**
3. Vì vậy, nó đưa nội dung của **A.h** vào **B.h**, và trong **A.h** lại bảo cần **B.h**!
4. Lại một lần nữa, máy tính đi tìm đồng chí **B.h** và gặp đồng chí **A.h** quen thuộc.
5. Vv và..vv,

Không cần phải là một cao thủ để hiểu rằng nó không bao giờ kết thúc giống như câu hỏi "Trứng hay gà có trước"

Trong thực tế, vì buộc phải thực hiện quá nhiều các inclusion, preprocessor dừng lại và bảo "Đm, tao chán mấy cái inclusion củ chuỗi này của mày lắm rồi !!" và đột nhiên việc biên dịch bị crash.

Vậy làm thế quái nào để tránh cơn ác mộng khủng khiếp này?

Đây là một trick. Và kể từ bây giờ, tôi yêu cầu bạn thực hiện chúng trong tất cả các file.h của bạn, và tất nhiên là không có trường hợp ngoại lệ:

Code C:

```
#ifndef DEF_FILENAME // Neu constant chua duoc xac dinh file nay chua duoc dua vao
#define DEF_FILENAME // Ta xac dinh constant de lan sau file nay se khong dua vao lai nua

/* Noi dung cua file.h (cac includes khac, cac prototypes cho cac functions, cac dong
defines...)*
#endif
```

Và sau đó bạn sẽ đặt vào giữa `#ifndef` và `#endif`, nội dung của file.h (các includes khác, các prototypes cho các functions, các dòng defines...)

Bạn đã hiểu rõ nó hoạt động thế nào rồi chứ? Tôi đã không hiểu trong lần đầu tiên khi tôi được hướng dẫn về phần này.

Hãy tưởng tượng, khi file.h được include lần đầu tiên. Máy tính đọc điều kiện "Nếu constant DEF_FILENAME chưa được xác định ". Vì đây chính là lần đầu tiên file được đọc, constant đó vẫn chưa xác định, do đó, preprocessor sẽ đọc nội dung bên trong if

Instruction đầu tiên máy tính thấy là như sau:

Code C:

```
#define DEF_FILENAME
```

Tại thời điểm hiện tại, constant đã được thiết lập. Nếu ở lần kế tiếp, file lại được yêu cầu include tiếp tục, điều kiện là không còn đúng nữa, và yêu cầu này không được thi hành.

Tất nhiên, bạn có thể gọi tên constant như cách bạn muốn. Tôi gọi nó là DEF_FILENAME vì thói quen,

Nhưng mỗi người có một sở thích riêng mà đúng không.

Điều quan trọng ở đây là đổi tên constant cho mỗi tập tin .h khác nhau. Không được dùng cùng một constant cho tất cả những file.h, nếu không máy tính chỉ đọc file.h đầu tiên chứ không phải các file khác.

Vì vậy, bạn hãy thay thế FILENAME với tên của file.h của bạn.



Nếu bạn muốn chắc chắn những điều bạn vừa nghe không phải vô nghĩa, tôi mời bạn tham khảo ý kiến các thư viện .h chuẩn có trong ổ cứng của bạn. Bạn sẽ tìm thấy người ta đã xây dựng chúng trên cùng một nguyên tắc (một ifndef ở đầu và một endif ở cuối). Họ muốn chắc chắn rằng không có bất kỳ inclusion infinie nào xảy ra.

Thật buồn cười, tôi cảm thấy tôi đang dạy bạn một ngôn ngữ lập trình mới.

Nghĩ thì cũng đúng một mặt nào đó, tuy nhiên, đối với các preprocessor, nó sẽ đọc mã nguồn của bạn ngay trước khi gửi cho trình biên dịch bằng ngôn ngữ của riêng nó.

Nó cũng có thể làm được 2-3 việc nhỏ nhặt khác mà tôi không hướng dẫn ở đây, nhưng nhìn chung, người ta thường sử dụng các preprocessor directives trong các file.h tương tự như cách bạn vừa được học.

Oh, đây là một lưu ý nhỏ trước khi kết thúc: Tôi chân thành khuyên bạn để thêm vài dòng trống ngay sau #endif ở cuối file.h của bạn.

Để tránh #endif là dòng cuối cùng của file, tôi đã gặp lỗi khi biên dịch và tôi đã vô cùng vất vả để tìm ra nguyên nhân xuất phát từ đâu. Nó báo lỗi "No new line at the end of file"

Vì vậy, hãy đặt 2-3 dòng trống sau #endif như thế này:

C Code:

```
#endif
```

Điều này cũng được áp dụng tương tự trong các file.c. Hãy đặt một số dòng trống ở cuối cùng, điều này sẽ giúp bạn tránh được các lỗi nhức đầu không đáng có.

Tôi chưa bao giờ nói với bạn rằng lập trình là một môn khoa học chính xác đúng không.

Đôi khi, bạn rơi vào các lỗi quá kỳ lạ mà đến nỗi bạn phải tự hỏi rằng ếo biết do chương trình mình viết hay do máy tính bị cái ếo gì mà biên dịch ếo được.

Đừng lo lắng nếu nó xảy ra với bạn, vì đây chỉ là một trong các tai nạn khi bạn chọn nghề nghiệp lập trình này. Nhưng trước sau gì thì các bạn cũng sẽ đúc kết được những kinh nghiệm đáng quý sau mỗi lần gặp lỗi thôi.

Nếu sau tất cả mọi nỗ lực tự mày mò tìm kiếm, đừng đập bỏ chiếc máy tính thân yêu và cũng đừng ngại ngần đi tham khảo ý kiến hoặc nhờ sự trợ giúp từ các bậc tiền bối hoặc bạn bè, những người đã có kinh nghiệm lập trình, họ sẽ chỉ cho bạn nguyên nhân và cách giải quyết các lỗi kỳ quặc này.

Chú thích: thuật ngữ, từ vựng:

1. Preprocessor - Tiền xử lý: Là 1 quá trình bao gồm những chương trình chạy trước khi biên dịch.
2. Preprocessor directives – Chỉ thị tiền xử lý : là những chỉ thị cung cấp hướng dẫn trình biên dịch xử lý trước các thông tin trước khi bước vào giai đoạn biên dịch thực tế.
3. Compile (v) & Compilation (n) – Biên dịch & Trình biên dịch: Là hoạt động dịch ngôn ngữ lập trình do bạn viết sang ngôn ngữ máy tính để tạo ra một chương trình máy tính.
4. Compiler – Công cụ thực hiện chức năng biên dịch (xem lại bài 1 – chương 1).
5. Prototype – Khai báo hàm nguyên mẫu: Một prototype thật ra là lời chỉ dẫn cho máy tính. Nó sẽ thông báo trước với máy tính có sự tồn tại của function (xem lại bài 1 – chương 2).
6. Macro - Tập lệnh thay thế: Đây được hiểu là một tập hợp các câu lệnh. Khi trong chương trình có những khối câu lệnh giống nhau thì người ta có thể định nghĩa chúng bằng 1 macro cho khối câu lệnh đó và sau đó dùng tên của macro này thay thế cho toàn bộ khối lệnh kia trong suốt quá trình viết chương trình.
7. Source code – Mã nguồn: Là tập hợp toàn bộ những gì bạn viết ra cho máy tính để nó biên dịch.
8. Constant – Giá trị hằng: Là những giá trị không đổi trong suốt quá trình làm việc của máy tính.

Bài 6: Tạo ra những biến kiểu riêng của bạn

Ngôn ngữ C cho phép chúng ta làm một số điều thật sự đặc biệt: tạo ra những biến kiểu riêng của bạn, ta có thể gọi là “những biến tự tạo”. Hôm nay chúng ta sẽ học về **struct** (cấu trúc) và **kiểu liệt kê** (enum).

Việc tạo kiểu biến của riêng bạn sẽ trở nên hữu ích khi các chương trình bắt đầu phức tạp hơn.

Hãy tập trung vì chúng ta sẽ tiếp tục sử dụng lại những kiến thức này trong những bài học tiếp theo.

Lưu ý rằng các thư viện thường định nghĩa theo kiểu riêng của chúng. Bạn sẽ không phải mất quá nhiều thời gian trước khi xử lý một kiểu biến, hoặc có thể sau này là những kiểu chương trình Window, Audio, Keyboard ...

Định nghĩa một cấu trúc (struct):

Cấu trúc (struct) là một tập hợp gồm những phần tử có nhiều kiểu khác nhau. Không giống như khi làm việc với mảng (array), chúng ta được yêu cầu sử dụng cùng một kiểu định dạng đối với các phần tử trong toàn bộ mảng. Với cấu trúc (struct), bạn có thể tạo ra một tập hợp gồm các biến kiểu int, long, char hoặc là double...

Những cấu trúc thường được định nghĩa trong file.h, cũng giống như khi khai báo những nguyên mẫu (prototypes) hay những định nghĩa (defines).

Chúng ta cùng xem một VD:

C code:

```
struct TenCauTruc
{
    int bien1;
    int bien2;
    int bienKhac;
    double soThapphan;
};
```

Để khai báo một cấu trúc, chúng ta sẽ bắt đầu bằng từ khóa “**struct**”, tiếp sau đó là tên đại diện cho tập hợp các phần tử (VD: sinhvien, taptin).



Riêng cá nhân tôi thường hay áp dụng nguyên tắc đặt tên của biến để đặt tên cho cấu trúc, tôi thường viết hoa chữ cái đầu cho dễ nhận biết. Giả sử, nếu tôi thấy cụm “**tenSinhvien**” thì có nghĩa đó là tên của một biến bình thường vì chữ cái đầu của nó không được viết hoa. Tương tự khi thấy cụm “**TenSinhVien**”, tôi biết đây là tên của một “biến tự tạo” trong cấu trúc.

Sau tên cấu trúc các bạn nhớ đóng mở ngoặc nhọn `{ }` giống khi thao tác với hàm.



Lưu ý một điều đặc biệt sau: Bạn phải đặt một dấu chấm phẩy (;) sau dấu đóng ngoặc nhọn. Điều này là bắt buộc vì nếu thiếu nó thì chương trình của bạn sẽ không thể biên dịch được.

Và những thứ nằm giữa 2 ngoặc nhọn đó có gì lạ ?

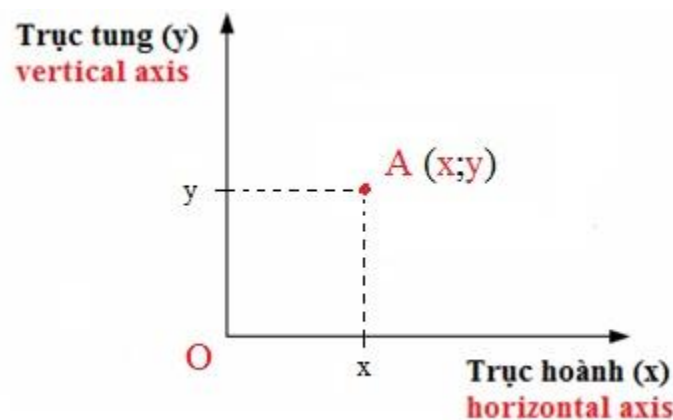
Đơn giản thôi, đó là những “biến thành phần” được tạo ra để xây dựng nên cấu trúc của bạn. Một cấu trúc thường có từ 2 biến trở lên, nhưng có lẽ bạn cũng không cần quan tâm quá nhiều về vấn đề này.

Như bạn đã biết, việc khởi tạo một biến cơ bản không quá phức tạp. Tất cả những cấu trúc mà bạn thấy thật ra chỉ là “tập hợp” của những biến kiểu cơ bản như *int*, *long*, ... Chẳng có điều gì mới mẻ ở đây cả.

Ví dụ về cấu trúc (struct):

Giả sử bạn muốn tạo ra một biến để lưu tọa độ một điểm trên màn hình. Chắc chắn bạn sẽ cần đến một cấu trúc như thế này khi tạo ra các game 2D trong những phần tiếp theo, đây là cơ hội để nâng cao trình độ một chút.

Đối với những bạn chưa có nhiều kiến thức về những khái niệm “hình học không gian 2 chiều” thì sau đây là một số giải thích cơ bản cho hình học 2 chiều (2D).



Hệ trục tọa độ Oxy chứa điểm A

Khi làm việc với không gian 2 chiều (2D), chúng ta có 1 hệ trục tọa độ gồm:

- Gốc tọa độ O. Tại đây giá trị gốc bằng 0 và tăng dần theo chiều mũi tên của 2 trục tọa độ.
- Trục hoành (trục x) chạy từ trái sang phải, chứa các giá trị gọi là hoành độ.
- Trục tung (trục y) chạy từ dưới lên trên, chứa các giá trị gọi là tung độ.

Một điểm bất kỳ trên mặt phẳng tọa độ 2 chiều thường có 2 thông số để xác định vị trí của nó trên mặt phẳng. Hai thông số đó gồm hoành độ (giá trị trên trục x) và tung độ (giá trị trên trục y).

Chúng ta thường dùng một biến tên “x” để biểu diễn giá trị của hoành độ, tương tự như vậy ta sẽ dùng biến “y” để biểu diễn giá trị của tung độ.

Nếu viết điểm **A (x;y)** có nghĩa là điểm này có tên là A, hoành độ của nó là x và tung độ là y.

VD: ta có một điểm **diembatky (20;10)**, có nghĩa là điểm này có tên là **diembatky**, hoành độ của nó là **x=20** và tung độ là **y=10**.

Bạn có thể viết một cấu trúc Toadodiem để lưu trữ các giá trị hoành độ trên trục x và tung độ trên trục y của một điểm.

Nào nào, nó thật sự không khó đâu:

C code:

```
struct Toadodiem
{
    int x; // hoành do cua diem
    int y; // tung do cua diem
};
```

Cấu trúc (struct) của chúng ta có tên là “Toadodiem” gồm có 2 biến “x” và “y” để lần lượt biểu diễn hoành độ trên trục x và tung độ trên trục y.

Nếu muốn bạn hoàn toàn có thể tạo ra một cấu trúc (struct) mới cho không gian 3 chiều (3D), chỉ việc thêm vào một biến “z” để biểu diễn tọa độ trên trục đó (thường gọi là cao độ). Với những kiến thức này, chúng ta có thể tạo ra một cấu trúc để quản lý các điểm trong không gian 3D.

Mảng cấu trúc (mảng struct):

Những cấu trúc có chứa mảng. Thật may mắn, chúng ta hoàn toàn có thể tạo các mảng cơ bản và mảng ký tự (string) trong cấu trúc.

Nào bây giờ giả sử chúng ta có một cấu trúc *Taikhuan* để lưu trữ thông tin của một người dùng:

C code:

```
struct Taikhuan
{
    char ten[100];
    char ho[100];
    char diachi[1000];

    int tuoi;
    int gioitinh; // Boolean : 1 = nam, 0 = nu
};
```


Cấu trúc Taikhoan chứa 5 biến thành phần, trong đó:

- 3 biến đầu tiên kiểu *char* lưu trữ các thông tin lần lượt là: tên, họ, địa chỉ.
- 2 biến còn lại kiểu *int* lưu trữ các thông tin: tuổi và giới tính. Riêng giới tính là một biến dạng boolean (bạn đã học ở các bài trước về boolean), biến này sẽ trả về 1 = đúng = giới tính là nam, trả về 0 = sai = giới tính là nữ, chúng ta tạm chia ra 2 giới tính thôi nhé ^^!

Bạn có thể ứng dụng cấu trúc này để tạo một chương trình lưu trữ danh sách người dùng. Dĩ nhiên là bạn có thêm một số biến khác để bổ sung những thông tin mà bạn muốn. Không có giới hạn số lượng biến trong một cấu trúc nên bạn đừng lo.

Sử dụng cấu trúc:

Bây giờ những cấu trúc của chúng ta đã được định nghĩa trong các file.h và chúng ta có thể sử dụng các function của chúng trong file.c

Vậy hãy cùng xem làm thế nào để tạo một biến mang kiểu **Toadodiem** (cấu trúc được tạo ở trên):

C code:

```
#include "main.h" // File.h chứa các prototypes và structs

int main (int argc, char *argv[ ])
{
    struct Toadodiem diembatky; // Khởi tạo biến diembatky có kiểu Toadodiem

    return 0;
}
```

Vừa rồi chúng ta đã tạo ra một biến “diembatky” mang kiểu biến “Toadodiem”. Biến này sẽ tự động bao gồm luôn 2 biến thành phần **x** và **y** (hoành độ và tung độ) mà ta đã khai báo trước đó.



Vậy chúng ta có bắt buộc phải thêm từ khóa “struct” mỗi lần khai báo biến không ?

Câu trả lời là **CÓ**: Điều này sẽ giúp máy tính phân biệt các biến tự tạo (VD như biến kiểu Toadodiem) với những biến cơ bản (VD như biến kiểu int).

Tuy nhiên các lập trình viên thường cảm thấy lười khi phải luôn thêm từ khóa “struct” mỗi khi khai báo các biến tự tạo. Để giải quyết vấn đề này, họ đã phát minh ra một lệnh đặc biệt, họ gọi nó là **typedef**.

Typedef:

Trở lại với những file.h có chứa những định nghĩa cấu trúc Toadodiem của chúng ta.

Chúng ta sẽ thêm vào một câu lệnh gọi là typedef để tạo ra một tên cấu trúc thay thế cho toàn bộ cấu trúc đó.

Bây giờ chúng ta sẽ thêm vào một dòng trước khi khai báo cấu trúc ở đầu đoạn code lúc nãy:

C code:

```
typedef struct Toadodiem Toadodiem;  
struct Toadodiem  
{  
    int x; // hoành do cua diem  
    int y; // tung do cua diem  
};
```

Tôi sẽ giải thích cho bạn về dòng mới được thêm vào này, nó sẽ được chia làm 3 phần chính (nói thêm với bạn là tôi không hề mắc lỗi khi lặp lại cụm Toadodiem 2 lần).

1. **typedef**: sẽ chỉ ra cho máy tính biết rằng chúng ta đang đặt một tên thay thế cho cấu trúc.
2. **struct Toadodiem**: đây là tên của cấu trúc mà bạn sẽ đặt tên thay thế với typedef.
3. **Toadodiem**: đây chính là tên mà bạn đặt để thay thế cho cấu trúc struct Toadodiem. Bạn có thể đặt một tên bất kỳ mà bạn thích, tôi đặt là Toadodiem để cho các bạn thấy rằng khi dùng typedef, bạn sẽ tạo ra một cụm từ thay thế cho cấu trúc với chức năng tương đương.

Rõ ràng điều này có nghĩa là khi bạn viết cụm từ **Toadodiem** thì nó sẽ thay thế cho toàn bộ cấu trúc **struct Toadodiem**. Bằng cách này, bạn sẽ không phải đặt cụm **struct Toadodiem** mỗi khi khai báo biến tự tạo của mình nữa.

Vậy bây giờ chúng ta sẽ viết lại đoạn code trong main.c sau khi đã dùng lệnh typedef nhé:

C code:

```
int main (int argc, char *argv[ ])
{
    Toadodiem diembatky; /* May tinh se hieu bien nay mang kieu cau truc Toadodiem sau khi da  
    duoc dat ten thay the boi typedef */

    return 0;
}
```

Tôi khuyến khích các bạn nên tập thói quen dùng typedef với các cấu trúc giống như cách tôi đã làm với cấu trúc *Toadodiem* trong bài học này. Hầu hết các lập trình viên đều làm như vậy. Việc này sẽ giúp các bạn tiết kiệm thời gian khi không phải viết lại nhiều lần từ “struct” trong cả đoạn code. Có một điều lạ là hình như một lập trình viên giỏi thì thường khá lười.

Chỉnh sửa các thành phần của cấu trúc:

Bây giờ thì biến **diembatky** đã được khởi tạo, và nếu chúng ta muốn thay đổi những thành phần trong nó thì sao. Làm thế nào để tác động vào biến **x** và **y**, nào cùng xem thử nhé:

C code:

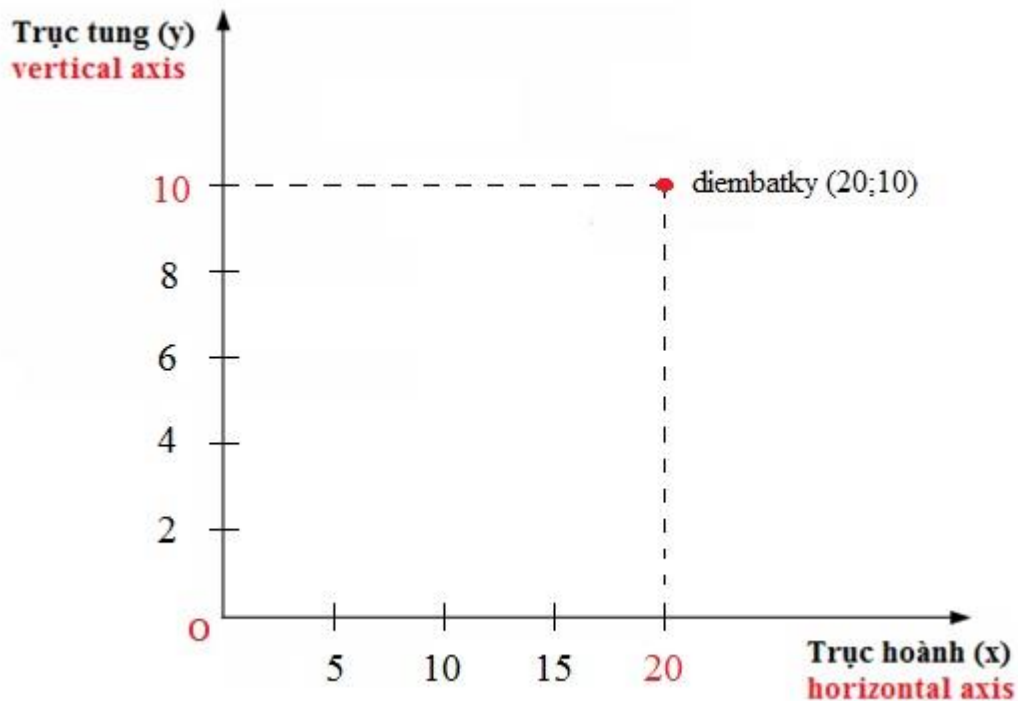
```
int main (int argc, char *argv[ ])
{
    Toadodiem diembatky;

    diembatky.x = 10;
    diembatky.y = 20;

    return 0;
}
```

Bây giờ thì giá trị của biến **diembatky** đã được thay đổi, chúng ta đã cho nó một giá trị hoành độ **x=20** và tung độ **y=10**. Bây giờ **diembatky** của chúng ta đang ở tọa độ (20;10).

Tôi sẽ minh họa một chút bằng mặt phẳng tọa độ Oxy trong hình học 2 chiều để các bạn có cái nhìn trực quan hơn:



Vậy tóm lại, để truy cập vào các biến thành phần của cấu trúc chúng ta sẽ viết theo cách sau:

C code:

```
tenbientutao.tenBienThanhPhanTrongCauTruc
```

Biến tự tạo **diembatky** tách biệt với biến thành phần **x** và **y** trong cấu trúc **Toadodiem**.

Sử dụng cấu trúc Taikhoan mà chúng ta đã tạo ở đầu bài và viết chương trình yêu cầu người dùng nhập tên, họ sau đó in ra màn hình.

Đoạn code sẽ như sau:

C code:

```
int main (int argc, char *argv[ ])
{
    Taikhoan nguoidung;

    printf ("Ten ban la gi ? ");
    scanf ("%s", nguoidung.ten);
    printf ("Ho cua ban la gi ? ");
    scanf ("%s", nguoidung.ho);

    printf ("Ho ten day du cua ban la %s %s", nguoidung.ho, nguoidung.ten);

    return 0;
}
```

Console:

```
Ten ban la gi ? Nhan
Ho cua ban la gi ? Sieu
Ho ten day du cua ban la Sieu Nhan
```

Chúng ta dùng hàm scanf để nhập giá trị cho biến **nguoidung.ten**, có nghĩa là lúc này giá trị đó sẽ được truyền thẳng vào biến **ten** của **nguoidung**. Bạn chỉ việc thực hiện tương tự như vậy đối với họ, tuổi, giới tính.

Dĩ nhiên là bạn cũng không cần phải học về cấu trúc (struct) thì mới có thể viết được chương trình trên. Mọi việc bạn cần làm chỉ đơn giản là tạo ra các biến lưu trữ tên, họ ... giống như các bài học mà ta đã học trước đây.

Nhưng điều thú vị ở struct là bạn có thể tạo ra những kiểu biến riêng cho từng trường hợp.

Giả sử bạn có một game dành cho 2 người:

C code:

```
Taikhoan nguoichoi1, nguoichoi2;
```

... Bạn thấy không, nhờ nó mà người chơi sẽ có thể lưu trữ tên, họ ... những thông tin riêng của từng người.

Tuy nhiên, chúng ta còn có thể làm tốt hơn, thậm chí chúng ta còn có thể tạo một mảng cho cấu trúc đó. Rất đơn giản thôi:

C code:

```
Taikhoan nguoichoi[2];
```

Để khai báo biến tên người chơi ở vị trí thứ 0 trong mảng trên thì bạn sẽ viết như sau:

C code:

```
nguoichoi[0].ten
```

Lợi ích của việc sử dụng mảng ở đây là bạn có thể tạo ra một vòng lặp để yêu cầu nhập thông tin người chơi thứ 1 và thứ 2 nhưng không cần phải viết đoạn code đó 2 lần. Chỉ cần tham chiếu vào từng thành phần của mảng để yêu cầu nhập từng tên, họ, địa chỉ ...

Bài Tập: Hãy tạo một mảng và sử dụng vòng lặp để yêu cầu người chơi nhập vào những thông tin khác. Hãy bắt đầu với 2 người chơi, nhưng sau khi đã nắm vững bạn có thể mở rộng hơn theo ý thích của bạn. Sau cùng hiển thị ra màn hình tất cả những thông tin mà bạn đã thu thập được từ những người chơi.

Khởi tạo một cấu trúc:

Đối với những cấu trúc cũng giống như các biến, mảng hay con trỏ, tôi khuyến khích các bạn nên khởi tạo chúng “không chứa giá trị nào” ngay từ đầu. Tôi nói thật đấy, hãy để tôi nhắc lại một chút, khi một biến bình thường được tạo ra, nó sẽ mang giá trị bất kỳ của ô địa chỉ được máy tính cấp phát cho nó. Đôi khi biến mang giá trị 0, nhưng đôi khi nó lại mang một giá trị rác của một chương trình khác đã sử dụng trước đó, những giá trị rác này thường không có ý nghĩa (chẳng hạn như: -69,69).

Sẵn tiện tôi sẽ nhắc lại một chút về các cách khởi tạo:

- Biến: chúng ta thường cho nó mang giá trị 0 lúc đầu (trong những trường hợp đơn giản).
- Con trỏ: chúng ta thường đặt giá trị là **NULL**. **NULL** là một định nghĩa có sẵn trong thư viện *stdlib.h* nói chung nó cũng có nghĩa là giá trị 0, nhưng chúng ta vẫn sử dụng NULL để biết được rằng đó là con trỏ chứ không phải biến bình thường.
- Mảng: chúng ta thường đặt giá trị 0 cho mỗi phần tử của mảng.

Đối với cấu trúc (struct), việc khởi tạo sẽ có một chút giống với mảng. Thật vậy, chúng ta cùng xem khi khởi tạo một biến cấu trúc thì như thế nào nhé:

C code:

```
Toadodiem diembatky = {0, 0};
```

Sau khi khởi tạo như trên máy tính sẽ tự động gửi giá trị lần lượt vào **diembatky.x = 0** và **diembatky.y = 0**.

Quay trở lại với cấu trúc Taikhoan (có chứa chuỗi ký tự). Bạn có thể bắt đầu tạo ra một chuỗi trong cấu trúc bằng cách viết cặp dấu ngoặc kép " " (không có thành phần nào chứa bên trong cặp dấu này). Tôi chưa nói cho bạn biết về nó ở những bài học về chuỗi trước đây, nhưng bây giờ vẫn chưa muộn để biết về nó.

Nhờ đó, chúng có thể khai báo trong cấu trúc Taikhoan những thông tin như ten, ho, diachi, tuoi, gioitinh như sau:

C code:

```
Taikhoan nguoidung = {"", "", "", 0, 0};
```

Cá nhân tôi không thường sử dụng cách này. Tôi thích sử dụng một hàm *taoToadodiem* với chức năng khởi tạo các biến cho biến *diembatky* của tôi.

Để làm được việc đó bạn phải tạo ra biến con trỏ. Nếu tôi chỉ sử dụng biến bình thường, một bản sao sẽ được tạo ra trong hàm (nó không phải là biến bạn đã tạo) và hàm sẽ thay đổi giá trị của bản sao đó chứ không phải giá trị của biến mà bạn đã khởi tạo. Nếu cảm thấy khó hiểu đoạn này, bạn có thể xem lại bài học cũ về con trỏ.

Vì vậy, bây giờ chúng ta sẽ phải học cách sử dụng con trỏ (pointer) trong cấu trúc (struct). Mọi thứ bắt đầu có chút thú vị rồi đây.

Con trỏ cấu trúc (pointer of struct):

Một con trỏ cấu trúc được tạo ra theo cách tương tự như một con trỏ kiểu int, double hay bất kỳ kiểu cơ bản nào khác:

C code:

```
Toadodiem* diembatky = NULL;
```

Chúng ta vừa khởi tạo một con trỏ cho cấu trúc Toadodiem, con trỏ có tên là diembatky.

Để tránh làm các bạn bối rối tôi xin phép nói thêm rằng bạn vẫn có thể đặt dấu * trước tên con trỏ như chúng ta thường làm trước đây, nó sẽ như thế này:

C code:

```
Toadodiem *diembatky = NULL;
```

Tôi vẫn thường làm như cách trên, vì trong trường hợp để định nghĩa nhiều con trỏ trên cùng một dòng thì bạn phải đặt dấu * trước mỗi con trỏ đó. VD:

C code:

```
Toadodiem *diembatky1 = NULL, *diembatky2 = NULL;
```

Gửi một hàm vào cấu trúc:

Những gì chúng ta quan tâm ở đây là làm sao để sử dụng một con trỏ cấu trúc trong hàm, từ đó ta có thể thay đổi trực tiếp giá trị của biến.

Chúng ta sẽ thử với vd này: Chúng ta chỉ cần đơn giản tạo ra một biến *Toadodiem* và sau đó gửi địa chỉ của nó vào hàm *taoToadodiem*. Hàm này sẽ qui định các thành phần có giá trị 0.

Hàm *taoToadodiem* sẽ cần một tham số (parameter): tham số đó là một con trỏ đến cấu trúc *Toadodiem* (a **Toadodiem*):

C code:

```
int main (int argc, char *argv[ ])
{
    Toadodiem diembatkyCuatoi;

    taoToadodiem(&diembatkyCuatoi);

    return 0;
}

void taoToadodiem(Toadodiem* diembatky)
{
    // Tao cac bien thanh phan cua cau truc o day
}
```

Biến *diembatkyCuatoi* đã được tạo ra và địa chỉ của nó sẽ được gửi vào hàm *taoToadodiem*, chúng ta gọi biến này là con trỏ (bạn có thể đặt tên nó như thế nào tùy ý, điều này không ảnh hưởng đến hàm).

Nào bây giờ đối với hàm *taoToadodiem*, chúng ta sẽ lần lượt khởi tạo giá trị cho các thành phần.

Đừng quên đặt dấu * trước tên của con trỏ để truy cập vào các biến của nó. Nếu thiếu dấu * bạn sẽ chỉ thay đổi địa chỉ con trỏ và đó không phải là điều chúng ta muốn máy tính thực hiện.

Ok, nhưng mà có một vấn đề ... chúng ta thật sự không thể làm được:

C code:

```
void taoToadodiem(Toadodiem* diembatky)
{
    *diembatky.x = 0;
    *diembatky.y = 0;
}
```

Trông thật đơn giản ... nhưng tại sao chúng ta lại không thể làm điều đó? Nguyên nhân là vì dấu chấm phân cách chỉ làm việc với các ký tự, nó không hiểu dấu * là gì. Nhưng chúng ta cần sử dụng dấu * để truy cập vào và thay đổi giá trị của biến.

Giải pháp cho vấn đề này là chúng ta sẽ đặt một cặp ngoặc đơn để bao phần dấu sao và trước dấu chấm ngăn cách lại. Lúc này chúng ta có thể truy cập vào và thay đổi giá trị của biến:

C code:

```
void taoToadodiem(Toadodiem* diembatky)
{
    (*diembatky).x = 0;
    (*diembatky).y = 0;
}
```

Đoạn code trên đã hoạt động, bạn có thể kiểm tra thử. Những biến kiểu cấu trúc Toadodiem đã được đưa vào hàm và khởi tạo cho chúng giá trị x=0, y=0.

Trong ngôn ngữ C, chúng ta thường khởi tạo các cấu trúc theo cách đơn giản mà ta đã thấy ở trên.

Tuy nhiên, đối với C++, việc khởi tạo thường được thiết lập trong các hàm.

C++ thật sự không có gì khác ngoài một sự “cải tiến” của C. Tất nhiên là có rất nhiều điều để nói về ngôn ngữ này, ít thì cũng tốn cả 1 cuốn sách để viết về nó, và chúng ta sẽ không thể học tất cả cùng một lúc bây giờ.

Một phép tắt thường được sử dụng phổ biến:

Bạn sẽ thấy rằng con trỏ được sử dụng rất thường xuyên. Thằng thẩn mà nói, ngôn ngữ C hầu như chỉ sử dụng những cấu trúc con trỏ. Tôi đang nói với bạn về điều này một cách rất nghiêm túc (không hề cười nhé, hehe)!

Như đã nói ở trên, khi sử dụng cấu trúc con trỏ thì ta phải viết thế này:

C code:

```
(*diembatky).x = 0;
```

Nhưng những nhà lập trình viên thiên tài vẫn thấy cách này chưa đủ nhanh, họ cảm thấy khó chịu với những cặp dấu ngoặc đơn. Ngay sau đó, những vị lười biếng thông minh này đã sáng tạo ra phím tắt sau đây để thay thế:

C code:

```
*diembatky -> x = 0;
```

Phím tắt này mô phỏng hình ảnh của một mũi tên, nó là sự kết hợp của một dấu trừ (-) và một dấu lớn (>).



Khi chúng ta viết `diembatky -> x` cũng tương đương với `(*diembatky).x`

Nhớ rằng bạn có thể sử dụng mũi tên (`->`) khi thao tác với con trỏ và nếu làm việc trực tiếp với các biến, bạn phải sử dụng dấu chấm (`.`) như chúng ta đã học ở đầu bài.

Nào hãy thử áp dụng những phím tắt vừa rồi vào hàm `taoToadodiem` xem như thế nào:

C code:

```
void taoToadodiem(Toadodiem* diembatky)
{
    *diembatky->.x = 0;
    *diembatky->.y = 0;
}
```

Hãy nhớ rõ cách dùng phím tắt mũi tên này, chúng ta sẽ còn dùng lại nó nhiều lần nữa. Cần thận đừng nhầm lẫn giữa việc dùng mũi tên (`->`) với dấu chấm (`.`). Mũi tên là dành cho con trỏ, và dấu chấm là dành riêng cho biến.

Cùng xem một ví dụ nhỏ để phân biệt rõ hơn giữa chúng nhé:

C code:

```
int main (int argc, char *argv[ ])
{
    Toadodiem diembatkyCuatoi;
    Toadodiem *contro = &diembatkyCuatoi;

    diembatkyCuatoi.x = 10; // Làm việc với một biến ta sử dụng dấu chấm
    contro -> x = 10; // Làm việc với một con trỏ ta sử dụng mũi tên
}
```

Giá trị của `x` được gán bằng 10 theo 2 cách: đầu tiên ta làm việc trực tiếp trên biến, lần thứ hai ta làm việc thông qua con trỏ.

Kiểu liệt kê (enum):

Kiểu liệt kê (enum) là một cách hơi khác để tạo ra các biến riêng của bạn. Kiểu liệt kê (enum) không bắt buộc phải chứa các “biến thành phần” như cấu trúc (struct). Đây là một danh sách “các giá trị phù hợp” cho một biến. Do đó kiểu enum sẽ chiếm 1 địa chỉ bộ nhớ để dùng cho các giá trị mà bạn xác định (và chỉ duy nhất mỗi lần 1 giá trị).

Vd:

C code:

```
typedef enum Volume Volume;  
enum Volume  
{  
    LOW, MEDIUM, HIGH  
}
```

Bạn thấy rằng chúng ta lại sử dụng typedef như đã dùng trước đó.

Để tạo một danh sách liệt kê, chúng ta dùng cụm **enum**. Danh sách của chúng ta tên là **Volume** (cái này trên tivi người ta hay gọi là âm lượng đó mà). Đây là một biến tự tạo giúp chúng ta chọn ra 1 trong 3 giá trị đã được chỉ ra LOW hoặc MEDIUM hoặc HIGH.

Bây giờ chúng ta đã có thể tạo ra một biến Volume (âm lượng) để điều chỉnh độ lớn của âm thanh khi nghe nhạc trên máy tính.

Đây là một ví dụ, khởi tạo biến âm lượng vừa cho nhạc:

C code:

```
Volume music = MEDIUM;
```

Sau này chúng ta vẫn có thể yêu cầu máy tính thay đổi giá trị của âm lượng thành HIGH hoặc LOW.

Sự kết hợp của những giá trị:

Bạn có thấy rằng tôi đã viết IN HOA những giá trị trong danh sách liệt kê. Việc này gọi cho các bạn nhớ về những hằng số (constants) và định nghĩa (defines) đúng không ?

Thật vậy, nó cũng gần như tương tự nhưng không hoàn toàn giống hẵn.

Trình biên dịch sẽ tự động gán một số cho từng giá trị trong danh sách liệt kê.

Trong trường hợp danh sách liệt kê Volume của chúng ta, LOW mang giá trị 0, HIGH và MEDIUM lần lượt mang giá trị 1 và 2. Sự liên kết được máy tính tự động sắp xếp, và nó luôn bắt đầu bằng giá trị 0.

Không giống như #define là một trình biên dịch tạo ra MEDIUM 1, kiểu liệt kê không phải là một tiền xử lý. Nó chỉ gần giống thôi.

Thực tế là khi bạn đã khởi tạo một biến music = MEDIUM, chương trình sẽ đặt giá trị 1 vào ô bộ nhớ đó.



Vậy, có lợi ích gì khi biết trước giá trị của MEDIUM là 1 và HIGH là 2 ... ?

Theo tôi thì không, chúng ta không cần quan tâm tới việc này. Trình biên dịch sẽ tự động liên kết các giá trị và đặt chúng vào từng biến. Chúng ta chỉ cần viết như sau:

C code:

```
if (music == MEDIUM)
{
    // Am thanh của chương trình chơi nhạc sẽ theo giá trị của Volume
}
```

Đừng để ý đến giá trị của MEDIUM là bao nhiêu, bạn cứ để cho trình biên dịch tự động quản lý các số giá trị trong danh sách.

Lợi ích của việc này là gì? Việc này giúp cho code của bạn dễ đọc hơn. Thật vậy, mọi người có thể dễ dàng đọc hiểu trước If của bạn (điều kiện được hiểu là nếu biến music là MEDIUM thì chơi nhạc ở mức vừa).

Gán một giá trị cụ thể:

Từ bây giờ, trình biên dịch sẽ tự động đặt số 0 cho giá trị thứ nhất và lần lượt tiếp theo là 1, 2, 3 ... theo thứ tự.

Nó có thể yêu cầu gán từng con số cụ thể cho mỗi giá trị thành phần của kiểu liệt kê.

Đó là những gì thú vị từ nó sao? Nào để tôi cho bạn thấy, giả sử trên máy tính của bạn âm lượng (volume) được qui định các mức từ 0 đến 100 (mức 0 có nghĩa là câm nín và 100 tức là hát điếc cả tai). Đây chính là cơ hội để chúng ta thử gán giá trị cho các thành phần trong kiểu liệt kê.

C code:

```
typedef enum Volume Volume;
enum Volume
{
    LOW = 10, MEDIUM = 50, HIGH = 100
}
```

Như các bạn thấy, mức volume LOW bằng 10% mức volume của máy tính, MEDIUM thì bằng 50% và tương tự HIGH là 100%. Người ta cũng có thể tạo ra một giá trị mới tên MUTE (câm nín). Chúng ta sẽ gán số 0 cho giá trị này. Bạn đã hiểu ra vấn đề đúng không.

Tổng kết:

- Cấu trúc (struct) là một kiểu biến tự tạo, bạn có thể tự mình tạo ra nó và sử dụng trong chương trình của bạn. Biến này do bạn định nghĩa, không như các kiểu biến cơ bản như int, double ...
 - Một cấu trúc luôn chứa những “biến thành phần”, những biến thành phần này là các biến cơ bản kiểu int, double ..., tương tự như với mảng nhưng cấu trúc có thể chứa nhiều biến khác kiểu.
 - Khi bạn muốn truy cập vào một biến thành phần trong cấu trúc, bạn có thể sử dụng dấu chấm để ngăn cách tên của biến bình thường và tên của biến thành phần của cấu trúc, VD: bienbinhthuong.bienthanhphan1
 - Nếu bạn muốn sử dụng con trỏ để truy cập vào biến thành phần, bạn chỉ cần thay dấu chấm thành mũi tên (->), VD: contro -> bienthanhphan1
 - Một kiểu liệt kê cũng là biến tự tạo, bạn có thể đưa vào đây danh sách các giá trị như trong VD trên: LOW, MEDIUM, HIGH.
-

Bài 7: Những thao tác làm việc với tập tin

Những lỗi của các biến chỉ tồn tại trong RAM. Một khi bạn đã hoàn thành chương trình của mình, tất cả các biến của bạn sẽ bị xóa khỏi bộ nhớ và giá trị của chúng sẽ không thể khôi phục lại.

Vậy thì làm cách nào để chúng ta có thể lưu lại điểm số cao nhất trong game?

Làm sao để một chương trình soạn thảo văn bản hoạt động nếu tất cả những gì bạn viết ra sẽ bị xóa khi tắt chương trình?

May mắn thay, bạn có thể đọc và ghi dữ liệu của các tập tin trong ngôn ngữ C. Những tập tin này được lưu vào ổ cứng (hard drive) của máy tính: lợi ích là những tập tin sẽ được lưu lại tại đó, ngay cả khi bạn dừng chương trình hay tắt máy tính.

Để có thể đọc và ghi dữ liệu vào các tập tin chúng ta sẽ ứng dụng tất cả những gì đã học trước đây: con trỏ, cấu trúc, chuỗi ký tự ...

Mở và đóng một tập tin:

Để đọc và ghi dữ liệu trong một tập tin chúng ta sẽ sử dụng các hàm (functions) có sẵn trong thư viện “stdio” mà chúng ta được học trước giờ.

Vâng và cũng chính nó là nơi chứa 2 function quen thuộc “printf” và “scanf”. Nhưng nó không chỉ chứa duy nhất 2 function này mà còn có cả những function được tạo ra để làm việc với các tập tin.



Tất cả những thư viện (libraries) mà chúng ta đã từng sử dụng cho tới nay (stdlib.h, stdio.h, math.h, string.h ...) được gọi là các thư viện chuẩn. Chúng sẽ được IDE tự động nhận diện dù cho bạn có chạy chương trình trên bất kỳ hệ điều hành nào Windows, Linux, Mac hay một hệ điều hành nào đó.

Những thư viện chuẩn có số lượng giới hạn và hỗ trợ cho phép bạn thực hiện một số điều cơ bản như chúng ta đã từng thấy trong những bài trước. Đối với những chức năng cao cấp hơn, như là mở một cửa sổ chương trình, ta phải tải về và cài đặt một thư viện mới. Chúng ta sẽ học về chúng sau !!!

Để chắc cú thì bạn nên luôn luôn bắt đầu những dòng code của mình với việc khai báo các thư viện chuẩn như *stdio.h* và *stdlib.h* trên đầu tập tin “file.c” của bạn:

C code:

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

Những thư viện này rất cơ bản, thực sự rất cơ bản, nhưng tôi vẫn khuyên bạn hãy luôn thêm nó vào tất cả những chương trình của bạn sau này, dù cho chúng có tác dụng gì đi nữa.

Okie! Bây giờ chúng ta sẽ xem những thư viện tuyệt vời đó có thể làm gì, chúng ta có thể giải quyết câu hỏi làm sao để mở hoặc đọc hoặc ghi dữ liệu vào một tập tin. Dưới đây là những gì luôn diễn ra khi bạn yêu cầu máy tính thực hiện các thao tác đó:

1. Chúng ta sẽ gọi một hàm mở tập tin, thứ sẽ trả về cho chúng ta một con trỏ đến tập tin.
2. Chúng ta sẽ kiểm tra xem thao tác mở tập tin có thành công không (tất nhiên là trước tiên tập tin đó phải tồn tại) bằng cách kiểm tra các giá trị con trỏ mà ta nhận được. Nếu con trỏ mang giá trị NULL, có nghĩa là thao tác mở tập tin đã thất bại, trong trường hợp này chúng ta không thể tiếp tục nữa (màn hình sẽ hiển thị thông báo lỗi).
3. Trong trường hợp thao tác mở tập tin thành công (con trỏ không mang giá trị NULL), chúng ta có thể thoải mái đọc, ghi dữ liệu lên tập tin thông qua chức năng của các hàm mà các bạn sắp được học.
4. Một khi bạn đã làm xong việc với các tập tin, hẳn là bạn phải nghĩ tới việc đóng nó lại đúng không? Và chúng ta có *fclose*.

Đầu tiên chúng ta sẽ tìm hiểu về 2 hàm *fopen* và *fclose*. Sau khi đã nắm rõ vấn đề, chúng ta sẽ bắt đầu học cách đọc nội dung của một tập tin và ghi dữ liệu vào nó.

fopen: hàm mở một tập tin:

Trong bài học về chuỗi ký tự (string), chúng ta đã biết cách sử dụng “nguyên mẫu hàm” (prototype) như một “hướng dẫn sử dụng” của hàm (function). Đó cũng là thói quen phổ biến của các lập trình viên, họ đọc các prototypes để hiểu về những functions có trong đoạn code.

Tuy nhiên tôi thấy rằng chúng ta vẫn cần giải thích thêm một số thứ:

C code:

```
FILE* fopen(const char* tenTaptin, const char* chedoMotaptin);
```

Hàm này có 2 tham số:

- Tên tập tin được mở.
- Chế độ mở tập tin, nhìn vào tham số này bạn có thể thấy được nó diễn tả những gì bạn muốn làm với tập tin: chẳng hạn như đọc hoặc ghi dữ liệu hoặc cả 2 chức năng cùng lúc.

Hàm này sẽ trả về một con trỏ tới FILE. Đây là con trỏ đến cấu trúc FILE. Cấu trúc này đã được định nghĩa sẵn trong thư viện *stdio.h*, bạn có thể mở file thư viện ra để tự mình kiểm tra xem

FILE có cái quái gì trong đó nhưng tôi nghĩ dù có chứa gì đi nữa cũng không ảnh hưởng nhiều đến công việc của chúng ta.



Tại sao tên của cấu trúc (struct) FILE lại được viết hoa hết vậy? Tôi nhớ những cái tên được viết hoa là dành riêng cho các định nghĩa (define) và hằng (constant) cơ mà ???

Đây là quy tắc do tôi tự đặt ra (và rất nhiều lập trình viên khác cũng làm theo quy tắc tương tự). Bạn không bắt buộc phải làm theo những quy tắc của tôi và người tạo ra thư viện *stdio* chắc cũng có những quy tắc của riêng mình. Và lý do duy nhất tôi có thể giải thích cho bạn tại sao FILE lại được viết hoa trong thư viện *stdio*, là vì người tạo ra nó muốn như vậy ^^!

Đừng để một cái tên làm bạn mất tập trung. Rồi bạn sẽ thấy những thư viện chúng ta sắp học sau này cũng sử dụng quy tắc đặt tên rất giống với tôi, cụ thể là khi đặt tên cấu trúc thường chỉ cần viết hoa chữ cái đầu tiên.

Quay lại với hàm *fopen* của chúng ta. Nó trả về FILE*. Con trỏ này thật sự rất quan trọng, nó sẽ giúp ta có thể đọc hoặc ghi dữ liệu vào một tập tin.

Bây giờ chúng ta sẽ tạo một con trỏ của FILE để bắt đầu hàm của bạn (VD function main):

C code:

```
int main (int argc, char *argv[ ])
{
    FILE* taptin = NULL;

    return 0;
}
```

Con trỏ được khởi tạo giá trị NULL ngay từ đầu. Tôi nhắc lại cho bạn nhớ đây là một nguyên tắc cơ bản, hãy luôn khởi tạo giá trị NULL cho con trỏ ngay từ đầu nếu bạn không có giá trị nào khác để gán cho nó. Nếu không thực hiện điều này, rồi bạn sẽ tha hồ đọc thông báo lỗi trong tương lai nhé.



Bạn sẽ thấy rằng việc viết `struct FILE *taptin = NULL` là không cần thiết. Người tạo ra *stdio* cũng đã tạo ra một *typedef* như tôi đã dạy bạn để ta có thể viết là `FILE *taptin = NULL`.

Lưu ý rằng cấu tạo của cấu trúc có thể thay đổi tùy theo từng hệ điều hành khác nhau (không nhất thiết lúc nào cũng phải có các biến thành phần giống nhau). Vì vậy, chúng ta sẽ không bao giờ trực tiếp thay đổi nội dung của FILE (vd như ta sẽ không sử dụng `taptin.bienthanhphan1` để tác động vào biến thành phần của FILE). Nhưng bằng cách sử dụng các hàm chúng ta có thể làm việc được với FILE.

Nào bây giờ chúng ta sẽ gọi hàm *fopen* và lấy giá trị nó đã trả về trong con trỏ *taptin*. Nhưng trước khi làm điều đó, tôi phải giải thích cho các bạn về tham số thứ hai, tham số *chedoMotaptin*. Thật ra có một đoạn code đã được gửi cho máy tính để nói cho nó biết rằng tập tin này sẽ được mở trong chế độ “chỉ được đọc dữ liệu” (read-only) hoặc “chỉ được ghi dữ liệu” (write-only) hoặc cả 2 chế độ cùng một lúc.

Sau đây là những chế độ có thể áp dụng cho tập tin của bạn trong máy tính:

- “**r**” – read only – chế độ chỉ đọc: Bạn có thể đọc được nội dung của tập tin nhưng không thể viết thêm gì vào (*Tất nhiên là tập tin đã được tạo ra trước đó*).
- “**w**” – write only – chế độ chỉ viết: Bạn có thể viết thêm hoặc chỉnh sửa nội dung tập tin nhưng lại không đọc được nó (*Nếu chưa tồn tại thì tập tin sẽ được tạo ra ở chế độ này*).
- “**a**” – user addition – chế độ bổ sung: Bạn có thể viết thêm vào nội dung của một tập tin bắt đầu từ vị trí kết thúc nội dung tập tin đó (*Nếu chưa tồn tại thì tập tin sẽ được tạo ra ở chế độ này*).
- “**r+**” – read and write – chế độ đọc và viết: Bạn có thể đọc và viết trong nội dung của tập tin (*Tất nhiên là tập tin đã được tạo ra trước đó*).
- “**w+**” – read and write, with removal of content beforehand – chế độ đọc và viết đồng thời xóa sạch nội dung tập tin trước đó: Trước hết tập tin sẽ không chứa bất kỳ nội dung nào trước đó. Bạn có thể viết trong nội dung của tập tin và đọc nó sau (*Nếu chưa tồn tại thì tập tin sẽ được tạo ra ở chế độ này*).
- “**a+**” – add read/write at the end – chế độ đọc và viết ở vị trí cuối cùng: Bạn có thể viết và đọc nội dung của tập tin bắt đầu từ vị trí kết thúc của nội dung trong tập tin (*Nếu chưa tồn tại thì tập tin sẽ được tạo ra ở chế độ này*).

Để tôi nói thêm cho bạn một thông tin, tôi chỉ mới cho bạn thấy một phần các chế độ của thao tác mở tập tin. Thật ra còn một điều nữa, với mỗi chế độ mà bạn đang thấy ở trên, nếu bạn thêm ký tự “b” sau ký tự đầu tiên của mỗi chế độ (“rb”, “wb”, “ab”, “rb+”, “wb+”, “ab+”), lúc này các tập tin của bạn sẽ được mở trong chế độ nhị phân (binary). Đây là một cách khá đặc biệt mà chúng ta sẽ không đề cập ở đây. Thật ra chế độ hiển thị văn bản (text mode) là để lưu trữ dữ liệu dạng văn bản vd như “tên hiển thị” (chỉ hiển thị các ký tự), trong khi chế độ hiển thị số nhị phân (binary mode) là để lưu trữ dữ liệu dạng số vd như “dung lượng bộ nhớ” (hầu như chỉ toàn là số). Đây là sự khác nhau duy nhất giữa 2 chế độ hiển thị.

Dù là chế độ nào thì chúng vẫn hoạt động theo một cách tương tự nhau như chúng ta thấy trong bài học này.

Cá nhân tôi thường sử dụng chế độ “r” (chỉ đọc), “w” (chỉ viết), “r+” (đọc và viết). Chế độ “w+” có một chút nguy hiểm vì nó sẽ xóa sạch nội dung tập tin của bạn mà chẳng cho bạn một thông báo nào. Bạn chỉ nên sử dụng chế độ này khi muốn làm mới (reset) lại tập tin của mình.

Về chế độ “a” có lẽ nó sẽ hữu dụng trong một số trường hợp khi mà bạn chỉ muốn bổ sung một số thông tin vào cuối tập tin.



Nếu bạn chỉ có ý định đọc một tập tin, tôi khuyến khích các bạn nên chọn chế độ “r”, dĩ nhiên là chế độ “r+” cũng sẽ giúp bạn đọc được tập tin nhưng trong chế độ “r”, tập tin của bạn sẽ đảm bảo tính bảo mật, bạn sẽ không sợ người nào đó thay đổi nội dung của tập tin, đây cũng là một cách bảo vệ tập tin của mình.

Nếu bạn muốn viết một hàm loadLevel (để tải các cấp độ trò chơi) thì nên chọn chế độ “r”, còn nếu bạn viết một hàm saveLevel (để lưu lại các cấp độ trò chơi) thì chỉ cần chọn chế độ “w” là đủ.

Đoạn code sau đây sẽ mở một tập tin *test.txt* ở chế độ “r+” (đọc / ghi):

C code:

```
int main (int argc, char *argv[ ])
{
    FILE* taptin = NULL;

    taptin = fopen("test.txt", "r+");

    return 0;
}
```

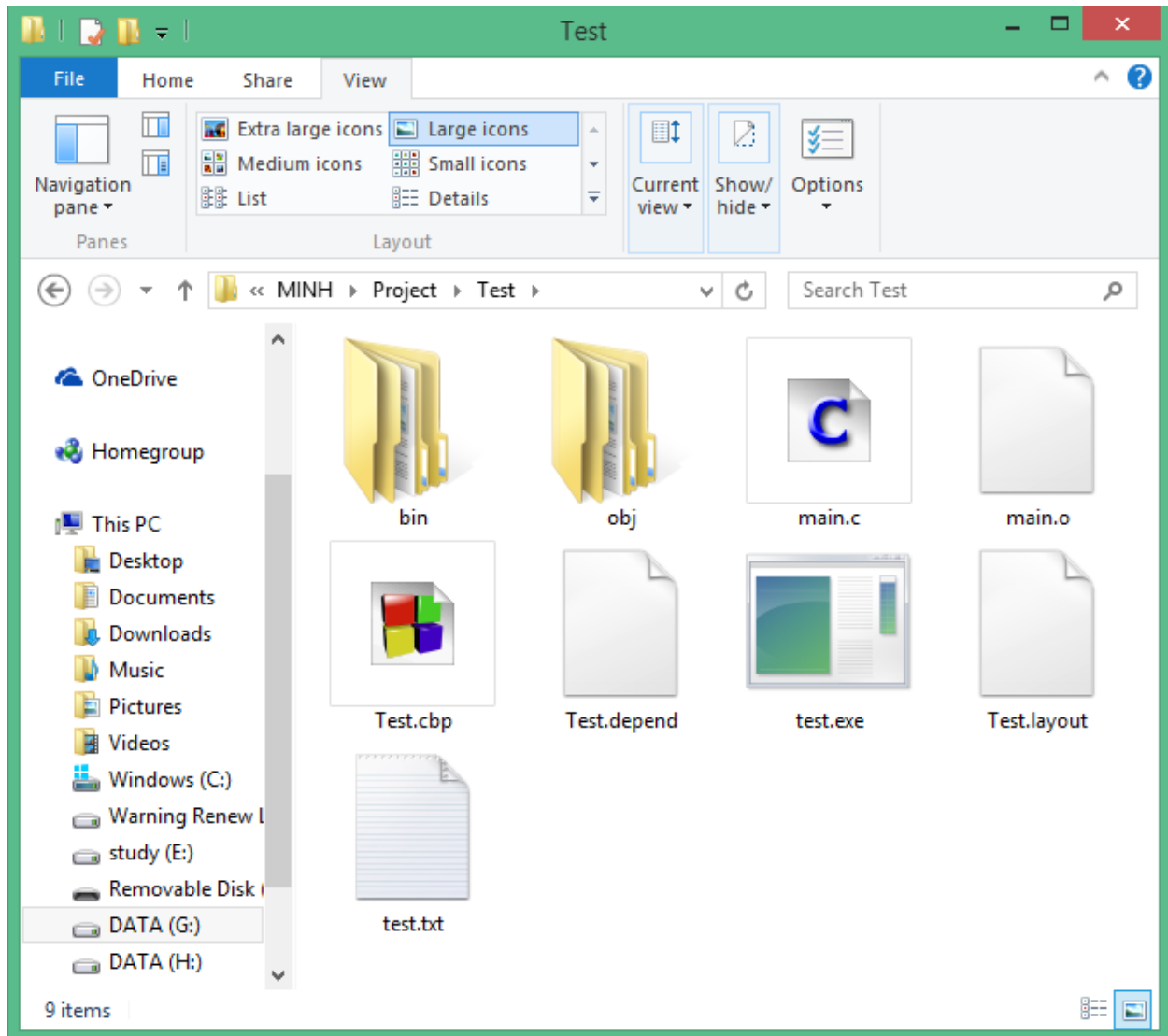
Con trỏ *taptin* sau đó sẽ thành con trỏ hướng đến tập tin *test.txt*



Nhưng tập tin *test.txt* được đặt ở đâu?

Nó nên được đặt trong cùng một thư mục với tập tin thực thi (executable) .exe của bạn.

Để tiện theo dõi bài học này, hãy tạo một tập tin *test.txt* trong thư mục như tôi đã làm trong hình dưới:



Như bạn thấy, tôi đang sử dụng IDE Code::Blocks, đây là câu trả lời cho câu hỏi tại sao lại tồn tại tập tin dự án (project) có đuôi **.cbp** (chứ không phải là tập tin dự án có đuôi chấm **.sln** như những người dùng Visual C++). Tạm thời, điều quan trọng cần để ý ở đây là tập tin *test.txt* được đặt trong cùng một thư mục với chương trình *test.exe*.



Vậy tập tin được tạo phải có đuôi *.txt* mới được sao?

Không. Đó là do bạn chọn phần định dạng của tập tin khi bạn mở tập tin. Bạn có thể sáng tạo ra kiểu định dạng của bạn VD như tạo một tập tin đuôi chấm **.level** để lưu lại cấp độ trò chơi của mình.



Vậy tập tin này lúc nào cũng phải nằm cùng thư mục chứa chương trình thực thi .exe sao?

Cũng không luôn. Tập tin này có thể được đặt trong thư mục con:

C code:

```
taptin = fopen("folder/test.txt", "r+");
```

Bây giờ tập tin *test.txt* được đặt trong thư mục con tên là *folder*. Phương pháp này được gọi là “đường dẫn tương đối”, và thường được sử dụng rộng rãi. Với cách này, chương trình của bạn sẽ ít bị gặp lỗi hơn.

Chúng ta cũng có thể mở một tập tin bất kỳ ở đâu đó trong ổ cứng máy tính. Trong trường hợp này bạn phải viết đường dẫn một cách chính xác và đầy đủ (ta có thể gọi là đường dẫn tuyệt đối):

C code:

```
taptin = fopen("C:\\Program Files\\Notepad++\\readme.txt", "r+");
```

Đoạn code này sẽ mở các tập tin *readme.txt* nằm trong C:\Program Files\Notepad++



Tôi đã dùng 2 dấu \ mỗi lần rẽ nhánh thư mục như bạn thấy. Nếu tôi chỉ dùng một dấu \ máy tính sẽ hiểu nhầm rằng bạn đang thêm vào một ký tự đặc biệt như \n hoặc \t. Để viết một dấu \ trong chuỗi ký tự, bạn phải viết nó 2 lần (viết là \\), lúc này máy tính sẽ hiểu rằng bạn muốn sử dụng một dấu \ này.

Có một nhược điểm là những đường dẫn tuyệt đối có thể chỉ hoạt động trên một hệ điều hành cụ thể. Đây không phải là một giải pháp linh động. Chẳng hạn như cũng là một đường dẫn nhưng trên Linux bạn phải viết như sau:

C code:

```
taptin = fopen("/home/Minh/folder/readme.txt", "r+");
```

Tôi khuyên bạn nên sử dụng đường dẫn tương đối thay vì đường dẫn tuyệt đối. Đừng sử dụng đường dẫn tuyệt đối nếu chương trình của bạn được viết riêng cho một hệ điều hành nào đó và các tập tin của bạn cũng nên được lưu ở một thư mục cụ thể trong ổ cứng của máy tính.

Kiểm tra thao tác mở tập tin:

Tập tin chứa địa chỉ con trỏ *taptin* của cấu trúc FILE. Nó đã được cấp phát bộ nhớ bởi *fopen* ().

Bây giờ có 2 trường hợp có thể xảy ra:

- Mở tập tin thành công và bạn hoàn toàn có thể tiếp tục thao tác với tập tin (có thể là đọc, hoặc viết thêm vào tập tin).
- Mở tập tin thất bại do tập tin đó chưa tồn tại hoặc tập tin đang được sử dụng bởi chương trình khác. Trong trường hợp này bạn không thể làm gì với tập tin đó hết.

Sau khi thực hiện thao tác mở tập tin chúng ta có thể kiểm tra xem có thành công không. Cách kiểm tra rất đơn giản thôi: nếu con trỏ mang giá trị NULL, mở tập tin thất bại. Nếu con trỏ mang một giá trị bất kì nào khác NULL, mở tập tin thành công.

Chúng ta sẽ làm như sau để kiểm tra:

C code:

```
int main (int argc, char *argv[ ])
{
    FILE* taptin = NULL;

    taptin = fopen("test.txt", "r+");

    if (taptin != NULL)
    {
        // khác giá trị NULL thì bạn có thể đọc và ghi dữ liệu vào tập tin rồi
    }
    else
    {
        // Bạn có thể cho nó hiện thông báo lỗi nếu thích
        printf ("Không thể mở tập tin test.txt");
    }

    return 0;
}
```

Hãy làm như trên mỗi khi muốn mở tập tin. Nếu bạn không làm hoặc tập tin không tồn tại, chương trình sẽ gặp lỗi.

fclose: đóng một tập tin

Nếu tập tin được mở thành công thì bạn có thể đọc và ghi thêm dữ liệu vào nội dung của nó (chút nữa thôi chúng ta sẽ thấy cách làm).

Một khi bạn đã xong việc với tập tin thì bạn phải đóng nó lại đúng không? Chúng ta sẽ thực hiện thao tác này bằng *fclose*, việc này có vai trò giúp giải phóng bộ nhớ cho máy tính, điều đó cũng có nghĩa là những tập tin được nạp vào RAM sẽ được xóa sạch.

Prototype của *fclose* là:

C code:

```
int fclose(FILE* taptin);
```

Hàm này chỉ có một tham số: đó là con trỏ của tập tin.

Nó trả về một giá trị kiểu *int* cho biết đã đóng được tập tin chưa, giá trị đó là:

- Giá trị bằng 0: Nếu tập tin được đóng thành công.
- Giá trị là EOF (End Of File): Nếu việc đóng tập tin thất bại. EOF được định nghĩa sẵn trong *stdio.h* tương ứng với một số đặc biệt, giá trị này có nhiệm vụ thông báo cho máy tính về một lỗi đã xảy ra.

Để đóng một tập tin chúng ta sẽ làm như sau:

C code:

```
fclose(taptin);
```

Tóm lại chúng ta sẽ làm theo cách sau để mở và đóng một tập tin trong project như sau:

C code:

```
int main (int argc, char *argv[ ])
{
    FILE* taptin = NULL;

    taptin = fopen("test.txt", "r+");

    if (taptin != NULL)
    {
        // Ban co the doc va ghi du lieu vao noi dung tap tin

        // ...

        fclose(taptin); // Dong tap tin da duoc mo
    }

    return 0;
}
```

Tôi đã không sử dụng *else* để hiển thị thông báo lỗi nếu thao tác mở tập tin thất bại. Nhưng nếu muốn, tôi tin là bạn biết cách làm mà đúng không.

Hãy luôn nhớ đóng tập tin lại mỗi khi hoàn thành công việc, điều này giúp giải phóng bộ nhớ của máy tính.

Nếu bạn không giải phóng bộ nhớ cho máy tính, sau khi hoàn thành, chương trình của bạn sẽ chiếm rất nhiều bộ nhớ và nó có thể không sử dụng được. Với những ví dụ nhỏ như trên có lẽ bạn sẽ không thấy được sự quan trọng của việc này nhưng trong một chương trình lớn hơn, đây thật sự là một vấn đề quan trọng.

Việc quên không giải phóng bộ nhớ trước sau gì cũng sẽ xảy ra và bạn sẽ gặp phải một sự cố mang tên “tràn bộ nhớ”. Chương trình của bạn sẽ sử dụng nhiều bộ nhớ hơn cần thiết mà bạn không hiểu được lý do tại sao. Thường thì nguyên nhân đơn giản chỉ vì 1 hoặc 2 chi tiết nhỏ như vì quên dùng *fclose*.

Những phương pháp đọc/ghi dữ liệu trong tập tin:

Sau khi đã biết cách mở và đóng một tập tin, bây giờ chúng ta chỉ việc thêm vào một vài dòng code để đọc và ghi dữ liệu vào.

Chúng ta sẽ bắt đầu với việc học cách ghi dữ liệu vào một tập tin trước (cũng đơn giản thôi) và sau đó bạn sẽ học cách làm thế nào để đọc tập tin.

Ghi vào một tập tin:

Có một vài hàm có chức năng ghi dữ liệu vào tập tin, việc chọn ra cách nào thích hợp nhất là phụ thuộc vào bạn.

Đây là 3 hàm mà chúng ta sẽ học:

- *fputc*: viết một ký tự vào tập tin (duy nhất mỗi lần 1 ký tự).
- *fputs*: viết một chuỗi vào tập tin.
- *fprintf*: viết một chuỗi có định dạng vào tập tin, gần giống như hàm *printf*.

Hàm fputc:

Hàm này sẽ thêm vào tập tin mỗi lần 1 ký tự. Đây là prototype của nó:

C code:

```
int fputc (int kytu, FILE* taptin);
```

Hàm này có 2 tham số:

- Tham số 1: Biến đại diện cho ký tự được viết thêm vào (biến được khai báo kiểu int, như tôi đã từng nói với bạn nó cũng tương đương khi khai báo kiểu char, khác ở chỗ số ký tự có thể sử dụng ở đây nhiều hơn). VD bạn có thể viết trực tiếp ký tự 'A'.
- Tham số 2: Con trỏ đến tập tin để viết. theo như vd của chúng ta con trỏ tên là *taptin*. Lợi thế của việc gọi con trỏ *taptin* mỗi lần cần sử dụng là có thể mở nhiều tập tin cùng lúc và nhờ vậy bạn có thể đọc và viết thêm vào mỗi tập tin. Bạn không bị giới hạn phải mở 1 tập tin tại 1 thời điểm.

Hàm này trả về một giá trị *int*, tương đương với giá trị của ký tự được thêm vào. Giá trị *int* này sẽ là EOF thể hiện cho 1 lỗi, nếu hàm trả về 1 giá trị khác EOF thì mọi thứ vẫn bình thường.

Giống như khi thao tác mở tập tin thành công, tôi không thường kiểm tra xem *fputc* có thực hiện tốt nhiệm vụ của nó không, nhưng nếu bạn muốn thì bạn cứ việc kiểm tra lại.

Đoạn code sau đây sẽ thêm ký tự 'A' trong *test.txt* (nếu tập tin đã tồn tại nó sẽ được thay thế, nếu chưa thì nó sẽ được tạo ra). Có đầy đủ mọi thứ trong đoạn code bên dưới, mở tập tin, ghi thêm dữ liệu và đóng tập tin:

C code:

```
int main (int argc, char *argv[ ])
{
    FILE* taptin = NULL;

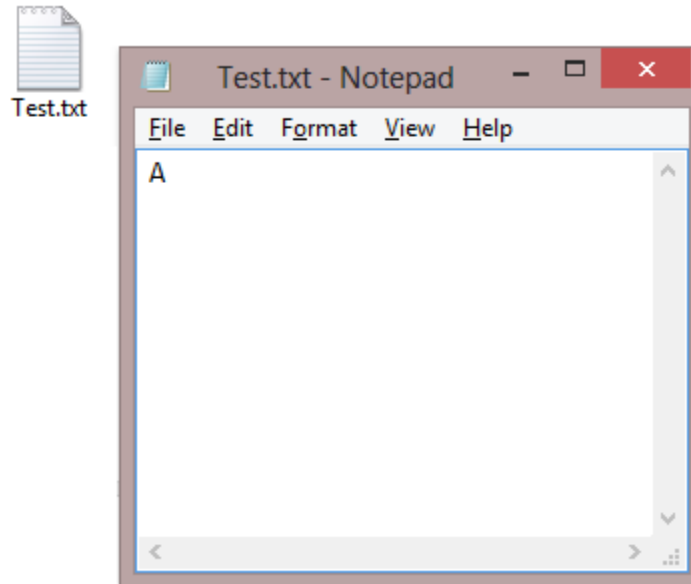
    taptin = fopen("test.txt", "w");

    if (taptin != NULL)
    {
        fputc('A', taptin); // Ghi thêm vào tập tin ký tự A
        fclose(taptin);
    }

    return 0;
}
```

Thử mở tập tin *test.txt* của bạn ra để kiểm tra thử xem.

Thật kỳ diệu đúng không, bây giờ nó đã chứa ký tự A như hình dưới:



Hàm *fputs*:

Hàm này rất giống với hàm *fputc*, chỉ có một khác biệt đó là nó sẽ ghi vào tập tin một chuỗi (string), khác với *fputc* chỉ có thể thêm vào một ký tự.

Điều này không có nghĩa là hàm *fputc* trở nên vô dụng, bạn sẽ phải sử dụng *fputc* trong những trường hợp chương trình muốn yêu cầu người dùng điền một ký tự duy nhất (trả lời câu hỏi trắc nghiệm chẳng hạn).

Sau đây là prototype của hàm:

C code:

```
char* fputs(const char* chuỗi, FILE* taptin);
```

Cả hai tham số của hàm này cũng rất đơn giản để hiểu:

Tham số 1: *chuỗi*, đây là chuỗi được thêm vào tập tin. Để ý rằng tham số này có kiểu *const char**: bằng cách thêm từ *const* vào, hàm này muốn nói rằng chuỗi này sẽ được xem như một hằng số. Tóm lại, bạn sẽ không thể sửa nội dung của chuỗi, điều này sẽ giúp bạn hiểu là: hàm *fputs* chỉ đọc và thêm vào chuỗi của bạn chứ nó không hề thay đổi gì nội dung chuỗi, cũng có nghĩa là thông tin bạn muốn thêm vào sẽ được bảo vệ an toàn.

Tham số 2: *taptin*, tương tự như trong hàm *fputc*, nó là con trỏ *FILE** của bạn để dẫn đề tập tin được đã được mở.

Nào bây giờ chúng ta sẽ thử thêm một chuỗi vào tập tin:

C code:

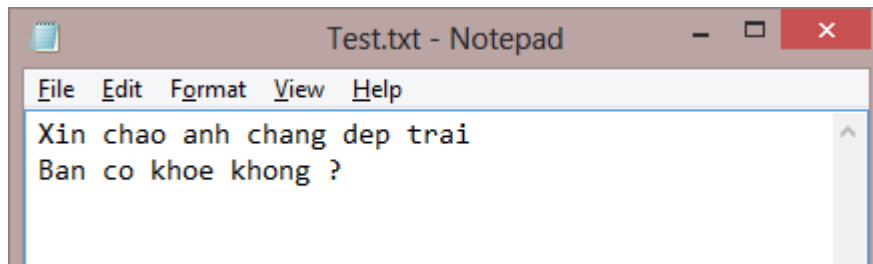
```
int main (int argc, char *argv[ ])
{
    FILE* taptin = NULL;

    taptin = fopen("test.txt", "w");

    if (taptin != NULL)
    {
        fputs("Xin chao anh chang dep trai\nBan co khoe khong ?", taptin);
        fclose(taptin);
    }

    return 0;
}
```

Tập tin sau khi chạy dòng code trên sẽ như sau:



Hàm fputs:

Sau đây là một vd khác cho hàm *fputs*. Chúng ta có thể sử dụng cách này để ghi dữ liệu vào một tập tin. Cách sử dụng cũng gần giống như hàm *printf*, ngoại trừ việc bạn phải chỉ định một con trỏ ở vị trí tham số đầu tiên.

Đoạn code sau đây sẽ hỏi tuổi của một người và ghi kết quả nhận được vào tập tin:

C code:

```
int main (int argc, char *argv[ ])
{
    FILE* taptin = NULL;
    int tuoi = 0;

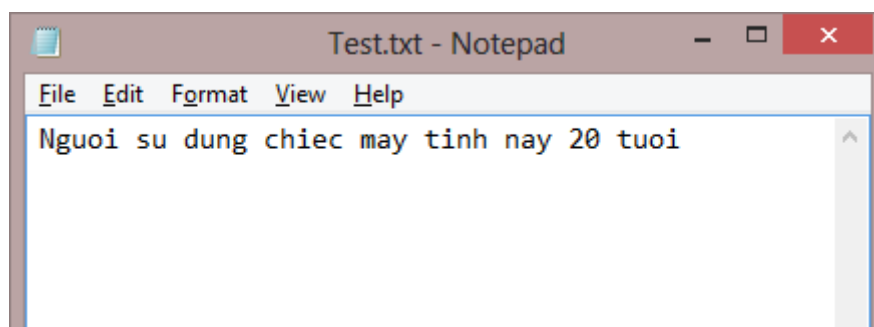
    taptin = fopen("test.txt", "w");

    if (taptin != NULL)
    {
        // Sau đây chúng ta sẽ hỏi tuổi
        printf (" Ban bao nhieu tuoi ? ");
        scanf ("%d", &tuoi);

        // Và bây giờ là ghi dữ liệu vào tập tin
        fprintf (taptin, "Nguoi dang su dung chiec may tinh nay %d tuoi", tuoi);
        fclose(taptin);
    }

    return 0;
}
```

Và kết quả khi bạn mở tập tin *test.txt* sẽ như hình sau:



Bạn có thể vận dụng lại những kiến thức đã học về hàm *printf* để áp dụng cho việc ghi dữ liệu vào tập tin đối với hàm *fprintf*. Đây cũng là lý do tại sao tôi rất hay sử dụng *fprintf* mỗi lần muốn thêm dữ liệu vào tập tin.

Đọc một tập tin:

Chúng ta sẽ sử dụng lại hầu hết các hàm dùng để ghi dữ liệu ở trên, chỉ có một chút thay đổi trong tên của chúng:

1. *fgetc*: Đọc một ký tự
2. *fgets*: Đọc một chuỗi
3. *fscanf*: Đọc một chuỗi có định dạng

Lần này tôi sẽ đi nhanh hơn một chút trong việc giải thích về những hàm này. Nếu bạn đã hiểu những gì tôi nói ở trên về việc ghi dữ liệu vào tập tin thì những kiến thức này chỉ là chuyện nhỏ.

Hàm *fgetc*:

Prototype là:

C code:

```
int fgetc(FILE* taptin);
```

Hàm sẽ trả về một giá trị *int*, có nghĩa là ký tự đó đã được đọc. Ngược lại nếu không thể đọc được, hàm sẽ trả về EOF.



Nhưng làm sao ta biết được mình sẽ đọc ký tự nào? Chẳng hạn như bạn muốn đọc ký tự thứ 3 hoặc thứ 10 thì làm sao để đọc?

Thật ra trong thực tế, khi bạn đọc một tập tin, sẽ xuất hiện một “dấu nhảy ảo”. Dấu nhảy này sẽ không hiển thị lên màn hình cho bạn thấy. Bạn có thể tưởng tượng nó giống như dấu nhảy khi bạn chỉnh sửa nội dung văn bản trên Notepad hoặc Microsoft Word. Nó có nhiệm vụ chỉ ra bạn đang ở đâu trong tập tin.

Chúng ta sẽ thấy ngay sau đây dấu nhảy ảo nằm ở vị trí nào trong tập tin và làm thế nào để thay đổi vị trí dấu nhảy đó. (chuyển nó về đầu tập tin hoặc một vị trí cụ thể nào đó của ký tự, vd như vị trí của ký tự thứ 10).

Mỗi lần bạn dùng *fgetc* để đọc một ký tự thì hàm sẽ gửi “dấu nhảy ảo” vào đó. Nếu bạn gọi hàm này lại một lần nữa, thì hàm sẽ tiếp tục từ vị trí đó đọc tiếp ký tự thứ 2, thứ 3 và cứ thế đọc tiếp. Bạn có thể tạo ra một vòng lặp (loop) để lần lượt đọc từng ký tự trong tập tin.

Chúng ta sẽ viết một đoạn code để lần lượt đọc tất cả các ký tự trong tập tin, đồng thời in ra màn hình những ký tự đọc được. Vòng lặp sẽ ngừng lại khi hàm *fgetc* trả về giá trị EOF (End Of File, nhằm thông báo cho máy tính là “kết thúc một tập tin”):

C Code:

```
int main (int argc, char *argv[ ])
{
    FILE* taptin = NULL;
    int kytuHientai = 0;

    taptin = fopen("test.txt", "r");

    if (taptin != NULL)
    {
        // Vòng lặp lần lượt đọc từng ký tự
        do
        {
            kytuHientai = fgetc(taptin); // Đọc ký tự
            printf ("%c", kytuHientai); // In ký tự do ra màn hình
        } while (kytuHientai != EOF); // fgetc tiếp tục được gọi lại vì biến kytuHientai khác EOF
        fclose(taptin);
    }

    return 0;
}
```

Màn hình console sẽ hiển thị toàn bộ nội dung tập tin, VD:

Console:

Xin chào, đây là nội dung của tập tin test.txt !

Hàm *fgets*:

Hàm này sẽ đọc một chuỗi trong tập tin. Nó sẽ giúp bạn tiết kiệm thời gian thay vì phải đọc từng ký tự một. Hàm sẽ đọc một chuỗi trên một dòng (nó sẽ dừng lại khi gặp ký tự xuống dòng **\n**). Nếu bạn muốn đọc nhiều dòng thì hãy tạo ra một vòng lặp.

Sau đây là prototype của hàm *fgets*:

C code:

```
char* fgets(char* chuỗi, int soKytuDuocdoc, FILE* taptin);
```

Hàm này có chứa một tham số hơi kỳ quặc, nhưng nó thực sự sẽ rất hữu ích: *soKytuDuocdoc*. Nó sẽ thông báo cho hàm *fgets* ngừng đọc dòng nội dung nếu vượt quá số lượng ký tự tương ứng.

Lợi ích: Nó sẽ bảo đảm rằng chúng ta sẽ không rơi vào tình trạng “tràn bộ nhớ”. Thật vậy, nếu một dòng của bạn quá lớn so với chuỗi, hàm có thể sẽ phải đọc nhiều ký tự hơn. Điều này có thể gây ra lỗi cho chương trình.

Đầu tiên chúng ta sẽ xem làm thế nào để đọc nội dung trên một dòng với hàm *fgets* (chúng ta cũng sẽ biết cách để đọc toàn bộ tập tin).

Để thực hiện điều này, chúng ta sẽ tạo ra một chuỗi đủ lớn để chứa nội dung của dòng mà ta sẽ đọc (ít nhất là vậy, vì ta không thể chắc chắn được 100% nội dung dòng đó là bao nhiêu). Bạn sẽ thấy lợi ích của việc sử dụng định nghĩa (define) để xác định trước kích thước của mảng (array):

C code:

```
#define SO_KY_TU_TOI_DA 1000 // Kích thước của mảng là 1000

int main (int argc, char *argv[ ])
{
    FILE* taptin = NULL;
    char chuoi[SO_KY_TU_TOI_DA] = ""; // Chuỗi có kích thước bằng SO_KY_TU_TOI_DA

    taptin = fopen("test.txt", "r");

    if (taptin != NULL)
    {
        fgets (chuoi, SO_KY_TU_TOI_DA, taptin); /* Có tối đa SO_KY_TU_TOI_DA trong tập
        tin được đọc, chúng được lưu trữ vào "chuoi" */
        printf ("%s", chuoi); // Hiện thị chuỗi lên màn hình

        fclose (taptin);
    }

    return 0;
}
```

Kết quả vẫn giống như code của hàm *fgetc* mà chúng ta đã thấy trước đó, đây là những nội dung hiển thị trên console:

Console:

```
Xin chào, đây là nội dung của tập tin test.txt !
```

Sự khác biệt ở đây là chúng ta không sử dụng vòng lặp. Nó hiển thị toàn bộ chuỗi trong một lần.

Chắc hẳn là bạn đã thấy những lợi ích của #define trong những dòng code trên nhằm xác định kích thước cho mảng, VD như SO_KY_TU_TOI_DA đã được sử dụng 2 lần trong đoạn code:

- Một lần để xác định kích thước cho mảng khi được khởi tạo.
- Và lần thứ hai trong hàm *fgets* nhằm giới hạn số ký tự tối đa sẽ được đọc.

Lợi ích của việc này là nếu bạn thấy rằng chuỗi chưa đủ lớn để đọc hết nội dung của một dòng trong tập tin, bạn chỉ cần thay đổi giá trị trên dòng định nghĩa (dòng chứa #define) và sau đó biên dịch lại chương trình. Điều này giúp bạn tiết kiệm thời gian vì không cần phải đọc lại toàn bộ code để tìm chỗ thay đổi kích thước của mảng. Tiền xử lý (preprocessor) sẽ thay thế toàn bộ giá trị cũ của SO_KY_TU_TOI_DA bằng giá trị mới mà bạn muốn.

Như tôi đã nói, hàm *fgets* đọc nội dung trên một dòng. Nó sẽ ngừng đọc dòng đó nếu vượt quá số ký tự tối đa được đọc do bạn quy định.

Nhưng câu hỏi là: Bây giờ nó chỉ đọc được mỗi lần một dòng vậy thì làm thế quái nào để ta có thể đọc được toàn bộ nội dung của tập tin? Câu trả lời vô cùng đơn giản: Vòng lặp (loop)

Hàm *fgets* sẽ trả về giá trị NULL nếu không đọc được nội dung mà bạn yêu cầu.

Các vòng lặp sẽ ngừng trước khi *fgets* trả về giá trị NULL.

Cần thêm vào một thứ để *fgets* không trả về NULL:

C code:

```
#define SO_KY_TU_TOI_DA 1000

int main (int argc, char *argv[ ])
{
    FILE* taptin = NULL;
    char chuoi[SO_KY_TU_TOI_DA] = "";

    taptin = fopen("test.txt", "r");

    if (taptin != NULL)
    {
        while (fgets(chuoi, SO_KY_TU_TOI_DA, taptin) != NULL); /* Cu viec doc noi dung tap
tin mien sao khong xuat hien loi (NULL)*/
        printf ("%s", chuoi); // Hien thi noi dung doc duoc len man hinh

        fclose(taptin);
    }

    return 0;
}
```

Đoạn code trên sẽ đọc toàn bộ nội dung của tập tin, từng dòng một.

Đây là những dòng thứ vị nhất của đoạn code, dòng code có sử dụng vòng lặp while:

C code:

```
while (fgets (chuoi, SO_KY_TU_TOI_DA, taptin) != NULL);
```

Dòng code trên thực hiện 2 việc: Nó sẽ đọc nội dung của một dòng trong tập tin và kiểm tra xem *fgets* có trả về giá trị NULL hay không. Có thể hiểu nội dung của dòng code trên là “đọc nội dung của một dòng và tiếp tục đọc dòng tiếp theo cho tới khi kết thúc tập tin”.

Hàm *fscanf*:

Vẫn là một nguyên tắc hoạt động tương tự với hàm *scanf* mà chúng ta đã từng học.

Hàm này đọc nội dung của tập tin nhưng nó phải được viết một cách chính xác.

Giả sử tập tin của bạn chứa 3 số được phân cách bằng khoảng trắng, VD đó là những điểm số cao nhất của một trò chơi 15 20 30.

Và bạn muốn lấy từng số đó dưới dạng biến kiểu int.

Hàm *fscanf* sẽ giúp bạn làm được điều đó một cách dễ dàng:

C code:

```
int main (int argc, char *argv[ ])
{
    FILE* taptin = NULL;
    int diemso[3] = {0}; // Mang chua 3 gia tri diem so cao nhat

    taptin = fopen("test.txt", "r");

    if (taptin != NULL)
    {
        fscanf (taptin, "%d %d %d", &diemso[0], &diemso[1], &diemso[2]);
        printf ("Cac diem so cao nhat la: %d, %d va %d", diemso[0], diemso[1], diemso[2]);

        fclose (taptin);
    }

    return 0;
}
```

Console:

Cac diem so cao nhat la: 15, 20 va 30

Như bạn sẽ thấy, hàm *fscanf* sẽ nhận biết 3 giá trị được phân cách nhau bằng những khoảng trắng ("%d %d %d"). Nó sẽ đưa vào mảng của chúng ta 3 thành phần. Sau đó bạn có thể dùng *printf* để hiển thị mỗi giá trị nhận được.



Bạn có để ý rằng trước đây tôi chỉ đặt một **%d** trong dấu ngoặc kép của hàm *scanf* thì trong lần này với hàm *fscanf* chúng ta có thể đặt một lúc nhiều **%d** nhập giá trị. Nếu tập tin của bạn được viết theo một quy chuẩn rõ ràng thì việc thu thập các giá trị sẽ được tiến hành dễ dàng hơn.

Di chuyển một tập tin:

Khi này, tôi đã nói với bạn về một “dấu nhảy ảo” đúng không? Bây giờ chúng ta sẽ tìm hiểu chi tiết hơn để thấy hết công dụng của nó.

Mỗi lần bạn mở một tập tin, sẽ có một “dấu nhảy ảo” xuất hiện và chỉ ra vị trí hiện tại của bạn trong tập tin. Bạn có thể tưởng tượng nó tương tự như dấu nhảy trong các trình soạn thảo văn bản (Notepad hoặc Microsoft Word), nó chỉ ra vị trí của bạn trong tập tin và bạn sẽ bắt đầu đọc hoặc ghi thêm dữ liệu từ vị trí đó.

Tóm lại, “dấu nhảy ảo” cho phép bạn đọc hoặc ghi thêm dữ liệu vào một tập tin từ một vị trí cụ thể.

Có 3 hàm chúng ta cần phải biết:

ftel: cho biết vị trí hiện tại của bạn trong tập tin.

fseek: chỉ định vị trí của “dấu nhảy ảo” tại một khu vực cụ thể.

rewind: đưa “dấu nhảy ảo” về vị trí bắt đầu của tập tin (tương tự, chúng ta cũng có thể sử dụng *fseek* để chỉ định vị trí “dấu nhảy ảo” về vị trí bắt đầu của tập tin).

Hàm *ftell*: Vị trí hiện tại trong tập tin

Cách sử dụng hàm này rất đơn giản. Nó trả về giá trị của “dấu nhảy ảo” như một biến kiểu *long*.

C code:

```
long ftell (FILE* taptin);
```

Giá trị được trả về cho biết vị trí hiện tại của “dấu nhảy ảo” trong tập tin.

Hàm *fseek*:

Prototype của hàm *fseek* là:

C code:

```
int fseek(FILE* taptin, long vitri_chuyenden, int vitri_hientai);
```

Hàm `fseek` sẽ di chuyển “dấu nhảy ảo” từ vị trí gốc (được chỉ định bởi biến `vitri_hientai`) đến vị trí của một ký tự trong tập tin (được chỉ định theo giá trị của biến `vitri_chuyenden`).

- Giá trị của `vitri_chuyenden` có thể là một số dương (để dấu nhảy di chuyển tiến lên), đứng im không di chuyển (giá trị bằng 0), và số âm (để di chuyển lùi lại).
- Giá trị khởi tạo có thể là một trong ba hằng số (constant) sau (khai báo `#define` nhé), xem nào:
 1. `SEEK_SET`: chỉ ra vị trí bắt đầu của tập tin
 2. `SEEK_CUR`: chỉ ra vị trí hiện tại của “dấu nhảy ảo”.
 3. `SEEK_END`: chỉ ra vị trí kết thúc của tập tin.

Sau đây là một vài ví dụ để chúng ta biết cách làm việc với những biến `vitri_hientai` và `vitri_chuyenden`.

- Đoạn code sau đây sẽ đặt “dấu nhảy ảo” vào vị trí của ký tự thứ 2 **sau** vị trí bắt đầu tập tin:

C code:

```
fseek (taptin, 2, SEEK_SET);
```

- Đoạn code này sẽ đặt “dấu nhảy ảo” vào vị trí của ký tự thứ 4 **trước** vị trí hiện tại của dấu nhảy:

C code:

```
fseek (taptin, -4, SEEK_CUR);
```

Lưu ý là với giá trị âm như trên, dấu nhảy sẽ di chuyển ngược về trước.

- Đoạn code sau đây sẽ đặt “dấu nhảy ảo” về vị trí cuối tập tin:

C code:

```
fseek (taptin, 0, SEEK_END);
```

Nếu bạn ghi thêm dữ liệu vào sau vị trí kết thúc của tập tin, máy tính sẽ bổ sung thêm dữ liệu đó cho tập tin của bạn (lần sau khi mở lại tập tin này bạn sẽ thấy những thông tin được bổ sung thêm ở vị trí cuối cùng).

Nhưng nếu bạn đặt “dấu nhảy ảo” ở đầu tập tin và bắt đầu ghi thêm dữ liệu vào thì lúc này, những dữ liệu cũ sẽ bị ghi đè lên. Chúng ta không thể “chèn” thêm dữ liệu vào tập tin trừ khi ta dùng một hàm để lưu lại những dữ liệu đứng sau trước khi chúng bị ghi đè lên.



Nhưng làm thế nào biết được vị trí nào là ở đâu để tìm đến mà đọc hoặc ghi dữ liệu?

Điều này tùy thuộc vào cách sắp xếp nội dung tập tin của bạn. Nếu tập tin này là do bạn viết ra, thì chắc hẳn là bạn phải biết rõ nó có cấu trúc như thế nào. Bởi vậy bạn sẽ biết những thông tin mà bạn cần nằm ở đâu. VD: bạn sắp xếp những điểm số người chơi ở vị trí 0, tên của người chơi cuối cùng nằm ở vị trí thứ 50...

Sau này chúng ta sẽ làm việc với những tập tin khổng lồ, và nếu như bạn không có một quy tắc sắp xếp nội dung tập tin riêng của mình, bạn sẽ không biết phải làm thế nào để lấy những thông tin mà mình cần ở đâu. Hãy nhớ rằng bạn chính là người sắp xếp tất cả những nội dung này trong tập tin, tất cả đều tùy thuộc vào bạn. Chẳng hạn như bạn quy định: “tôi để điểm của người chơi thứ nhất tại dòng 1, điểm của người chơi thứ 2 tại dòng 2...”

Hàm *fseek* có thể sẽ không hoạt động tốt khi dùng nó để mở những tập tin chứa nội dung dạng văn bản. Nói chung người ta thường dùng nó để mở các tập tin chứa nội dung dạng nhị phân.



Khi một người đọc hoặc ghi dữ liệu trong tập tin dạng văn bản, thường thì ký tự sẽ được thay thế bằng ký tự. Điều duy nhất hữu dụng trong chế độ tập tin văn bản khi sử dụng hàm *fseek* là nó giúp đưa bạn về vị trí đầu hoặc cuối tập tin.

Hàm rewind: quay về vị trí ban đầu.

Cách này tương tự như việc bạn sử dụng *fseek* để đưa “dấu nháy ảo” về vị trí 0 của tập tin.

C code:

```
void rewind(FILE* taptin);
```

Cách viết hàm của nó giống như nguyên mẫu ở trên, không có gì để giải thích thêm.

Đổi tên và xóa tập tin:

Chúng ta sẽ kết thúc bài học kỳ này với 2 hàm đơn giản:

- *rename*: đổi tên tập tin.
- *remove*: xóa tập tin.

Điểm khác biệt ở những hàm này là chúng không yêu cầu bạn sử dụng con trỏ *taptin*. Những hàm này chỉ cần xác định tên của tập tin để xóa hoặc đổi tên tập tin đó.

Hàm rename: Đổi tên tập tin.

Prototype của hàm là:

C code:

```
int rename(const char* teCu, const char* tenMoi);
```

Hàm sẽ trả về giá trị **0** nếu đổi tên thành công. Nếu không thành công nó sẽ trả về một giá trị khác 0. Và sau đây là một ví dụ:

C code:

```
int main (int argc, char *argv[ ])
{
    rename("test.txt", "test_rename.txt");

    return 0;
}
```

Hàm remove: Xóa tập tin.

Hàm này sẽ xóa tập tin ngay và luôn mà không cần hỏi ý kiến hay thông báo cho bạn.

C code:

```
int remove(const char* tentaptinMuonXoa);
```



Bạn phải cẩn thận khi sử dụng hàm này. Nó sẽ xóa tập tin của bạn mà không cần xác nhận lại. Tập tin sẽ không bị đưa vào thùng rác (recycle bin) mà nó sẽ bị xóa hoàn toàn khỏi máy tính của bạn. Sẽ không có cách nào khôi phục lại tập tin đó (trừ khi bạn dùng thủ thuật đặc biệt để khôi phục lại tập tin, nhưng thường thì rất khó thành công).

Và hàm *remove* này cũng sẽ kết thúc bài học đúng theo công năng của nó. Chúng ta đã tạo ra tập tin *test.txt* và bây giờ tự tay mình sẽ xóa nó:

C code:

```
int main (int argc, char *argv[ ])
{
    remove("test.txt");

    return 0;
}
```

Kết thúc bài học!

Bài 8: Cấp phát động

Tất cả những biến mà chúng ta từng sử dụng cho đến bây giờ đều được tự động tạo ra bởi trình biên dịch ngôn ngữ C. Phương pháp hoàn toàn đơn giản dễ hiểu. Nhưng vẫn có một cách thủ công hơn để tạo ra các biến, gọi là “cấp phát động”.

Một trong những lợi ích chính của việc cấp phát động là cho phép chương trình dự trữ sẵn một không gian bắt buộc để lưu trữ mảng trong bộ nhớ. Cho đến bây giờ, những mảng mà chúng ta tạo ra đều đã được cố định kích thước trong code. Sau bài học này chúng ta sẽ biết cách làm việc với mảng một cách linh hoạt hơn.

Bạn bắt buộc phải nắm rõ cách làm việc với con trỏ để có thể hiểu được bài học này. Nếu bạn vẫn còn lẩn tẩn thì tôi khuyên bạn nên dành chút thời gian xem lại những kiến thức của các bài học về con trỏ trước khi bắt đầu.

Khi bạn khai báo một biến, có nghĩa là chúng ta đang yêu cầu máy tính cấp phát bộ nhớ:

C code:

```
int number = 0;
```

Khi chương trình nhận được dòng code như trên thì sẽ xảy ra những điều sau:

1. Chương trình sẽ yêu cầu hệ điều hành (Windows, Linux, Mac Os ...) cho phép sử dụng bộ nhớ.
2. Hệ điều hành sẽ tiếp nhận yêu cầu và cho chương trình biết nơi nào có thể lưu trữ các biến (nó sẽ cho chương trình địa chỉ bộ nhớ mà nó dự trữ sẵn từ trước).
3. Khi hàm kết thúc công việc của nó thì biến cũng đồng thời bị xóa khỏi bộ nhớ. Chương trình của bạn sẽ nói với hệ điều hành rằng “Tao không cần mượn không gian bộ nhớ của mày ở địa chỉ này nữa, cảm ơn nhiều!”. Chuyện không đơn giản là nói lời cảm ơn với hệ điều hành mà bạn phải thấy rõ không phải ai khác, chính hệ điều hành điều khiển bộ nhớ.

Cho đến thời điểm này mọi thứ đều diễn ra một cách tự động. Mỗi khi khai báo biến, hệ điều hành tự động được chương trình liên hệ để yêu cầu cấp phát bộ nhớ.

Vậy chúng ta có thể làm điều này bằng tay một cách thủ công không? Không phải vì chúng ta thích tự làm khó mình với những thứ phức tạp (kể cả khi nó có hấp dẫn đi nữa), mà vì có đôi khi ta bắt buộc phải tự làm như vậy.

Trong chương này chúng ta sẽ:

1. Tìm hiểu về các chức năng của bộ nhớ (vâng, một lần nữa) để biết kích thước của biến sẽ thay đổi thế nào tùy thuộc vào kiểu của nó (int/float/char/double...)
2. Sau đó chúng ta đi thẳng vào vấn đề chính, làm thế nào để yêu cầu hệ điều hành cấp phát bộ nhớ một cách thủ công. Chúng ta sẽ làm cái việc ở đầu bài học đã nêu ra: “cấp phát động”.
3. Cuối cùng, chúng ta sẽ xem xem việc cấp phát động này sẽ mang lại lợi ích gì thông qua việc học cách tạo ra mảng mà không biết trước kích thước của nó trong chương trình là bao nhiêu.

Kích thước của các biến:

Tùy thuộc vào loại biến mà bạn đang muốn tạo ra (int/char/double/float...), bạn sẽ cần nhiều hoặc ít bộ nhớ.

Thực tế là để lưu trữ một số chạy từ -128 đến 127 (một biến kiểu *float*), máy tính chỉ cần 1 byte trong bộ nhớ, nó thật sự rất nhỏ.

Tuy nhiên với một biến *int*, nó thường chiếm 4 byte trong bộ nhớ, còn với một biến kiểu *double* sẽ là 8 byte.

Vấn đề là ... không phải lúc nào mọi thứ cũng diễn ra đúng như những gì chúng ta nói. Nó còn phụ thuộc vào máy tính nữa: chẳng hạn như biến kiểu *int* của bạn có thể chiếm 8 byte, ai mà biết được ?

Và mục đích của chúng ta là kiểm tra xem mỗi kiểu biến sẽ chiếm bao nhiêu bộ nhớ trên máy tính của bạn.

Có một cách rất dễ để kiểm tra đó là dùng *sizeof* ().

Không giống như những gì chúng ta từng học trước đây, nó không phải là một hàm (function), mà là một chức năng cơ bản của ngôn ngữ C, bạn chỉ cần đặt đối tượng bạn muốn kiểm tra kích thước vào trong cặp dấu () để thấy được điều bạn muốn.

Để biết kích thước của một biến kiểu *int* chúng ta sẽ làm như sau:

C code:

```
sizeof(int)
```

Tại thời điểm bắt đầu biên dịch, nó sẽ được thay thế bởi những con số: đó là số bộ nhớ mà một biến kiểu *int* sẽ chiếm trong bộ nhớ máy tính của bạn, *sizeof* (int) là 4, nghĩa là nó chiếm 4 byte trong bộ nhớ. Theo lý thuyết thì nó cũng sẽ có giá trị tương tự nhưng đây không phải là một quy luật không đổi. Chúng ta cùng kiểm tra bằng cách dùng *printf* để hiển thị giá trị,

VD:

C code:

```
printf ("char : %d byte\n", sizeof(char));  
printf ("int : %d byte\n", sizeof(int));  
printf ("long : %d byte\n", sizeof(long));  
printf ("double : %d byte\n", sizeof(double));
```

Màn hình console sẽ hiển thị:

```
char: 1 byte  
int: 4 byte  
dài: 4 byte  
Double: 8 byte
```

Tôi đã không kiểm tra tất cả các kiểu biến mà chúng ta đã biết. Tôi nhường phần đó cho bạn để kiểm tra kích thước các kiểu biến khác.

Bạn sẽ nhận thấy rằng kiểu *long* và *int* chiếm cùng một dung lượng bộ nhớ. Việc tạo một biến kiểu *long* cũng sẽ chiếm 4 byte như khi tạo một biến *int*.



Thật ra thì kiểu *long* chính là *long int*, tương tự như kiểu *int*. Đơn giản là nó cũng chỉ tạo ra thêm một cái tên mới chứ không có gì nhiều, chỉ vậy thôi! Trước đây khi bộ nhớ máy tính vẫn chưa tốt như bây giờ thì những cái tên khác nhau của kiểu biến thực sự rất hữu dụng cho máy tính của chúng ta. Các lập trình viên ngày trước đã luôn phải suy nghĩ chọn kiểu biến phù hợp nhất để tiết kiệm tối đa bộ nhớ.

Ngày nay thì dung lượng bộ nhớ máy tính thực sự đã rất lớn và vấn đề này đã không còn quá quan trọng. Nhưng vẫn có những người thích tạo ra những chương trình chiếm ít bộ nhớ nhất có thể. Tôi nghĩ rằng đó là các chương trình cho điện thoại di động, robot ...



Vậy chúng ta có thể biết kích thước của những kiểu biến tùy chỉnh do mình tạo ra (chẳng hạn như đối với “cấu trúc” – structure)

Câu trả lời là có! `sizeof ()` cũng hoạt động với cấu trúc (structure)!

C code:

```
typedef struct Toado Toado;
struct Toado
{
    int x;
    int y;
};

int main (int argc, char *argv[ ])
{
    printf ("Toado : %d byte\n", sizeof(Toado));

    return 0;
}
```

Console:

Toado : 8 byte

Với những cấu trúc chứa nhiều biến thành phần thì sẽ chiếm nhiều bộ nhớ hơn. Quá là hợp lý luôn đúng không ?

Một cách mới để kiểm tra.

Cho tới thời điểm này thì mô hình bộ nhớ của tôi vẫn chưa được rõ ràng lắm. Và bây giờ chúng ta sẽ làm rõ mọi thứ, cuối cùng thì ta cũng có thể biết chính xác kích thước của từng loại biến.

Nếu bạn khai báo một biến kiểu `int`:

C code:

```
int number = 18;
```

... và `sizeof (int)` đã chỉ ra rằng chúng ta sẽ mượn 4 byte của máy tính, sau đó biến sẽ chiếm 4 byte này của bộ nhớ.

Giả sử biến được cấp phát cho địa chỉ 1600 trong bộ nhớ. Chúng ta cùng xem hình sau để thấy rõ hơn:

Address	Value
1599	---
1600	18
1601	
1602	
1603	
1604	---

Bạn thấy rõ ràng ở đây là biến kiểu *int* chiếm 4 byte trong bộ nhớ.

Nó bắt đầu từ địa chỉ 1600 (địa chỉ mà biến được chỉ định lúc đầu) và kết thúc ở địa chỉ 1603. Biến tiếp theo sẽ không thể lưu vào địa chỉ nào trong 4 địa chỉ trên, nó sẽ được bắt đầu từ 1604.

Nếu chúng ta làm điều tương tự với một biến kiểu *char*, biến sẽ chỉ chiếm một byte trong bộ nhớ như hình sau:

Address	Value
1599	---
1600	18
1601	---
1602	---
1603	---
1604	---

Hãy thử tưởng tượng chúng ta có một mảng các biến kiểu *int*.

Mỗi một ô trong mảng sẽ chiếm 4 byte. Vậy nếu mảng của chúng ta có 100 ô thì sao:

C code:

```
int mang[100];
```

Vậy chính xác là chúng ta sẽ chiếm $4 \times 100 = 400$ byte trong bộ nhớ.



Vậy thậm chí mảng không chứa gì trong đó thì nó vẫn chiếm trước 400 byte bộ nhớ sao?

Dĩ nhiên rồi! Các không gian bộ nhớ đã được dự trữ sẵn, không có một chương trình nào khác có thể chạm vào nó (ngoại trừ sự tác động của bạn). Một khi biến đã được khởi tạo, nó sẽ ngay lập tức chiếm một vùng trong bộ nhớ.

Lưu ý nếu bạn tạo một mảng kiểu *Toado* thì:

C code:

```
Toado mang[100];
```

... Lần này chúng ta sẽ sử dụng $8 \times 100 = 800$ byte trong bộ nhớ.

Điều quan trọng là bạn phải cố gắng hiểu những tính toán nhỏ của phần sau.

Cấp phát động.

Nào bây giờ chúng ta sẽ tìm hiểu sâu về nó. Tôi sẽ nhắc lại mục tiêu của chúng ta: học cách yêu cầu cấp phát bộ nhớ bằng cách thủ công.

Chúng ta cần khai báo thư viện `<stdlib.h>`. Nếu bạn tin những gì tôi khuyên thì tốt nhất là nên khai báo thư viện này trong tất cả các chương trình của bạn. Thư viện này chứa 2 hàm mà chúng ta cần đến:

- `malloc` (“Memory allocation”, nghĩa là “cấp phát bộ nhớ động”): hàm sẽ yêu cầu hệ điều hành để được sử dụng bộ nhớ máy tính.
- `free` (“free”, nghĩa là “giải phóng”): hàm sẽ giải phóng vùng nhớ đã được hệ điều hành chỉ định cho yêu cầu cấp phát bộ nhớ của chúng ta trước đó, từ lúc này, các chương trình khác có thể tự do sử dụng vùng nhớ đó.

Mỗi khi muốn thực hiện việc cấp phát bộ nhớ động theo cách thủ công, bạn nên thực hiện lần lượt 3 bước sau:

1. Gọi hàm `malloc` để yêu cầu cấp phát bộ nhớ.
2. Kiểm tra giá trị trả về của hàm `malloc` để biết hệ điều hành có cấp phát bộ nhớ thành công hay không.
3. Sau khi sử dụng xong bạn phải tiến hành giải phóng bộ nhớ bằng hàm `free`. Nếu chúng ta không làm thao tác này, chương trình của bạn sẽ dễ gặp phải vấn đề tràn bộ nhớ, khi được hoàn tất chương trình của bạn sẽ chiếm một dung lượng bộ nhớ khổng lồ mà trong đó có những vùng nhớ bị sử dụng không cần thiết.

Ba bước này có gợi lại cho bạn những gì đã học về thao tác với tập tin không? Có đấy! Về nguyên tắc, nó gần như giống với những gì bạn đã học khi thao tác với các tập tin: đầu tiên nó sẽ được cấp phát bộ nhớ, sau đó kiểm tra xem việc cấp phát có thành công không, và khi đã được cấp phát nó sẽ sử dụng vùng bộ nhớ đó, sử dụng xong thì giải phóng vùng nhớ cho chương trình khác có thể sử dụng tiếp.

Hàm malloc: Yêu cầu cấp phát bộ nhớ.

Đây là prototype của hàm *malloc*, nó trông khá ngộ:

C code:

```
void* malloc(size_t soByteCansudung);
```

Hàm chỉ cần 1 tham số: đó là số byte cần sử dụng. Vì vậy, chỉ cần sử dụng *sizeof (int)* trong tham số này để dự trữ đủ không gian lưu trữ một biến kiểu *int*.

Nhưng hình như có gì đó hơi lạ về giá trị trả về của hàm: nó trả về 1 **void*** ...! Nếu các bạn còn nhớ về những gì đã học ở bài học về hàm (function), tôi đã từng nói với bạn void có nghĩa là trả về “không gì cả”, và chúng ta sử dụng kiểu void này để chỉ định hàm này sẽ không trả về giá trị nào hết.

Vậy ở đây chúng ta có 1 hàm sẽ trả về “con trỏ rỗng” sao? Điều này có ổn không? Xem ra các lập trình viên khá hài hước nhỉ.

Bạn cứ bình tĩnh, có lý do hết đấy. Thực tế là, hàm này sẽ trả về một con trỏ đến địa chỉ mà hệ điều hành đã cấp phát cho biến của bạn. Nếu hệ điều hành đã cấp cho biến của bạn ô nhớ ở địa chỉ 1600 thì nó sẽ trả về giá trị con trỏ tương ứng với địa chỉ 1600.

Vấn đề là hàm *malloc* không biết được kiểu biến mà bạn muốn tạo. Thực tế là bạn đưa cho hàm một tham số: số lượng byte cần sử dụng trong bộ nhớ. Nếu nó là 4 byte, nó có thể là biến kiểu *int* hoặc *long int* chẳng hạn.

Vì *malloc* không biết nên trả về giá trị kiểu nào, nó trả về kiểu *void**. Nó sẽ là một con trỏ đến bất kỳ kiểu nào cũng được. Chúng ta cũng có thể xem nó là một con trỏ linh động.

Nào chúng ta thực hành thôi.

Nếu tôi muốn vui vẻ một chút (e hèm!), tôi sẽ tự mình tạo một biến kiểu *int* trong bộ nhớ, tôi đang đề cập đến việc sử dụng *malloc* với *sizeof (int)* byte trong bộ nhớ.

Tôi nhận được kết quả của *malloc* là con trỏ đến *int*.

C code:

```
int* capphatBonho = NULL; // Tao mot con tro kieu int
```

```
capphatBonho = malloc(sizeof(int)); // Nhung dia chi danh cho con tro da duoc danh rieng
```

Kết thúc đoạn code trên, *capphatBonho* là một con đang chứa địa chỉ mà bạn muốn hệ điều hành dự trữ sẵn vùng nhớ cho bạn, vd như địa chỉ 1600.

Kiểm tra con trỏ.

Hàm *malloc* đã gọi một biến *capphatBonho* để chứa địa chỉ con trỏ mà hệ điều hành dành riêng cho bạn. Có 2 trường hợp có thể xảy ra:

- Nếu việc cấp phát thành công, con trỏ của chúng ta sẽ mang giá trị ứng với địa chỉ đã được máy tính chỉ định.
- Nếu việc cấp phát không thành công, con trỏ sẽ mang giá trị NULL.

Việc cấp phát bộ nhớ thất bại rất hiếm thấy nhưng nó có xảy ra, chẳng hạn bạn yêu cầu 34GB bộ nhớ RAM, khó mà yêu cầu máy tính đáp ứng cho bạn được.

Chúng ta sẽ gặp một hàm cơ bản mà bạn chưa từng thấy ở những bài trước: *exit ()*. Nó sẽ ngừng chương trình ngay lập tức. Nó cần một tham số: đó chính là giá trị mà chương trình phải trả về (chính xác hơn thì đây là giá trị trả về của hàm *main ()*).

C code:

```
int main (int argc, char *argv[ ])
{
    int* capphatBonho = NULL;

    capphatBonho = malloc(sizeof(int));
    if (capphatBonho == NULL) // Neu viec cap phat bo nho ko thanh cong
    {
        exit(0); // Chung ta se ngung chuong trinh ngay lap tuc
    }

    // Chuong trinh se tiep tuc hoat dong neu moi viec dien ra thuan loi

    return 0;
}
```

Nếu giá trị con trỏ khác NULL, chương trình có thể tiếp tục hoạt động, ngược lại nó sẽ hiển thị thông báo lỗi hoặc ngừng chương trình ngay lập tức. Nguyên nhân có thể là do dung lượng bộ nhớ bạn yêu cầu vượt quá khả năng của máy tính.

Hàm free: Giải phóng bộ nhớ.

Cũng giống như khi ta sử dụng *fclose* để đóng tập tin khi không còn sử dụng đến, chúng ta sẽ sử dụng hàm *free* để giải phóng những vùng bộ nhớ không còn cần sử dụng nữa.

C code:

```
void free(void* contro);
```

Chúng ta cần hàm *free* để giúp giải phóng các vùng địa chỉ của bộ nhớ.

Vì vậy chúng ta sẽ gửi cho nó tham số là những con trỏ, VD như con trỏ `capphatBonho` trong ví dụ lúc này.

Sau đây là toàn bộ chương trình của chúng ta từ đầu đến giờ.

Nhìn chung thì cách hoạt động cũng không có quá nhiều điểm khác biệt so với những gì ta đã học khi thao tác với tập tin.

C code:

```
int main (int argc, char *argv[ ])
{
    int* capphatBonho = NULL;

    capphatBonho = malloc(sizeof(int));
    if (capphatBonho == NULL) // chúng ta kiểm tra xem bộ nhớ đã được cấp phát chưa
    {
        exit(0); // Error: có lỗi và chương trình sẽ bị ngưng ngay lập tức
    }

    // Bộ nhớ đã được cấp phát và sẵn sàng để sử dụng
    free(capphatBonho); // Chúng ta không cần sử dụng bộ nhớ nữa, giải phóng nó thôi

    return 0;
}
```

Ví dụ cụ thể.

Chúng ta sẽ sắp xếp lại một số kiến thức bạn đã được học trước đây: Hỏi tuổi của người dùng và in nó ra.

Điều khác biệt duy nhất so với những gì chúng ta đã từng làm trước đó là lần này các biến sẽ được phân bổ một cách thủ công (còn gọi là cấp phát động) chứ không phải tự động như trước đây.

Vì vậy cho nên những dòng code nhìn sẽ phức tạp hơn. Nhưng các bạn hãy cố gắng hiểu được nó đi, điều này thật sự quan trọng đây:

C code:

```
int main (int argc, char *argv[ ])
{
    int* capphatBonho = NULL;

    capphatBonho = malloc(sizeof(int)); // Cap phat bo nho
    if (capphaBonho == NULL)
    {
        exit(0);
    }

    // Su dung bo nho
    printf ("Ban bao nhieu tuoi ? ");
    scanf ("%d", capphatBonho);
    printf ("Ban %d tuoi\n", *capphatBonho);

    free(capphatBonho); // Giai phong bo nho

    return 0;
}
```

Console:

```
Ban bao nhieu tuoi ? 69
Ban 69 tuoi
```



Hãy lưu ý: capphatBonho là một con trỏ, cách sử dụng nó khác với cách bạn làm việc với một biến bình thường. Để có được giá trị của con trỏ, bạn phải đặt dấu * trước capphatBonho (xem lại dòng code của *printf* để thấy rõ hơn). Trong khi đó, nếu muốn chỉ ra địa chỉ của con trỏ thì chỉ cần viết tên của nó capphatBonho (xem dòng *scanf* nhé).

Những điều này bạn đã được học trong bài về con trỏ rồi. Tuy nhiên, cần phải có một thời gian nhất định để bạn có thể quen với con trỏ, và khả năng nhầm lẫn vẫn rất cao. Trong trường hợp này bạn nên đọc lại một lần bài học về con trỏ, đây là những kiến thức cơ bản rất quan trọng.

Trở lại với đoạn code của chúng ta. Biến kiểu *int* đã được cấp phát động. Suy cho cùng những gì bạn đã viết cũng giống như trước đây khi tiến trình này diễn ra một cách tự động:

C code:

```
int main (int argc, char *argv[ ])
{
    int bienCuatoi = 0; // Cấp phát bộ nhớ (qua trình diễn ra hoán toán tự động)

    // Sử dụng bộ nhớ
    printf ("Ban bao nhiêu tuổi ? ");
    scanf ("%d", &bienCuatoi);
    printf ("Ban %d tuổi\n", bienCuatoi);

    return 0;
} // Giải phóng bộ nhớ (diễn ra tự động mỗi khi kết thúc hàm)
```

Tóm lại có 2 cách để tạo một biến, hay có thể nói là yêu cầu máy tính cấp phát bộ nhớ. Hai cách đó là:

- Tự động: Đây là những gì bạn đã từng học trước đây mỗi khi tạo ra một biến.
- Thủ công (cấp phát động): Cách mà tôi dạy cho bạn trong bài học này.



Tôi không hiểu sao chúng ta phải học cái cách phức tạp này và liệu có cần thiết không?

Đúng là có phức tạp hơn một chút ... nhưng nó có vô dụng không? Không! Đôi khi chúng ta buộc phải tự mình phân bổ bộ nhớ, tôi sẽ cho bạn thấy một trường hợp ngay sau đây.

Cấp phát động trong mảng.

Tới bây giờ chúng ta chỉ sử dụng phương pháp “cấp phát động” để tạo những biến bình thường đơn giản. Nhưng nhìn chung mọi người vẫn không thường dùng cách này, phương pháp tự động rõ ràng là đơn giản hơn nhiều.

Có phải bạn đang tự hỏi, vậy thì khi nào chúng ta nên dùng phương pháp này? Thường thì người ta sử dụng phương pháp “cấp phát động” khi tạo ra một mảng (array) mà ta không biết trước kích thước của nó khi chạy chương trình là bao nhiêu.

Ví dụ, tưởng tượng bạn sẽ tạo một chương trình lưu trữ tuổi những người bạn của bạn vào một mảng. Bạn có thể một mảng kiểu *int* để lưu trữ tuổi của họ, như sau:

C code:

```
int tuoiBanbe[15];
```

Nhưng ai dám đảm bảo với bạn rằng người dùng chỉ có 15 người bạn? Có thể nhiều hơn chứ. Khi viết chương trình, bạn không thể nào biết trước được kích thước nào là phù hợp cho mảng của bạn. Bạn chỉ có thể biết khi chương trình đã chạy, và bạn phải hỏi người dùng có bao nhiêu bạn bè.

Lợi ích của “cấp phát động” là ở đó: Chúng ta sẽ hỏi số lượng bạn bè của người dùng trước, và sau đó “cấp phát động” sẽ tạo ra mảng với kích thước chính xác như được yêu cầu (không quá nhỏ mà cũng không quá lớn). Nếu người dùng có 15 người bạn, chúng ta sẽ tạo một mảng với 15 giá trị *int*, tương tự nếu họ có 28 người bạn, nó sẽ tạo ra một mảng với 28 giá trị *int*, ...

Như tôi đã từng dạy bạn, ngôn ngữ C không thể tạo ra một mảng với kích thước được chỉ định bằng một biến:

C code:

```
int banbe[soluongBanbe];
```



Đoạn code trên có thể sẽ hoạt động với một số trình biên dịch trong một số trường hợp cụ thể, nhưng nó được khuyến cáo là không nên sử dụng như vậy.

Lợi ích của “cấp phát động” là nó cho phép ta tạo ra một mảng có kích thước khớp với số lượng bạn bè như biến *soluongBanbe*, dựa vào đó đoạn code có thể hoạt động ở bất kỳ đâu, với bất kỳ trình biên dịch nào.

Chúng ta sẽ dùng hàm *malloc* để yêu cầu cấp phát *soluongBanbe * sizeof(int)* byte trong bộ nhớ:

C code:

```
banbe = malloc(soluongBanbe * sizeof(int));
```

Đoạn code này sẽ giúp ta tạo ra một mảng kiểu *int* với kích thước đúng theo như số lượng bạn bè của người dùng.

Sau đây là những gì mà chương trình sẽ lần lượt thực hiện:

1. Yêu cầu người dùng cho biết họ có bao nhiêu bạn bè.
2. Tạo ra một mảng có kích thước bằng với số lượng bạn bè của người dùng (bằng cách sử dụng *malloc*)
3. Lần lượt hỏi tuổi của từng người bạn và lưu trữ vào trong mảng.
4. Hiển thị tất cả tuổi đã được lưu vào mảng trước đó.
5. Cuối cùng, vì chúng ta không còn cần đến mảng tuổi của những người bạn này nữa, ta sẽ giải phóng bộ nhớ bằng hàm *free*.

C code:

```
int main (int argc, char *argv[ ])
{
    int soluongBanbe = 0, i = 0;
    int* tuoiBanbe = NULL; // con tro nay se duoc su dung nhu mot mang sau khi dung malloc

    // Chung ta se yeu cau nguoi dung cho biet so luong ban be cua ho
    printf ("Ban co bao nhieu nguoi ban ? ");
    scanf ("%d", &soluongBanbe);

    if (soluongBanbe > 0) // Phai co it nhat mot nguoi ban (xin chia buon neu ko co ai ^^!)
    {
        tuoiBanbe = malloc(soluongBanbe * sizeof(int)); // Phan phoi bo nho cho mang
        if (tuoiBanbe == NULL) // Kiem tra xem viec cap phat bo nho co thanh cong ko?
        {
            exit(0); // Chuong trinh ngung lai ngay lap tuc
        }

        // Yeu cau nhap tuoi tung nguoi ban
        for (i = 0 ; i < soluongBanbe ; i++)
        {
            printf ("Nguoi ban thu %d bao nhieu tuoi ? ", i + 1);
            scanf ("%d", &tuoiBanbe[i]);
        }

        // Lan luot hien thi tuoi cua ban be
        printf ("\n\nTuoi cua ban be ban la :\n");
        for (i = 0 ; i < soluongBanbe ; i++)
        {
            printf ("%d tuoi\n", tuoiBanbe[i]);
        }

        // Giai phong bo nho da duoc cap phat cho mang boi malloc, no ko con can thiet nua.
        free(tuoiBanbe);
    }

    return 0;
}
```


Console:

```
Ban co bao nhieu nguoi ban ? 5
Nguoi ban thu 1 bao nhieu tuoi ? 18
Nguoi ban thu 2 bao nhieu tuoi ? 19
Nguoi ban thu 3 bao nhieu tuoi ? 20
Nguoi ban thu 4 bao nhieu tuoi ? 21
Nguoi ban thu 5 bao nhieu tuoi ? 22

Tuoi cua ban be ban la :
18 tuoi
19 tuoi
20 tuoi
21 tuoi
22 tuoi
```

Chương trình có vẻ không ứng dụng được gì nhiều trong thực tế nhưng tôi chọn làm nó vì nó khá đơn giản để các bạn có thể hiểu được cách làm việc của hàm *malloc*.

Tôi đảm bảo với bạn rằng ở những bài học sau, chúng ta sẽ có cơ hội sử dụng hàm *malloc* để lưu trữ những thứ thú vị hơn là tuổi của bạn bè người dùng.

Tổng kết:

- Một biến sẽ chiếm không gian bộ nhớ nhiều hay ít là tùy thuộc vào kiểu của nó (int hay double hay char ...)
- Người ta có thể biết được số byte mà kiểu biến đó sẽ chiếm trong bộ nhớ bằng cách sử dụng *sizeof* ().
- Cấp phát động là hành động dự trữ vùng nhớ cho biến hoặc mảng.
- Việc phân bổ bộ nhớ được thực hiện với hàm *malloc* () và điều quan trọng đừng nên quên đó là hãy nhớ giải phóng bộ nhớ bằng hàm *free* () ngay sau khi bạn không cần đến vùng nhớ đó nữa.
- Đặc biệt cấp phát động cho phép tạo ra mảng có kích thước được xác định bởi một biến trong khi chạy chương trình.

Bài 9: Test Program

Trò chơi người treo cổ

Chỉ nói ko thôi thì chưa đủ: Chắc hẳn là các bạn hiểu rõ tầm quan trọng của việc thực hành đúng không? Tôi chắc chắn việc thực hành thật sự cần thiết đối với bạn, chúng ta vừa được học rất nhiều những khái niệm, lý thuyết ... và bất cứ điều gì bạn được đọc, những gì bạn nói ra, bạn sẽ không bao giờ thực sự hiểu sâu về nó cho đến khi bạn bắt tay vào thực hành.

Trong lần học này, tôi đề nghị chúng ta sẽ làm ra trò chơi “Người treo cổ”. Đây là một trò chơi cổ điển quen thuộc về các từ (bạn nào xài kim từ điển thì biết ngay trò này), bạn sẽ phải đoán những chữ cái bị ẩn trong các từ. Trò người treo cổ lần này sẽ được chơi theo kiểu của ngôn ngữ C trên màn hình console.

Mục đích chính của chúng ta là giúp bạn có thể nắm vững tất cả những kiến thức đã được học trước giờ. Những con trỏ (pointer), chuỗi ký tự (string), tập tin (file), mảng (array), cấu trúc (structure)... ok, tất cả sẽ ổn thôi.

Một số chỉ dẫn.

Tôi muốn nói một chút về những nguyên tắc hoạt động của trò “Người treo cổ”. Bây giờ tôi sẽ đưa cho bạn một số chỉ dẫn, đồng thời tôi nghĩ sau này bạn cũng nên giải thích cho người khác biết cách hoạt động của trò chơi mà bạn tạo ra.

Đa phần chúng ta đều đã từng biết qua trò “người treo cổ” rồi đúng ko? Nhưng mà bây giờ nhắc lại một chút về cách chơi cũng chả mất gì đúng ko: Mục tiêu của trò chơi là tìm ra từ bị ẩn sau tối đa 10 lần đoán (bạn cũng có thể tự mình thay đổi số lần đoán tối đa để tùy chỉnh độ khó).

Quy cách chơi.

Giả sử từ khóa bị ẩn là RED.

Bạn đoán chữ A và máy tính sẽ kiểm tra xem chữ A có nằm trong từ đang bị ẩn ko.



Hãy nhớ là có sẵn một hàm trong thư viện *string.h* có thể tìm một chữ cái trong một từ ! (hàm `strchr`). Tuy nhiên, bạn không cần sử dụng nó (bản thân tôi cũng ít khi dùng đến).

Có 2 khả năng sẽ xảy ra:

- Chữ cái bạn đoán có chứa trong từ bị ẩn, lúc này màn hình sẽ hiển thị chữ mà bạn tìm thấy trong từ bị ẩn đó.
- Chữ cái bạn đoán không có trong từ đang bị ẩn (chẳng hạn như khi này bạn đoán chữ A nhưng từ bị ẩn là RED thì làm gì có chữ A): Máy tính sẽ thông báo cho người chơi là chữ vừa được đoán không có chứa trong từ đang bị ẩn và tự động trừ bớt đi một lần đoán của họ. Cho tới khi chúng ta sử dụng hết tất cả các lượt đoán mà vẫn chưa tìm ra từ bị ẩn thì xác định là thua cmnr!!! Đến đây thì GAME OVER.



Bạn cũng biết trong thực tế khi chơi trò này trên máy tính hoặc kim từ điển thì mỗi lần chúng ta đoán sai, hình ảnh một người bị treo cổ sẽ được vẽ thêm vào 1 nét cho đến khi bạn đoán đúng từ bí ẩn hoặc đoán sai hết thì hình vẽ hoàn thành (cũng có nghĩa là bị treo cổ và thua). Với console thì chúng ta ko thiết kế sinh động như vậy được và chỉ hiển thị chữ cái được thôi, vì vậy có gì xài nấy, chúng ta sẽ hiển thị một câu thông báo “Bạn còn xxx lần đoán trước khi người này bị treo cổ” mỗi khi đoán sai cho đến khi hết lượt đoán.

Giả sử người chơi đoán chữ D (trong trường hợp từ bí ẩn là RED). Rõ ràng là chữ D được chứa trong từ RED, vì vậy số lượt đoán của người chơi sẽ không bị giảm đi. Màn hình sẽ hiển thị từ bí ẩn kèm theo những từ bạn đã đoán đúng, như sau:

Console:

```
Tu bí an: **D
```

Và nếu sau đó người ta đoán chữ R, lại tiếp tục có chữ R, màn hình sẽ hiển thị một lần nữa từ bí ẩn kèm theo những từ bạn đã đoán đúng, như sau:

Console:

```
Tu bí an: R*D
```

Trường hợp từ chứa nhiều ký tự giống nhau?

Trong một số từ, sẽ có thể xuất hiện 2 hoặc nhiều chữ cái giống nhau. VD: có 2 chữ R trong từ PROGRAM hoặc có 3 chữ E trong EXCELLENT.

Vậy điều gì sẽ xảy ra trong những trường hợp trên? Luật của trò này là: nếu người chơi đoán đúng chữ E trong từ EXCELLENT thì màn hình sẽ hiển thị cả 3 chữ cùng một lúc:

Console:

```
Tu bí an: E**E**E**
```

Người chơi sẽ không cần phải gõ chữ E 3 lần.

Xem thử trò “người treo cổ” hoàn chỉnh.

Sau đây là những gì nên được tạo ra và bạn sẽ thấy chúng xuất hiện trong trò chơi mà bạn sẽ viết:

Console:

```
Chao mung ban tham gia tro choi NGUOI TREO CO!
```

```
Ban co 10 luot doan.
```

```
Tu bi an la gi? *****
```

```
Hay doan mot chu cai: Z
```

```
Ban co 9 luot doan.
```

```
Tu bi an la gi? *****
```

```
Hay doan mot chu cai: A
```

```
Ban co 9 luot doan.
```

```
Tu bi an la gi? *A*****
```

```
Hay doan mot chu cai: K
```

```
Ban co 9 luot doan.
```

```
Tu bi an la gi? *A*****K
```

```
Hay doan mot chu cai:
```

Và cứ như vậy cho tới khi người chơi đã tìm được từ bí ẩn (hoặc vượt quá số lượt đoán cho phép):

Console:

```
Ban co 3 luot doan.
```

```
Tu bi an la gi? FACEB**K
```

```
Hay doan mot chu cai: O
```

```
Xin chuc mung! Tu bi mat la: FACEBOOK
```

Nhập chữ cái vào màn hình console.

Đọc một chữ cái trên màn hình console có lẽ là phức tạp hơn bạn thấy.

Bằng chút trực giác, để có thể nhập một chữ cái bạn nên nghĩ về:

C code:

```
scanf ("%c", &chuCaiBiMat);
```

Và thực tế thì đây là một ý tưởng tốt, **%c** sẽ được thay bằng một chữ cái, chúng ta sẽ lưu trữ trong biến *chuCaiBiMat* (tất nhiên đây phải là một biến kiểu char).

Tất cả mọi việc đang diễn ra rất tốt ... miễn là chúng ta không thay đổi *scanf*. Bạn có thể thử chạy đoạn mã sau:

C code:

```
int main (int argc, char* argv[ ])
{
    char chuCaiBiMat = 0;

    scanf ("%c", &chuCaiBiMat);
    printf ("%c", chuCaiBiMat);

    scanf ("%c", &chuCaiBiMat);
    printf ("%c", chuCaiBiMat);

    return 0;
}
```

Thường thì theo những gì chúng ta đã được học bạn sẽ hiểu đoạn code trên yêu cầu người dùng nhập vào một chữ cái 2 lần sau đó in chúng ra 2 lần.

Ok bạn chạy thử nó đi! Oh, chuyện gì đang xảy ra vậy? Bạn nhập một chữ cái, ok, nhưng ... chương trình ngừng ngay sau khi bạn mới nhập chữ cái một lần, còn một lần nhập nữa cơ mà, chương trình đã không yêu cầu bạn nhập tiếp chữ cái tiếp theo. Tại sao vậy, sao nó lại bỏ qua chữ cái thứ 2 trong khi đoạn code trên rõ ràng dùng hàm *scanf* 2 lần để yêu cầu nhập chữ cái 2 lần cơ mà.



Chuyện gì đã xảy ra?

Thật ra thì, khi bạn nhập một chữ cái ở màn hình console, tất cả mọi thứ bạn gõ vào đã được lưu trữ ở đâu đó trong bộ nhớ, và nó cũng có bao gồm cả ký tự ENTER (\n).

Vì vậy khi bạn nhập ký tự đầu tiên (VD: ký tự A) và bạn bấm ENTER, ký tự A này được trả về bởi hàm *scanf* đầu tiên, sau đó *scanf* tiếp tục trả về ký tự đặc biệt **\n** tương ứng với phím ENTER mà bạn đã bấm.

Để tránh tình trạng này, tốt hơn hết là hãy tạo một hàm (function) của chúng ta *docKytu()*:

C code:

```
char docKytu ( )
{
    char kytuNhapVao = 0;

    kytuNhapVao = getchar ( ); // Doc ky tu duoc nhap dau tien
    kytuNhapVao = toupper (kytuNhapVao); // Viet hoa ky tu do

    // Lan luot doc tiep cac ky tu khac cho den khi gap \n
    while (getchar ( ) != '\n') ;

    return kytuNhapVao; // Tra ve ky tu dau tien doc duoc
}
```

Hàm này sử dụng *getchar ()* đây là một hàm cơ bản có chứa trong thư viện *stdio* cũng như *scanf* ("*%c*" &*kytu*);



Hàm *getchar* sẽ đọc dữ liệu được nhập vào, chỉ một ký tự tại một thời điểm từ bàn phím. Khi sử dụng hàm này, các ký tự nằm trong vùng đệm cho đến khi người dùng nhấn phím xuống dòng. Vì vậy nó sẽ đợi cho đến khi phím Enter được gõ. Hàm *getchar()* không có tham số, nhưng vẫn phải có các cặp dấu ngoặc đơn. Nó đơn giản là lấy về ký tự tiếp theo và sẵn sàng đưa ra cho chương trình. Chúng ta nói rằng hàm này trả về một giá trị có kiểu char.

Sau đó, chúng ta sử dụng một hàm tiêu chuẩn mà bạn chưa có cơ hội được học ở những bài trước: hàm *toupper ()* giúp viết hoa ký tự. Bằng cách đó, trò chơi vẫn sẽ hoạt động dù cho người chơi có nhập vào một ký tự không viết hoa. Chúng ta sẽ phải khai báo thư viện *#include ctype.h* ở đầu để có thể sử dụng hàm này (đừng quên nhé).

Và bây giờ là phần thú vị nhất: Chúng ta sẽ xóa đi tất cả những ký tự mà người dùng đã nhập để làm trống bộ nhớ. Việc gọi hàm *getchar* sẽ lấy một ký tự tiếp theo như tôi đã giải thích cho bạn ở trên. Giả sử khi người dùng nhấn Enter thì ký tự mà hàm lấy vào sẽ là **\n**.

Những gì chúng ta phải làm vô cùng đơn giản: chỉ cần một vòng, chúng ta gọi hàm *getchar* trong vòng lặp cho tới khi gặp ký tự **\n** thì ngừng vòng lặp. Sau khi vòng lặp kết thúc, nó đã đọc tới ký tự **\n** khi người dùng nhấn Enter, điều đó cũng có nghĩa là hàm *getchar* đã đọc hết những ký tự được người dùng nhập vào trước đó, nhờ đó vùng đệm đã được làm trống theo đúng phương thức hoạt động của hàm *getchar* mà tôi đã nói với bạn ở trên.



Sao lại xuất hiện dấu chấm phẩy ở vị trí kết thúc dòng chứa vòng lặp while và chúng ta cũng không thấy cặp ngoặc nhọn của vòng lặp như ta đã học ở các bài trước.

Như bạn đã thấy, vòng lặp while của chúng không chứa những câu lệnh (instructions), nó chỉ chứa duy nhất hàm *getchar* trong phần điều kiện. Trong trường hợp này, sử dụng cặp ngoặc nhọn cho vòng lặp là không cần thiết, vì vậy tôi đã sử dụng dấu chấm phẩy để thay cho cặp dấu `{ }`. Dấu chấm phẩy này cũng có nghĩa là “Không cần làm gì mỗi lần qua vòng lặp”.

Đây là một chút kỹ thuật đặc biệt được sử dụng bởi rất nhiều lập trình viên và tôi nghĩ bạn cần phải biết để ứng dụng cho những vòng lặp cực ngắn và đơn giản.

Nhưng nếu lỡ bạn không biết kỹ thuật đặc biệt trên thì vòng lặp trong code của bạn sẽ như sau:

C code:

```
while (getchar( ) != '\n')  
{  
  
}
```

Không có gì trong cặp dấu ngoặc nhọn là hoàn toàn bình thường, bởi vì trong trường hợp này chúng ta thật sự không có câu lệnh nào cần thực hiện cả. Việc sử dụng dấu chấm phẩy thay thế cho cặp dấu ngoặc nhọn chỉ giúp code của bạn gọn gàng hơn thôi.

Cuối cùng, hàm *docKytu* sẽ trả về ký tự đầu tiên nó đọc được, cũng chính là giá trị của biến *kytu*.

Tóm lại, để có được một từ trong code của bạn, chúng ta sẽ không sử dụng:

C code:

```
scanf ("%c", &chuCaiBiMat);
```

Thay vào đó chúng ta sẽ dùng một cách cao cấp hơn:

C code:

```
chuCaiBiMat = docKytu( );
```

Danh mục từ bí ẩn:

Bắt đầu chương trình test của bạn, tôi sẽ yêu cầu bạn đặt những từ bí mật trực tiếp vào những dòng code. Ví dụ như sau:

C code:

```
char tuBimat [ ] = "LOVE"
```

Nếu chúng ta làm như trên, dĩ nhiên là tất cả các từ bí mật của người chơi sẽ giống như nhau, và chả có gì vui cả đúng không. Tôi yêu cầu bạn làm vậy lúc đầu chỉ để giúp bạn tránh gặp phải những vấn đề rắc rối thôi (chỉ vào thời điểm đó thôi nhé). Sau khi bạn chắc rằng trò chơi đã hoạt động đúng như ý bạn muốn, chúng ta đã có thể thoải mái cải tiến nó sang giai đoạn hai: chúng ta sẽ tạo ra cả một danh mục đầy những từ bí mật khác nhau.



Cái gì gọi là “danh mục các từ bí mật”?

Nó chính xác là một tập tin chứa rất nhiều từ bí mật cho trò “Người treo cổ” của bạn. Và chắc chắn là mỗi từ sẽ nằm trên 1 dòng. VD như sau:

```
LOVE  
MONEY  
PROGRAM  
FUNCTION  
POINTER  
LOOP  
INSTRUCTION  
STRING  
VARIABLE  
CONSTANT  
DEFINE
```

Mỗi lần trò chơi bắt đầu, chương trình của bạn sẽ mở tập tin này lên và chọn ngẫu nhiên một từ trong danh sách. Bằng cách này, bạn sẽ có một tập tin riêng biệt để có thể chỉnh sửa theo ý bạn muốn và thêm vào những từ bí mật mới cho trò “Người treo cổ” thêm thú vị.



Có thể bạn đã nhận ra rằng từ đầu đến giờ những từ bí mật trong trò chơi đều được tôi viết hoa toàn bộ. Chúng ta sẽ không phân biệt giữa những ký tự viết hoa và viết thường, vậy nên tốt nhất là nên nói rõ ngay từ đầu “tất cả các từ sẽ được viết hoa”. Bạn có thể thông báo trước cho người chơi ở phần hướng dẫn, yêu cầu họ nhập ký tự viết hoa chứ không phải chữ thường.\

Hơn nữa, chúng ta cũng bỏ qua những trường hợp từ có dấu để đơn giản hóa trò chơi (nếu chương trình test của chúng ta có những từ có dấu như vậy thì tới giờ này nó vẫn chưa xong đâu). Bạn sẽ cần phải viết những từ bí mật vào tập tin danh mục từ bí mật và chúng phải được viết hoa không dấu.

Có một vấn đề là máy tính sẽ hỏi bạn, có bao nhiêu từ bí mật trong danh mục. Thật vậy, nếu bạn muốn chọn ngẫu nhiên một từ, sẽ có rất nhiều lựa chọn giữa từ mang số thứ tự 0 với một số thứ tự ngẫu nhiên bất kỳ, và máy tính làm sao biết số bất kỳ đó là bao nhiêu.

Để giải quyết vấn đề đó, có hai giải pháp. Bạn có thể định nghĩa số từ chứa trong danh mục ngay dòng đầu tiên của tập tin:

```
3
LOVE
MONEY
LIFE
```

Tuy nhiên, cách này hơi nhàm chán, bởi vì chúng ta sẽ phải tự mình cập nhật lại thông số mỗi lần thêm vào 1 từ trong danh mục (cách này vẫn còn 1 chút bất tiện ở điểm này). Vì vậy tôi gợi ý các bạn có thể tạo ra chức năng tự động đếm số từ có chứa trong tập tin danh mục từ bí ẩn cho chương trình của bạn. Cũng đơn giản thôi, ý tưởng liên quan đến việc đếm ký tự `\n` mỗi dòng trong tập tin.

Mỗi lần chương trình đọc tập tin, nó sẽ đếm ký tự `\n` lại từ đầu. Sau đó bạn sẽ lấy kết quả đếm được đó thế vào con số mà chương trình sẽ chọn ngẫu nhiên giữa số 0 và nó để chọn ra từ sẽ được lưu vào bộ nhớ khi trò chơi bắt đầu.

Tôi nhường cho bạn suy nghĩ về giải pháp này đây. Tôi sẽ không giúp bạn thêm nữa nếu không đây đâu còn là chương trình test của bạn nữa. Hãy sử dụng tất cả những kiến thức mà bạn đã được học, bạn hoàn toàn có thể thiết kế trò chơi này theo như tôi đã nói. Có thể sẽ mất ít nhiều thời gian cũng như bạn sẽ cảm thấy dễ hoặc khó nhưng hãy cố gắng tự mình sắp xếp, tổ chức mọi thứ theo cách của bạn (nhớ tạo đủ function cho những gì bạn muốn chương trình làm nhé), rồi bạn cũng sẽ làm được thôi.

Chúc may mắn, và hơn nữa là “Đừng nản chí” !!!

Giải pháp thứ 1: Mã số trò chơi

Khi bạn đang đọc những dòng này, có thể là bạn đã hoàn tất chương trình hoặc cũng có thể là không.

Bản thân tôi đã mất nhiều thời gian hơn mình nghĩ để biến trò chơi này trông có vẻ khá ngu ngốc. Nó thường là như vậy: nhìn qua thì ta thường nghĩ “ui xời, dễ ợt” nhưng thực tế thì có một số trường hợp khá phức tạp cần phải giải quyết.

Nhưng tôi đã nói rồi, bạn vẫn có thể tự mình giải quyết được hết, chỉ là mất ít nhiều thời gian hơn thôi. Đây đâu phải là một cuộc chạy đua, bạn cứ thoải mái ra, dành thêm chút thời gian suy nghĩ trước khi tìm đến phần giải pháp này, thử suy nghĩ thêm 5 phút nữa xem sao, biết đâu lại nảy ra được ý tưởng gì đó.

Đừng có nghĩ rằng tôi chỉ việc đặt đít xuống một chút là đã viết xong chương trình. Tôi cũng như các bạn thôi, cũng phải bắt đầu từ những gì đơn giản nhất và cải tiến chúng từ từ để có được kết quả cuối cùng như mình muốn.

Tôi cũng đã từng quên những kiến thức cơ bản như: quên khởi tạo giá trị khi khai báo biến, quên đặt các nguyên mẫu hàm (prototype) hoặc xóa đi những biến không còn giá trị sử dụng trong chương trình để giải phóng bộ nhớ. Thậm chí, tôi phải thú nhận với bạn là tôi còn quên cả việc đặt dấu chấm phẩy ở cuối câu lệnh (instruction).

Tôi nói với bạn những điều trên để làm gì? Tôi không hoàn hảo, và cuộc sống cũng như vậy, luôn có những sai lầm. “LẬP TRÌNH CŨNG NHƯ VẬY ... BẠN SẼ TIẾP TỤC VỚI NÓ CHỨ!!! YES OR NO ???”

Ok, bây giờ tôi sẽ cho bạn thấy giải pháp qua 2 giai đoạn:

1. Đầu tiên tôi sẽ cho các bạn thấy làm thế nào để chương trình xử lý các ký tự được ẩn sau mỗi lượt đoán của người dùng. Tôi chọn từ FACEBOOK vì nó cho phép tôi kiểm tra xem mình có giải quyết được trường hợp có 2 ký tự giống nhau trong một từ không.
2. Sau đó tôi sẽ cho bạn thấy làm thế nào để quản lý tập tin danh mục từ bí ẩn để thêm vào các từ bí ẩn cho trò chơi.

Dù có thể nhưng tôi sẽ không viết toàn bộ code cho bạn thấy một lần. Sẽ rất dài và nhìn khá kinh khủng, tôi sợ sẽ làm các bạn bị choáng.

Tôi sẽ cố gắng giải thích lý do tại sao tôi không làm vậy. Hãy khoan quá khao khát đạt được kết quả bạn muốn, mà quan trọng là cách tư duy của chúng ta về vấn đề đó. Hãy tập trung phân tích vào cách giải quyết vấn đề trước nhé.

Phân tích hàm main:

Như mọi người vẫn thường nói rằng “mọi thứ đều bắt đầu từ đôi tay”. Hehe. Đừng quên thêm vào những thư viện *stdio*, *stdlib*, *ctype* (thư viện này để sử dụng hàm *toupper* () nhé). Chúng thật sự hữu ích và cần thiết cho chương trình của bạn đây:

C code:

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

int main(int argc, char* argv[ ])
{

    return 0;
}
```

Ok, đến lúc này thì chúng ta nên làm theo tôi. Rồi chúng ta sẽ thấy được sự hữu ích của những thư viện, chúng ta sẽ có thể quản lý chương trình và hầu hết các function trong chương trình sẽ cần đến những thư viện này.

Nào, hãy bắt đầu bằng việc khai báo những biến cần thiết. Hãy yên tâm, tôi đã không nghĩ hết về tất cả những biến cần khai báo cho chương trình cùng một lúc, ít ra thì vẫn ít hơn lần đầu tiên tôi tập tành viết code:

C code:

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

int main(int argc, char* argv[ ])
{
    char kytu = 0; // Biến này sẽ lưu trữ ký tự của người chơi (được tra về bởi hàm scanf)
    char tuBimat[ ] = "FACEBOOK"; // Đây là từ bí mật cần tìm ra
    int sokytuBimat[8] = {0}; /* Một mảng có chứa các thành phần dạng Boolean. Mỗi ô trong
    mảng sẽ tương ứng với một ký tự người chơi sẽ đoán. Nếu đoán đúng thì giá trị = 1, và nếu sai thì
    giá trị = 0 */
    int soluoatDoan = 10; // Số lượt đoán còn lại của người chơi (0 = thua)
    int i = 0; // Biến hỗ trợ

    return 0;
}
```

Tôi đã cố tình khai báo mỗi biến trên một dòng riêng biệt và ghi chú thêm cho từng biến để các bạn dễ hiểu. Trong thực tế, bạn không bắt buộc phải ghi chú tất cả ra như thế và có thể khai báo tất cả những biến cùng kiểu trên cùng một dòng.

Các biến trên được khai báo khá hợp lý đúng không: biến *kytu* để lưu trữ ký tự người chơi đoán được, biến *tuBimat* để lưu trữ từ bí mật của chúng ta, biến *soluoatDoan* để đếm số lượt đoán còn lại của người chơi (lượt đoán = 0 nghĩa là thua) ...

Biến *i* là biến hỗ trợ tôi làm việc với mảng, vòng lặp, nó sẽ giúp bạn kết thúc vòng lặp.

Cuối cùng, biến mà bạn cần phải nghĩ đến nhất, thứ khác biệt với những biến còn lại, mảng *sokytuBimat* chứa các thành phần kiểu Boolean. Để ý thì bạn sẽ thấy tôi đã quy định kích thước sẵn cho nó đúng với số ký tự của từ bí ẩn (là 8 ký tự). Không phải một số ngẫu nhiên nhé: mỗi một ô thành phần trong mảng đại diện cho một ký tự của từ bí mật. Ô đầu tiên đại diện cho ký tự đầu tiên, ô thứ 2 là ký tự thứ 2 và cứ thế cho đến ký tự cuối.

Giá trị của các thành phần trong mảng được khởi tạo lúc đầu bằng 0, điều này cũng có nghĩa là “không tìm thấy ký tự nào”. Và theo tiến trình hoạt động của game, giá trị này sẽ được thay đổi tương ứng với diễn biến trò chơi. Mỗi một khi có ký tự được tìm đúng, giá trị của ô thành phần trong mảng *sokytuBimat* sẽ mang giá trị 1, ngược lại là giá trị 0.

Ví dụ, nếu đến một lúc người chơi tìm được đến ***A*E*OO*** thì lúc này mảng của chúng ta sẽ có các giá trị thành phần là **01010110** (giá trị 1 cho mỗi ký tự tìm đúng).

Khá dễ để biết khi nào thì chúng ta thắng trò chơi đúng không. Đó là lúc các giá trị Boolean chỉ toàn là giá trị 1. Ngược lại nếu đoán sai thì giá trị sẽ là 0.

Ok, tiếp tục thôi:

C code:

```
printf("Chao mung den voi tro choi Nguoi treo co !\n\n");
```

Đơn giản chỉ là một lời chào mừng thôi mà. Hihihi. Nhưng vòng lặp sau đây mới là điều thú vị:

C code:

```
while (soluotDoan > 0 && !win (sokytuBimat))  
{
```

Trò chơi sẽ tiếp tục nếu *soluotDoan* vẫn lớn hơn 0 hoặc chưa thắng. Nếu không còn lượt đoán nào hoặc chiến thắng trò chơi, trong cả 2 trường hợp này, ngừng trò chơi ngay lập tức, và vòng lặp cũng sẽ được kết thúc.

win là một function được phân tích dựa vào mảng *sokytuBimat*. Nó trả về giá trị “true” tương ứng với giá trị 1 nếu người chơi chiến thắng (mảng *sokytuBimat* chỉ chứa toàn giá trị 1), “false” tương ứng giá trị 0 nếu người chơi thua.

Tôi không thể giải thích chi tiết cho bạn về function *win* ngay bây giờ nhưng chúng ta sẽ nói về nó sau ít phút nữa. Bây giờ bạn tạm thời chỉ cần biết chức năng mà nó sẽ làm trong chương trình.

Xem thử nhé:

C code:

```
printf("\n\n Ban co %d luot doan de choi", soluotDoan);  
printf("\n Tu bi mat la gi ?");  
  
/* Hien thi nhung ky tu bi mat va an di nhung ky tu chua duoc tim thay  
Vi du: *A***OO*/  
for (i = 0 ; i < 6 ; i++)  
{  
    if (sokytuBimat[i]) // Neu nguoi choi tim duoc ky tu thu i  
        printf("%c", tuBimat[i]); // Hien thi ky tu thu i duoc tim thay  
    else  
        printf("*"); // Hien thi dau * doi voi nhung ky tu chua duoc tim thay  
}
```

Phân tích đoạn code trên, chương trình sẽ hiển thị từng ký tự tìm được và phần còn lại của từ bí mật (phần này được ẩn đi bởi những dấu *). Mỗi khi vòng lặp kết thúc, màn hình sẽ hiển thị những ký tự chưa được tìm thấy bằng dấu * và đồng thời phân tích xem ký tự được người chơi đoán có chứa trong mảng *sokytuBimat* hay không (dựa vào (*if sokytuBimat [i]*)). Nếu ký tự được tìm thấy có chứa trong mảng đó thì hiển thị lên, ngược lại thì hiển thị dấu * để ẩn đi.

Bây giờ chúng ta đã có những thứ cần hiển thị rồi, hãy yêu cầu người chơi nhập ký tự họ đoán thôi:

C code:

```
printf ("\n Xin moi ban doan mot ky tu: ");  
kytu = docKytu();
```

Tôi đã gọi hàm *docKytu ()* của chúng ta. Nó đọc các ký tự đầu tiên được nhập vào, viết hoa nó lên và làm trống bộ nhớ đệm.

C code:

```
// Neu ky tu nhap vao khong dung  
if (!kiemtraKytu (kytu, tuBimat, sokytuBimat))  
{  
    soluotDoan--; // Giam bot mot lan doan cua nguoi choi  
}  
}
```

Chúng ta kiểm tra xem ký tự người chơi nhập vào có chứa trong từ bí mật không. Đoạn code trên gọi hàm *kiemtraKytu* để thực hiện điều đó và tí nữa bạn sẽ thấy cụ thể hàm đó trông như thế nào.

Tạm thời bây giờ những gì chúng ta cần biết là hàm này trả về giá trị “true” nếu ký tự được nhập vào có chứa trong từ bí mật, ngược lại thì trả về giá trị “false”.

Các bạn có để ý *if* của chúng ta bắt đầu bằng một dấu chấm than, điều này có nghĩa là “không”. Vậy bây giờ điều kiện của chúng ta sẽ được hiểu là “Nếu không tìm thấy ký tự”.

Và điều gì sẽ xảy ra? Nó sẽ giảm bớt một lượt đoán của người chơi.



Lưu ý rằng hàm *kiemtraKytu* cũng sẽ đặt giá trị vào các ô trong mảng *sokytuBimat*. Nó sẽ đặt giá trị 1 vào những ô nào chứa ký tự đúng được tìm thấy.

Vòng lặp chính của trò chơi cần phải ngừng lại. Vì vậy chúng ta sẽ quay lại đầu vòng lặp và kiểm tra xem người chơi còn lượt đoán nào không và liệu rằng họ đã thắng hay chưa.

Khi ra khỏi vòng lặp chính của trò chơi, chương trình sẽ kiểm tra xem liệu người chơi đã thắng hay chưa trước khi chương trình ngừng lại.

C code:

```
if (win(sokytuBimat))  
    printf ("\n\n Chuc mung, ban da chien thang ! Tu bi mat la : %s", tuBimat);  
else  
    printf ("\n\n Xin chia buon, ban da thua !\n\n Tu bi mat la : %s", tuBimat);  
  
return 0;  
}
```

Chúng ta dùng hàm *win* để kiểm tra xem người chơi có thắng hay không để hiển thị thông báo cho họ biết.

Phân tích hàm win:

Bây giờ chúng ta sẽ thấy code của hàm *win*:

C code:

```
int win(int sokytuBimat[ ])
{
    int i = 0;
    int nguoiChoiChienThang = 1;

    for (i = 0 ; i < 6 ; i++)
    {
        if (sokytuBimat[i] == 0)
            nguoiChoiChienThang = 0;
    }

    return nguoiChoiChienThang;
}
```

Hàm này lấy tham số là mảng *sokytuBimat* chứa giá trị dạng Boolean. Hàm sẽ trả về một giá trị “true” nếu người chơi thắng, hoặc giá trị “false” nếu không thắng.

Code của hàm này nhìn khá đơn giản đúng không, chắc là các bạn đều đọc hiểu đúng không. Chúng ta sẽ kiểm tra thử xem trong mảng *sokytuBimat* có ô nào chứa giá trị 0 không. Nếu xuất hiện bất kỳ ô nào trong mảng có giá trị 0, điều đó có nghĩa là người chơi không thắng được, lúc này biến *nguoiChoiChienThang* (biến kiểu Boolean) sẽ mang giá trị “false” tương đương bằng 0. Và nếu tất cả các ký tự đều được tìm thấy, biến này sẽ có giá trị “true” tương đương bằng 1, và hiển nhiên hàm này cũng sẽ trả về giá trị 1 luôn.

Phân tích hàm *kiemtraKytu*:

Hàm *kiemtraKytu* có 2 nhiệm vụ chính:

- Trả về một Boolean chỉ ra rằng ký tự do người dùng đoán có chứa trong từ bí mật hay không.
- Gửi giá trị (vd: giá trị 1) vào các ô giá trị của mảng *sokytuBimat* tương ứng với vị trí của ký tự đó trong mảng.

C code:

```
int kiemtraKytu(char kytu, char tuBimat[ ], int sokytuBimat[ ])
{
    int i = 0;
    int kytuChinhXac = 0;

    // Kiem tra xem ky tu cua nguoi choi da doan co nam trong tu bi mat ko
    for (i = 0 ; tuBimat[i] != '\0' ; i++)
    {
        if (kytu == tuBimat[i]) // Neu ky tu co chua trong tu bi mat
        {
            kytuChinhXac = 1; // Ky tu se duoc luu tru gia tri the hien no la ky tu chinh xac
            sokytuBimat[i] = 1; // Gui gia tri 1 vao o tuong ung voi vi tri cua ky tu do trong mang
        }
    }

    return kytuChinhXac;
}
```

Hàm sẽ kiểm tra ký tự mà người dùng đã nhập vào có nằm trong ký tự bí mật không. Nếu có, có 2 điều sẽ diễn ra:

- Giá trị Boolean của biến *kytuChinhXac* sẽ bằng 1, và giá trị 1 sẽ được gửi vào vị trí tương ứng của ký tự đó trong *kytuBimat*.
- Chúng ta cập nhật giá trị trong mảng *sokytuBimat* tương ứng với vị trí của ký tự đó trong mảng.

Lợi ích của kỹ thuật này là chúng ta sẽ kiểm tra một lượt toàn bộ các giá trị trong mảng (chứ không ngừng lại ngay sau khi tìm được ký tự đầu tiên). Việc này cho phép chúng ta cập nhật chính xác các giá trị trong mảng *sokytuBimat*, kể cả trong trường hợp có 2 ký tự giống nhau trong một từ như 2 ký tự O trong từ FACEBOOK.

Giải pháp 2: Quản lý tập tin “danh mục từ bí mật”

Chúng ta đã có biết tạo ra hầu hết các chức năng cơ bản của trò chơi, và các bạn đã có thể quản lý chương trình của mình nhưng vẫn còn một phần chưa hoàn thiện, đó là cách chọn ngẫu nhiên một từ bí mật trong “danh mục từ bí mật”. Bạn có thể tưởng tượng danh mục đó nó giống như tôi đã cho bạn xem ở trên, tôi không cho bạn thấy cụ thể toàn bộ nội dung tập tin đó ở đây được vì có thể nó dài tới vài trang giấy chứ không ít đâu.

Trước khi tiếp tục, việc đầu tiên cần làm là tạo ra một “danh mục từ bí mật” cho trò chơi. Ở thời điểm này, dù nó ngắn hay dài không quan trọng, chỉ là để làm thử nghiệm cho bài học này thôi.

Tôi sẽ tạo một tập tin *danhmuc.txt* trong thư mục chứa project của tôi. Và tạm thời nội dung tập tin sẽ trông như sau:

```
LOVE  
MONEY  
PROGRAM  
FUNCTION  
POINTER  
LOOP  
INSTRUCTION  
STRING  
VARIABLE  
CONSTANT  
DEFINE
```

Sau khi hoàn thành chương trình, dĩ nhiên tôi sẽ có thể quay lại danh mục trên và thêm vào tập tin những từ bí mật khác để tăng thêm sự phong phú cho trò chơi.

Những chuẩn bị cho một tập tin mới.

Riêng từ “danh mục” trong tên của tập tin này cũng đủ làm bạn tưởng tượng ra một danh sách có độ dài gần như vô hạn đúng không. Do đó, tôi sẽ thêm một tập tin mới vào project của mình, nó là tập tin *danhmuc.c* (tập tin này có nhiệm vụ đọc *danhmuc.txt*). Và đối với quá trình này, tôi sẽ tạo một tập tin *danhmuc.h*, tập tin này sẽ chứa những prototype của *danhmuc.c*.

Trong *danhmuc.c* tôi sẽ thêm vào các thư viện cần thiết và dĩ nhiên là có cả *danhmuc.h*. Như thường lệ vẫn là những thư viện chuẩn được ưu tiên trước *stdlib*, *stdio* bên cạnh đó, chúng ta cần chọn một số ngẫu nhiên từ danh mục, vì vậy ta sẽ phải thêm vào thư viện *time.h* giống như đã từng làm với chương trình test của chương 1 (trò “lớn hơn hay nhỏ hơn”, bạn còn nhớ ko). Và thêm nữa, ta sẽ sử dụng hàm *strlen* nên bạn phải thêm vào *string.h* nữa nhé:

Xem thử nào:

C code:

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

#include "danhmuc.h"
```

Hàm chọnTu:

Hàm này sẽ cần một tham số. Đó là một con trỏ đến vùng bộ nhớ, nơi mà nó có thể lưu trữ từ ngẫu nhiên trong danh mục vào. Con trỏ này sẽ được cung cấp bởi *main* ().

Hàm sẽ trả về một giá trị kiểu *int* và nó có kiểu Boolean: giá trị 1 nếu mọi việc diễn ra tốt đẹp và 0 nếu có bất kỳ lỗi gì xảy ra.

Sau đây là đoạn đầu của hàm:

C code:

```
int chọnTu(char *tuDuocChon)
{
    FILE* danhmuc = NULL; // Con trỏ tập tin để chứa các tập tin của chúng ta
    int soThuTuCuaTu = 0, soThuTuCuaTuDuocChon = 0, i = 0;
    int lưuKytu = 0;
```

Tôi tạo một số biến cần thiết cho chương trình của chúng ta. Trong *main* (), có thể bạn đã thấy tôi đã không tạo ra tất cả biến cùng một lúc ngay khi mới bắt đầu, có những thứ bạn có thể tạo ra sau nếu bạn nhận thấy bạn cần đến chúng.

Tên của các biến trên cũng cho bạn thấy nhiệm vụ, chức năng của chúng rồi. Chúng ta đã có con trỏ *danhmuc* để đọc tập tin *danhmuc.txt*, những biến tạm thời để lưu trữ ký tự ...

Bạn có để ý tôi khai báo kiểu *int* cho biến *lưuKytu* để lưu trữ ký tự? Hơi lạ đúng không, bởi vì hàm *fgetc* mà tí nữa tôi sẽ dùng nó trả về một giá trị kiểu *int*, vậy nên tốt nhất là nên lưu trữ kết quả của chúng ta kiểu *int*.

Nào xem thử nhé:

C code:

```
danhmuc = fopen("danhmuc.txt", "r"); // Tập tin được mở trong chế độ "read-only"

// Chúng ta kiểm tra xem thao tác mở tập tin có thành công không
if (danhmuc == NULL) // Nếu bạn không thể mở tập tin
{
    printf("\n Không thể mở danh mục từ bí mật");
    return 0; // Trả về giá trị 0 cho biết thao tác mở tập tin thất bại
    // Sau khi nhận được giá trị trả về của return, hàm kết thúc.
}
```

Không có quá nhiều điều mới mẻ ở đây. Tôi chọn chế độ mở tập tin *danhmuc.txt* là “read-only” (bằng cách chọn “r”) và kiểm tra xem thao tác này có thành công hay không bằng cách sử dụng *if* nếu giá trị trả về của *danhmuc* là NULL thì rõ ràng việc mở tập tin đã thất bại (có thể chương trình không tìm thấy tập tin *danhmuc.txt* hoặc nó đang được chương trình khác sử dụng). Trong trường hợp này, màn hình sẽ hiển thị thông báo lỗi và giá trị trả về là 0.

Tại sao *return* lại nằm ở vị trí đó. Thật ra thì, *return* có chức năng dừng các hoạt động của hàm lại. Nếu không thể mở được tập tin, hàm sẽ dừng lại tại đó và máy tính cũng sẽ không tiếp tục đọc thêm gì nữa. Nó trả về giá trị 0 để cho biết rằng hàm này không thực hiện được.

Và đây là phần còn lại của hàm, giả sử việc mở tập tin đã thành công:

C code:

```
// Đếm các từ được chứa trong tập tin (chỉ việc đếm có bao nhiêu ký tự \n thôi)
do
{
    luuKytu = fgetc(danhmuc);
    if (luuKytu == '\n')
        soThuTuCuaTu ++;
} while (luuKytu != EOF);
```

Bạn thấy gì không, chúng ta sẽ đọc qua toàn bộ tập tin nhờ vào hàm *fgetc* (và đọc lần lượt từng ký tự một nhé). Chương trình chỉ việc đếm số lần xuất hiện ký tự **\n**. Mỗi lần ký tự **\n** xuất hiện, giá trị của biến *soThuTuCuaTu* sẽ tăng thêm 1.

Với phần code này, chúng ta sẽ biết được có bao nhiêu từ bí mật nằm trong tập tin. Và nhớ là mỗi một dòng trong tập tin chỉ chứa 1 từ thôi nhé.

C code:

```
soThuTuCuaTuDuocChon = tuNgauNhan(soThuTuCuaTu); // Chọn một từ ngẫu nhiên
```

Như bạn thấy, tôi gọi một hàm theo kiểu riêng của mình, chọn ra từ có số thứ tự ngẫu nhiên từ 1 đến giá trị của *soThuTuCuaTu* (tham số này sẽ được gửi vào hàm).

Đây là một hàm đơn giản mà tôi đã đặt vào file *danhmuc.c* (tôi sẽ cho bạn thấy rõ chi tiết tí nữa). Nó sẽ trả về một giá trị số (tương ứng với số thứ tự của dòng chứa từ đó trong tập tin), giá trị này sẽ được gửi vào biến *soThuTuCuaTuDuocChon*.

C code:

```
// Chương trình đọc lại từ đầu tập tin và ngưng lại khi tìm thấy từ ngẫu nhiên được chọn
rewind(danhmuc);
while (soThuTuCuaTuDuocChon > 0)
{
    luuKytu = fgetc(danhmuc);
    if (luuKytu == '\n')
        soThuTuCuaTuDuocChon --;
}
```

Bây giờ thì chúng ta đã có được số thứ tự của từ được chọn ngẫu nhiên, bằng việc gọi hàm *rewind* () chương trình sẽ bắt đầu đọc tập tin từ đầu. Nó sẽ đếm lần lượt từng ký tự *\n* trong tập tin. Lần này, giá trị của biến *soThuTuCuaTuDuocChon* sẽ giảm xuống. Ví dụ từ được chọn ngẫu nhiên là từ thứ 5 trong tập tin, thì sau mỗi vòng lặp giá trị này sẽ giảm xuống còn 4, 3, 2, 1 và 0. Khi giá trị của *soThuTuCuaTuDuocChon* bằng 0 thì chúng ta sẽ thoát ra khỏi vòng lặp, vì lúc này nó không còn thỏa mãn điều kiện biến này có giá trị lớn hơn 0 nữa.

Bạn cần phải hiểu được ý nghĩa của đoạn code này, nó giúp ta có thể tìm được vị trí hiện tại của “dấu nhảy ảo” trong tập tin. Điều này cũng không thực sự quá khó hiểu nhưng bạn phải cố gắng nhìn nhận mọi việc một cách rõ ràng. Hãy cố hết sức đảm bảo rằng bạn hiểu được những gì tôi đang nói và đang làm.

Bây giờ chúng ta cần một “dấu nháy” để chỉ ra vị trí của từ được chọn trong tập tin.

Chúng ta sẽ gửi đến con trỏ *tuDuocChon* (đây là tham số mà hàm cần) với *fgets* để đọc các từ:

C code:

```
/* Con trỏ tập tin danhmuc được đặt đúng vị trí của nó.
Chúng ta sử dụng hàm fgets và quy định hàm không đọc quá số lượng ký tự cho phép */
fgets(tuDuocChon, 100, danhmuc);

// Chúng ta sẽ thay thế ký tự \n
tuDuocChon[strlen(tuDuocChon) - 1] = '\0';
```

Chúng ta yêu cầu hàm *fgets* không được đọc quá 100 ký tự (cũng tương ứng với kích thước của mảng). Hãy nhớ rằng hàm *fgets* sẽ đọc toàn bộ một dòng bao gồm luôn ký tự *\n*.

Khi bạn không muốn gặp ký tự \n này ở cuối, có thể thay thế bằng \0, ký tự này có tác dụng cắt một chuỗi trước khi gặp ký tự \n.

Và ... nó đã hoạt động ... chúng ta đã lưu trữ ký tự bí mật tại địa chỉ con trỏ *tuDuocChon*.

Chúng ta vẫn chưa đóng tập tin lại, một giá trị trả về 1 để ngừng hàm lại và cho thấy rằng chương trình hoạt động tốt:

C code:

```
fclose(danhmuc);

return 1; // Gia tri tra ve = 1, tat ca deu hoat dong tot
}
```

Và tôi cũng không có gì để nói thêm về hàm *chonTu*.

Hàm tuNgauNhiem:

Đây là hàm mà tôi đã hứa sẽ giải thích cho bạn khi nãy. Nó sẽ chọn một số thứ tự ngẫu nhiên và gửi nó về:

C code:

```
int tuNgauNhiem(int sothutuLonNhat)
{
    srand(time(NULL));
    return (rand() % sothutuLonNhat);
}
```

Dòng đầu tiên `srand(time(NULL));` sẽ giúp chương trình của bạn không chọn trùng các số ngẫu nhiên, giống những gì chúng ta đã từng làm với chương trình “lớn hơn hay nhỏ hơn” trong chương 1.

Dòng thứ 2 sẽ chọn ra một số ngẫu nhiên trong chuỗi giá trị từ 0 đến giá trị của biến *sothutuLonNhat*.

Tập tin danhmuc.h

Sau đây là prototype của các function. Và một điều nữa, bạn có còn nhớ về “người bảo vệ” `#ifndef` mà tôi đã từng yêu cầu các bạn nên thêm vào tất cả những tập tin `.h` của bạn (tham khảo lại những kiến thức của các bài học về prototype và tiền xử lý nhé).

C code:

```
#ifndef DEF_DANHMUC
#define DEF_DANHMUC

int chonTu(char *tuDuocChon);
int tuNgauNhiem(int sothutuLonNhat);

#endif
```

Tập tin danhmuc.c

Và sau đây là toàn bộ nội dung của tập tin *danhmuc.c*, xin mời các bạn thưởng thức:u

C code:

```
/*
Nguoi Treo Co

danhmuc.c
-----

Nhưng function này sẽ chọn ra một số ngẫu nhiên trong tập tin chứa danh mục số bí ẩn của trò
chơi Nguoi Treo Co
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#include "danhmuc.h"

int chonTu(char *tuDuocChon)
{
    FILE* danhmuc = NULL; // Con trỏ tập tin để chứa các tập tin của chúng ta
    int soThuTuCuaTu = 0, soThuTuCuaTuDuocChon = 0, i = 0;
    int luuKytu = 0;
    danhmuc = fopen("danhmuc.txt", "r"); // Tập tin được mở trong chế độ "read-only"

    // Chúng ta kiểm tra xem thao tác mở tập tin có thành công không
    if (danhmuc == NULL) // Nếu bạn không thể mở tập tin
    {
        printf("\n Không thể mở danh mục số bí ẩn");
        return 0; // Trả về giá trị 0 cho biết thao tác mở tập tin thất bại
        // Sau khi nhận được giá trị trả về của return, hàm kết thúc.
    }
}
```

```
// Dem cac tu duoc chua trong tap tin (chi viec dem co bao nhieu ky tu \n thoi)
do
{
    luuKytu = fgetc(danhmuc);
    if (luuKytu == '\n')
        soThuTuCuaTu++;
} while(luuKytu != EOF);

soThuTuCuaTuDuocChon = tuNgauNhien(soThuTuCuaTu); // Chon mot tu ngau nhien

// Chuong trinh doc lai tu dau tap tin va ngung lai khi tim thay tu ngau nhien duoc chon
rewind(danhmuc);
while (soThuTuCuaTuDuocChon > 0)
{
    luuKytu = fgetc(danhmuc);
    if (luuKytu == '\n')
        soThuTuCuaTuDuocChon--;
}

/* Con tro tap tin danhmuc duoc dat dung vi tri cua no.
Chung ta su dung ham fgets va quy dinh ham khong doc qua so luong ky tu cho phép*/
fgets(tuDuocChon, 100, danhmuc);

// Chung ta se thay the ky tu \n
tuDuocChon[strlen(tuDuocChon) - 1] = '\0';
fclose(danhmuc);

return 1; // Gia tri tra ve = 1, tat ca deu hoat dong tot
}

int tuNgauNhien(int sothutuLonNhat)
{
    srand(time(NULL));
    return (rand() % sothutuLonNhat);
}
```

Chúng ta sẽ làm vài thứ trong main.c:

Bây giờ tập tin *danhmuc.c* đã sẵn sàng, chúng ta cùng trở lại với *main.c* để thêm vào một số thứ cho phù hợp.

Với những gì đã làm từ đầu đến giờ, trước hết chúng ta sẽ thêm vào file *danhmuc.h* nếu bạn muốn sử dụng các function của *danhmuc.c*. Ngoài ra, chúng ta cũng sẽ không quên khai báo thêm thư viện *string.h* bởi vì chúng ta sẽ phải sử dụng hàm *strlen*:

C code:

```
#include <string.h>
#include "dico.h"
```

Để bắt đầu, việc khai báo các biến sẽ có một chút thay đổi. Bạn không cần khởi tạo một chuỗi ký tự *tuBimat* bởi vì chúng ta đã có sẵn một mảng kiểu char cho nó (chứa sẵn 100 ô luôn).

Với mảng *sokytuBimat*, kích thước của nó phụ thuộc vào độ dài của từ bí mật. Bởi vì chúng ta chưa biết được kích thước đó, nên sẽ có một con trỏ được tạo ra, cùng với *malloc* chúng ta có thể gửi con trỏ đến vị trí bộ nhớ mà nó sẽ được cấp phát.

Đây là một vd tuyệt vời cho bài học về cấp phát động của chúng ta trước đây: chúng ta không thể nào biết trước được kích thước của mảng nếu chương trình chưa chạy, vì vậy bắt buộc bạn phải tạo ra một con trỏ và sử dụng hàm *malloc*.

Bạn không được quên giải phóng bộ nhớ để còn sử dụng cho những mục đích khác nhé. Đó cũng chính là lý do cho sự xuất hiện của *free ()* ở cuối *main.c*.

Chúng ta sẽ cần một biến *dodaiTu* để lưu trữ giá trị thể hiện số lượng ký tự của từ. Như bạn đã thấy trong phần đầu, chúng ta đã giả định từ bí mật sẽ có 8 ký tự (bởi vì chúng ta chọn FACEBOOK làm vd mà). Nhưng bây giờ chúng ta đã có thể làm việc linh động với mọi từ với kích thước tùy biến.

Và sau đây là đầy đủ tất cả những biến cần cho chương trình của chúng ta:

C code:

```
int main(int argc, char* argv[ ])
{
    char kytu = 0; // Biến này sẽ lưu trữ ký tự của người chơi (được tra về bởi hàm scanf)
    char tuBimat[100] = {0}; // Đây là từ bí mật cần tìm ra
    int *sokytuBimat = NULL; /* Một mảng có chứa các thành phần dạng Boolean. Mỗi ô trong
mảng sẽ tương ứng với một ký tự người chơi sẽ đoán. Nếu đoán đúng thì giá trị = 1, và nếu sai thì
giá trị = 0 */
    int soluoatDoan = 10; // Số lượt đoán còn lại của người chơi (0 = thua)
    int i = 0; // Biến hỗ trợ
    int dodaiTu = 0;
```

Trên đây chủ yếu là những thay đổi ban đầu, bây giờ chúng ta sẽ đi sâu hơn một chút:

C code:

```
if (!chonTu(tuBimat))  
    exit(0);
```

Đầu tiên chúng ta đặt hàm *chonTu* vào phần điều kiện của if, hàm này sẽ lấy tham số là biến *tuBimat*.

Và như ta đã biết, hàm sẽ trả về một giá trị Boolean (1 hoặc 0) để cho biết nó có được thực hiện thành công hay không. Vai trò của *if* là phân tích các giá trị Boolean được trả về bởi hàm *chonTu*, nếu hàm KHÔNG hoạt động (hãy lưu ý trong if có chứa dấu ! để thể hiện phủ định), thì ngừng lại tất cả bằng *exit (0)*.

C code:

```
dodaiTu = strlen(tuBimat);
```

Giá trị thể hiện kích thước của *tuBimat* được lưu trữ trong biến *dodaiTu*.

C code:

```
sokytuBimat = malloc(dodaiTu * sizeof(int)); /* mang sokytuBimat se duoc cap phat dong bo  
nho (luc dau chung ta khong biet duoc kích thước của mảng này) */  
if (sokytuBimat == NULL)  
    exit(0);
```

Bây giờ chúng ta phải cấp phát bộ nhớ cho mảng *sokytuBimat*. Kích thước của mảng này đã được cung cấp bởi giá trị của biến *dodaiTu*.

Sau đó dùng if để kiểm tra giá trị của con trỏ có bằng NULL không. Trong trường hợp nếu bằng NULL, chúng ta việc cấp phát bộ nhớ đã thất bại, chúng ta sẽ dừng chương trình ngay lập tức (bằng cách gọi hàm *exit (0)*)

Đây là tất cả những gì bạn cần chuẩn bị cho chương trình. Bây giờ chúng ta cần thay đổi nốt phần còn lại của *main.c* để thay thế tất cả các con số 8 (số thể hiện độ dài của từ FACEBOOK mà ta giả định là từ bí mật lúc đầu bài học) bằng biến *dodaiTu*. VD:

C code:

```
for (i = 0 ; i < dodaiTu ; i++)  
    sokytuBimat[i] = 0;
```

Ban đầu, đoạn code trên đặt giá trị 0 vào các ô trong mảng *sokytuBimat* và có thể hiểu là không có ô nào trong mảng, dần dần giá trị đó sẽ tăng lên cho đến khi nào bằng với giá trị của biến *dodaiTu* thì ngừng lại, và đó cũng chính là kích thước chính xác cho mảng.

Tôi cũng phải thiết kế lại prototype của function *win* để thêm vào biến *dodaiTu*. Nếu không các hàm sẽ không biết khi nào phải dừng vòng lặp.

Sau đây là tập tin *main.c* hoàn thiện:

C code:

```
/*
Nguoi Treo Co

main.c
-----

Nhưng function này sẽ chọn ra một từ ngẫu nhiên trong tập tin chứa danh mục từ bí ẩn của trò
chơi Nguoi Treo Co
*/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#include "danhmuc.h"

int win(int sokyTuBimat[ ], long dodaiTu);
int kiểmTraKytu(char kytu, char tuBimat[ ], int sokyTuBimat[ ]);
char docKytu( );

int main(int argc, char* argv[ ])
{
    char kytu = 0; // Biến này sẽ lưu trữ ký tự của người chơi (được trả về bởi hàm scanf)
    char tuBimat[100] = {0}; // Đây là từ bí mật cần tìm ra
    int *sokyTuBimat = NULL; /* Một mảng có chứa các thành phần dạng Boolean. Mỗi ô trong
mảng sẽ tương ứng với một ký tự người chơi sẽ đoán. Nếu đoán đúng thì giá trị = 1, và nếu sai thì
giá trị = 0 */
    long soluotDoan = 10; // Số lượt đoán còn lại của người chơi (0 = thua)
    long i = 0; // Biến hỗ trợ
    long dodaiTu = 0;

    printf("Chào mừng đến với trò chơi Nguoi treo co !\n\n");

    if (!chọnTu(tuBimat))
        exit(0);
}
```

```
dodaiTu = strlen(tuBimat);

sokytuBimat = malloc(dodaiTu * sizeof(int)); /* mang sokytuBimat se duoc cap phat dong bo
nho (luc dau chung ta khong biet duoc kích thước của mảng này) */
if (sokytuBimat == NULL)
    exit(0);

for (i = 0 ; i < dodaiTu ; i++)
    sokytuBimat[i] = 0;

/* Chúng ta sẽ tiếp tục trò chơi nếu còn ít nhất một lượt đoán
   Và vẫn chưa tìm được từ bí mật*/
while (soluotDoan > 0 && !win(sokytuBimat, dodaiTu))
{
    printf ("\n\n Bạn có %d lượt đoán để chơi ", soluotDoan);
    printf ("\n Tu bí mật là gì ?");

    /* Hiện thị những ký tự bí mật và ẩn đi những ký tự chưa được tìm thấy
       Ví dụ: *A***OO*/
    for (i = 0 ; i < dodaiTu ; i++)
    {
        if (sokytuBimat[i] // Nếu người chơi tìm được ký tự thứ i
            printf ("%c", tuBimat[i]); // Hiện thị ký tự thứ i được tìm thấy
        else
            printf ("*");// Hiện thị dấu * đối với những ký tự chưa được tìm thấy
    }

    printf ("\n Xin mời bạn đoán một ký tự: ");
    kytu = docKytu( );

    // Nếu ký tự nhập vào không đúng
    if (!kiemtraKytu(kytu, tuBimat, sokytuBimat))
    {
        soluotDoan--; // Giảm bớt một lần đoán của người chơi
    }
}

if (win(sokytuBimat, dodaiTu))
    printf ("\n\n Chúc mừng, bạn đã chiến thắng ! Từ bí mật là : %s", tuBimat);
```

```
else
    printf ("\n\n Xin chia buồn, bạn đã thua !\n\n Tu bí mật là : %s", tuBimat);

free (sokytuBimat); // Giải phóng bộ nhớ đã được phân bổ (bởi hàm malloc)

return 0;
}

char docKytu( )
{
    char kytuNhapVao = 0;

    kytuNhapVao = getchar( ); // Đọc ký tự được nhập đầu tiên
    kytuNhapVao = toupper(kytuNhapVao); // Viết hoa ký tự đó

    // Lan lượt đọc tiếp các ký tự khác cho đến khi gặp \n
    while (getchar( ) != '\n') ;

    return kytuNhapVao; // Trả về ký tự đầu tiên đọc được
}

int win(int sokytuBimat[ ], long dodaiTu)
{
    long i = 0;
    int nguoiChoiChienThang = 1;

    for (i = 0 ; i < dodaiTu ; i++)
    {
        if (sokytuBimat[i] == 0)
            nguoiChoiChienThang = 0;
    }

    return nguoiChoiChienThang;
}

int kiểmtraKytu(char kytu, char tuBimat[ ], int sokytuBimat[ ])
{
    long i = 0;
    int kytuChinhXac = 0;
```

```
// Kiểm tra xem ký tự của người chơi đã đoán có nằm trong từ bí mật không
for (i = 0 ; tuBimat[i] != '\0' ; i++)
{
    if (kytu == tuBimat[i]) // Nếu ký tự có chứa trong từ bí mật
    {
        kytuChinhXac = 1; // Ký tự sẽ được lưu trữ giá trị thể hiện nó là ký tự chính xác
        sokytuBimat[i] = 1; // Gán giá trị 1 vào ô tương ứng với vị trí của ký tự đó trong mảng
    }
}

return kytuChinhXac;
}
```

Ý tưởng cải tiến:

Chà, trò “Người treo cổ” này có hơi phức tạp với bạn không. Bây giờ bạn đã có một chương trình chọn ra từ ngẫu nhiên từ một tập tin rồi đúng không.

Sau đây là một số ý tưởng cải tiến để bạn thử sức:

- Hiện nay, chúng ta chỉ mới cho phép mọi người chơi một lần. Nghĩa là chương trình sẽ ngừng lại khi có người chiến thắng hoặc sử dụng hết lượt đoán. Bây giờ, bạn hãy thử tạo một yêu cầu người chơi xem họ có muốn chơi lại không.
- Bạn cũng có thể tạo ra chế độ chơi 2 người, người thứ nhất sẽ nhập từ bí ẩn vào cho người thứ 2 đoán.
- Mặc dù không bắt buộc nhưng sao bạn không thử vẽ hình người bị treo cổ trên màn hình console (như các kim từ điển hay có) gợi ý là bạn có thể làm bằng hàm *printf*.

Hãy cố gắng bỏ thời gian ra để hiểu bài học này và cải tiến nó hết mức có thể nhé.

Cố lên nào, đừng nản lòng.