Các câu hỏi phỏng vấn Embedded MCU thường gặp - 2023

### **Một số lưu ý trước phỏng vấn:**

* Thời gian phỏng vấn thường khá lâu, có thể 1.5 giờ, có thể 1 vòng hoặc chia làm 2 vòng (vòng 1 về kĩ thuật, vòng 2 có thể là PM (Project Manager) / khách hàng của công ty Outsource phỏng vấn).
* Nên chuẩn bị tinh thần hứng khởi, cầu thị, trước khi phỏng vấn. Sắc mặt sáng sủa, tác phong gọn gàng nghiêm túc.
* Chuẩn bị kiến thức tốt để tự tin phỏng vấn.
* Buổi phỏng vấn thường trao đổi bằng tiếng anh.
* Câu hỏi nào nghe không rõ thì xin họ nhắc lại.
* Câu nào không biết thì nói họ cái này tôi không biết.
* Nếu hay run thì nên mang theo chai nước uống, nước trà hay nước lọc gì cũng được, thấy căng căng là làm ngụm liền.

### **1. Giới thiệu về bản thân:**

Đối với Fresher:

* Tên, trường đại học, chuyên nghành.
* Trình bày quá trình học tập, kinh nghiệm đồ án, kinh nghiệm thực tập,...

**Lưu ý:**

* Nếu học điện, điện tử / cơ điện tử / điện tử viễn thông thì nên giới thiệu là thích lập trình và tự học lập trình.
* Tham gia các cuộc thi robot, cuộc thi lập trình là điểm +
* Thể hiện được niềm đam mê của mình là điểm +
* Có kinh nghiệm đi thực tập là điểm +
* Thực hành giải thuật toán trên các trang luyện thuật toán ví dụ: [leetcode](https://leetcode.com/" \t "_blank),... là điểm +
* Nếu có một đồ án / sản phẩm gì tâm đắc và giới thiệu say sưa được là điểm +
* Tiếng Anh ngon là điểm +

 Đối với Developer có kinh nghiệm làm việc:

* Hỏi về sao em nghỉ công ty cũ, đồng nghiệp thế nào ?
* Vì sao chọn công ty này ? Từng làm việc với người nước ngoài chưa…

Phần này thì mình không đề cập nhiều, cơ bản đã có kinh nghiệm làm việc rồi thì khắc sẽ biết :) . Đừng nói xấu công ty cũ tưng bừng là được :).

### **2. Đối với phần kiểm tra kiến thức và kinh nghiệm.**

#### **Lập trình C**

Struct là gì ?

Tuy nhiên, trong phạm vi của lập trình cơ bản, struct sẽ được sử dụng chủ yếu trong việc tạo ra các kiểu dữ liệu tự định nghĩa, và mức độ sử dụng cũng rất đơn giản. Ta có thể hiểu struct là một dạng ***bản ghi***, dùng để lưu trữ những thông tin chung nhất của nhiều đối tượng dữ liệu, ví dụ như mọi bản ghi của các sinh viên đại học trong hệ thống của nhà trường đều phải có tên, mã số sinh viên, ngày tháng năm sinh và ngành học; hoặc như ví dụ về lưu trữ các cuốn sách ở trên,...

Kích thước của Struct tính như thế nào ?

# **Struct Alignment trong C++**C/C++ cung cấp cho chúng ta một số kiểu dữ liệu cơ bản như char, int, float, double, long... C/C++ cũng cho phép chúng ta tạo ra những kiểu dữ liệu mới tổng hợp nhiều kiểu dữ liệu bằng cách dùng struct, class. Việc cấp phát và tổ chức bộ nhớ đối với dữ liệu cơ bản là khá đơn giản, còn đối với struct, class thì cấp phát, tổ chức bộ nhớ như thế nào.

Thông tin chi tiết về các kiểu dữ liệu này được thể hiện ở bảng dưới:

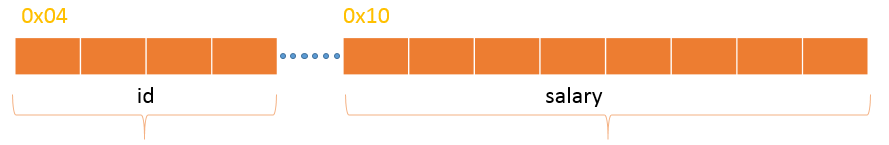
|  |  |  |
| --- | --- | --- |
| KIỂU DỮ LIỆU | KÍCH THƯỚC | MIỀN GIÁ TRỊ |
| char | 1 byte | -127 to 127 or 0 to 255 |
| unsigned char | 1 byte | 0 to 255 |
| signed char | 1 byte | -127 to 127 |
| int | 4 bytes | -2147483648 to 2147483647 |
| unsigned int | 4 bytes | 0 to 4294967295 |
| signed int | 4 bytes | -2147483648 to 2147483647 |
| short int | 2 bytes | -32768 to 32767 |
| unsigned short int | 2 bytes | 0 to 65,535 |
| signed short int | 2 bytes | -32768 to 32767 |
| long int | 4 bytes | -2,147,483,647 to 2,147,483,647 |
| signed long int | 4 bytes | -2,147,483,647 to 2,147,483,647 |
| unsigned long int | 4 bytes | 0 to 4,294,967,295 |
| float | 4 bytes | +/- 3.4e +/- 38 (~7 digits) |
| double | 8 bytes | +/- 1.7e +/- 308 (~15 digits) |
| long double | 8 bytes | +/- 1.7e +/- 308 (~15 digits) |

Trước khi đi vào chủ đề của bài viết bạn cần nắm rõ những biến này được tổ chức như thế nào trên bộ nhớ. Giả sử tôi có những câu lệnh dưới đây:

int id = 13520546; // 4 bytes

double salary = 500.000; // 8 bytes

Và tổ chức bộ nhớ sẽ là:



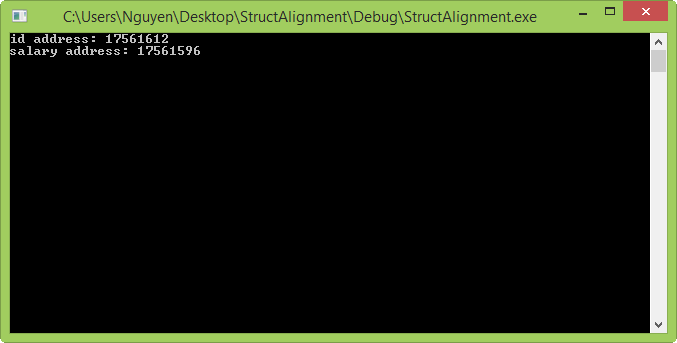
Vùng nhớ của id và salary có thể là không liên tục trên bộ nhớ.

Kết quả khi chạy chương trình dưới đây và in ra địa chỉ của hai biến id và salary:

int id = 13520546;

double salary = 500.000;

printf("id address: %d\nsalary address: %d\n", &id, &salary);



## Tổ chức bộ nhớ trong struct, class

Tôi tạo một struct Point thể hiện tọa độ của điểm trong không gian hai chiều:

struct Point

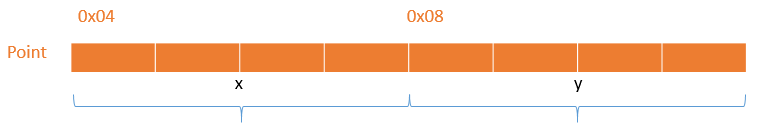
{

int x;

int y;

};

Lấy kích thước của struct bằng cách sử dụng toán tử sizeof được cung cấp trong C++, kích thước của nó là 8 bytes đúng bằng các kích thước các trường dữ liệu thành viên cộng lại.



Tạo tiếp một struct Student thể hiện thông tin của một sinh viên gồm id, age, gpa là điểm trung bình học kì:

struct Student

{

char id;

int age;

double gpa;

};

Lấy kích thước của của struct Student:

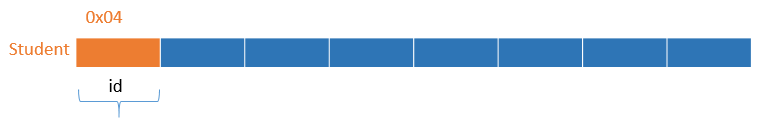
int size = sizeof(Student);

Kích thước mong đợi là 13 bytes bằng kích thước của id (1 bytes) cộng với age (4 bytes) và gpa (8 bytes). Nhưng thực tế kích thước có thể là **16 bytes**.

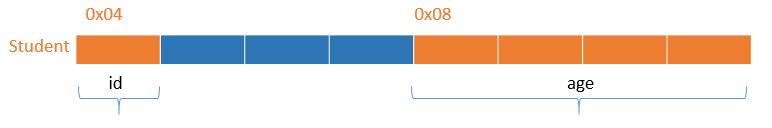
Đầu tiên trình biên dịch sẽ lấy kích thước của trường dữ liệu thành viên có kích thước lớn nhất (theo đơn vị byte) mà kích thước đó là lũy thừa của 2.

Trong trường hợp này là 8 bytes theo kích thước của gpa.

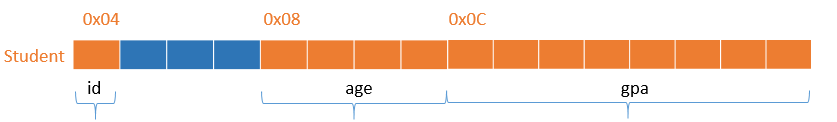
Sau đó cấp phát 1 block gồm đúng bằng kích thước này là 8 bytes, id với kích thức 1 bytes sẽ ở vị trí như hình dưới:



Ta vẫn còn 7 chỗ trống, ở đây tùy theo trình biên dịch mà nó sẽ padding bao nhiêu bytes rồi "đẩy" age kiểu int có kích thước 4 bytes vào:



Đã hết chỗ trống, tiếp tục cấp thêm block 8 bytes nữa và "đẩy" gpa kiểu double có kích thước 8 bytes vào:



Vậy kích thước của struct này là 16 bytes.

Vùng nhớ cấp phát cho struct luôn luôn là liên tục trong bộ nhớ như ở trên ta thấy là địa chỉ của id, age, gpa lần lượt là 0x04, 0x08, 0xC.

Khảo sát thêm một struct nữa có tên là Demo

struct Demo

{

int a, b;

double c, d;

};

Kích thước struct này là: 24 bytes;

Sắp xếp lại các trường dữ liệu như sau:

struct Demo

{

int a;

double c, d;

  int b;

};

Kiểm tra thấy kích thức của struct này là 32 bytes

Như vậy struct có cùng các trường dữ liệu như nhau, nhưng nếu thay đổi vị trí của chúng trong struct thì có thể thay đổi kích thước của struct đó. Dựa vào cách sắp xếp bộ nhớ trong struct mà tôi đã trình bày ở trên giúp các bạn có thể tránh được việc mất mát vùng nhớ.

Lưu ý: Cách tổ chức bộ nhớ trong class cũng tượng tự như struct.

## Struct, class không có dữ liệu thành viên

Chúng ta đã khảo sát struct có các trường dữ liệu, nếu struct đó không có dữ liệu thành viên thì sao, việc lưu trữ như thế nào? Một ví dụ điển hình là class xử lý toán học Math trong C#.

struct Person

{

};

1 struct Person không có thành viên và lấy kích thước của nó như bên dưới:

int size = sizeof(Person);

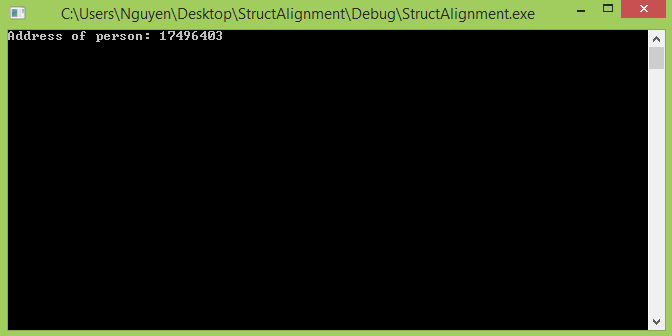
Kích thước sẽ là 1 byte. Đối với những struct hay class như thế này thì trình biên dịch sẽ cấp phát 1 byte để có thể lưu trữ nó dưới bộ nhớ.

Tạo 1 biến có tên là person và lấy địa chỉ của nó trong bộ nhớ:

Person person;

printf("Address of person %d\n", &person);

Kết quả in lên màn hình



## Địa chỉ của struct, class

Cũng giống như mảng, địa chỉ của mảng chính là địa chỉ của phần tử đầu tiên, đối với class, struct cũng vậy. Địa chỉ của nó chính là địa chỉ của thành viên đầu tiên trong struct, class đó.

struct Student

{

    char id;

    int age;

    double gpa;

};

int main()

{

Student student;

printf("Address of student: %d\n", &student);

printf("Address of student.id: %d\n", &student.id);

printf("Address of student.age: %d\n", &student.age);

printf("Address of student.gpa: %d\n", &student.gpa);

  return 0;

}

Và kết quả cho thấy rằng địa chỉ của student cũng chính là địa chỉ của  id trong struct đó:



Để tìm kích cỡ của Struct chúng ta sử dụng toán tử **sizeof()** trong C. **sizeof** là một toán tử chứ KHÔNG phải một hàm trong C. Toán tử **sizeof** này nhận một tham số là bất kỳ biến nào và trả về kích cỡ của biến đó.

Giả sử một Struct sinh viên gồm các thuộc tính: name, id, marks

**Kích cỡ của Struct = sizeof(name) + sizeof(id) + sizeof(marks)**

Dưới đây là chương trình C để giải bài tính kích cỡ của Struct trong C

struct ST{

   int Index;

   char Team;

}

// phần tử lớn nhất là Index có kích thước 4 byte. -> Kích thước struct là bội 4

// Index nằm trong block 4 byte đầu tiên. Block này không còn byte trống, do con chip thường là 32 bit thì 4 byte là đủ rồi mà.

// Team có kích thước 1 byte phải nằm ở block tiếp theo. Block này là 4 byte thừa 3 byte

// => kích thức struct này là 8 byte.

Struct khác gì với Union ?

**Sự khác nhau giữa struct và union**

Việc định nghĩa, khai báo biến, truy cập đến các thành phần của struct và union là giống nhau. Tuy nhiên, giữa struct và union có một vài điểm khác nhau:

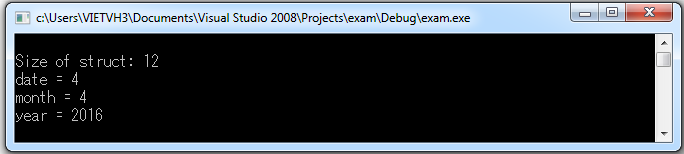
|  |  |
| --- | --- |
| **struct** | **union** |
| Size của struct ít nhất bằng tổng size của các thành phần của struct. Mình sử dụng từ “ít nhất” là vì size struct còn phụ thuộc vào alignment struct.  sizeof(A) = 4 + 10 + 4 = 18 (trong trường hợp struct alignment member = 1 byte)   |  |  | | --- | --- | | 1  2  3  4  5  6 | **struct** A  {  **int** x;  **char** s[10];  **float** f;  }; | | Size của union bằng size của thành phần có size lớn nhất trong union.  sizeof(A) = 10 (kích thước của thành phần lớn nhất trong union)   |  |  | | --- | --- | | 1  2  3  4  5  6 | **union** A  {  **int** x;  **char** s[10];  **float** f;  }; | |
| Tại cùng 1 thời điểm run-time, có thể truy cập vào tất cả các thành phần của struct | Tại cùng 1 thời điểm run-time, chỉ có thể truy cập 1 thành phần của union |

**Ví dụ struct và union**

Ví dụ 1: Biểu diễn thời gian ngày tháng năm, tính size của struct và hiển thị từng thành phần của struct

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | #include <stdio.h>  #include <conio.h>    **struct** date  {  **int** d;  **int** m;  **long** y;  };    **void** main()  {      date dat = {4, 4, 2016};  **printf**("\nSize of struct: %d", **sizeof**(date));    **printf**("\ndate = %d", dat.d);  **printf**("\nmonth = %d", dat.m);  **printf**("\nyear = %d", dat.y);        getch();    } |

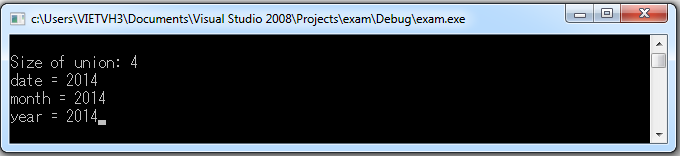
Kết quả:

struct

Ví dụ 2: Biểu diễn thời gian ngày tháng năm bằng union, tính size của union.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | #include <stdio.h>  #include <conio.h>    **union** date  {  **int** d;  **int** m;  **int** y;  };    **void** main()  {      date dat;    **printf**("\nSize of union: %d", **sizeof**(date));      dat.d = 24;      dat.m = 9;      dat.y = 2014;    **printf**("\ndate = %d", dat.d);  **printf**("\nmonth = %d", dat.m);  **printf**("\nyear = %d", dat.y);        getch();    } |

Kết quả:

union

* Kích thước của union = kích thước lớn nhất của thành phần của union = sizeof(int) = 4
* Vùng nhớ giành cho union date là 4 byte. Vùng nhớ này sẽ chứa giá trị 24 khi dat.d = 24 được thực hiện. Tiếp đó, 9 sẽ được copy đè vào vùng nhớ này khi dat.m = 9 được thực hiện. Cuối cùng, 2014 được copy đè vào vùng nhớ khi dat.y = 2014 được thực hiện.

Ví dụ 3:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | #include <stdio.h>  #include <conio.h>    **union** date  {  **int** d;  **int** m;  **int** y;  };    **void** main()  {      date dat;    **printf**("\nSize of union: %d", **sizeof**(date));        dat.d = 24;  **printf**("\ndate = %d", dat.d);        dat.m = 9;  **printf**("\nmonth = %d", dat.m);        dat.y = 2014;  **printf**("\nyear = %d", dat.y);        getch();  } |

Kết quả:

union

Ta thấy, tại một thời điểm trong chương trình chỉ có 1 thành phần của union được sử dụng

## So sánh struct và union

Về mặt ý nghĩa, struct và union cơ bản giống nhau. Tuy nhiên, về mặt lưu trữ trong bộ nhớ, chúng có sự khác biệt rõ rệt như sau:

* struct: Dữ liệu của các thành viên của struct được lưu trữ ở những vùng nhớ khác nhau. Do đó kích thước của 1 struct chắc chắn lớn hơn kích thước của các thành viên cộng lại.

### **Tại sao lại lớn hơn kích thước của thành viên?**

#### **Lý do**

Tùy thuộc và kích thước của khối nhớ do hệ thống quy định.

#### **Ví dụ**

Hệ thống quy định khối nhớ là 1 byte, thì kích thước của kiểu dữ liệu sSinhVien là 13 bytes.

struct sSinhVien

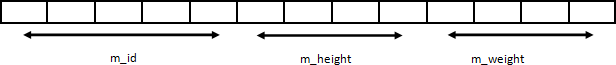
{

char m\_id[5];

float m\_height;

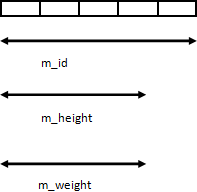
float m\_weight;

};



#### **Thông tin thêm**

* struct: thực tế struct trên nếu lấy kích thước thông thường sẽ có "hiệu ứng" Alignment nên kích thước sẽ không phải là 13 bytes, có thể đọc thêm bài [Struct Alignment Trong C++](https://www.iostream.vn/modern-cpp/struct-alignment-trong-c-f0QnH).
* union: với union, các thành viên sẽ dùng chung 1 vùng nhớ. Kích thước của union được tính là kích thước lớn nhất của kiểu dữ liệu trong union. Việc thay đổi nội dung của 1 thành viên sẽ dẫn đến thay đổi nội dung của các thành viên khác.



Padding trong Struct là thế nào ?

Chúng ta đều biết rằng các CPU hiện đại của chúng ta luôn thao tác trên memory theo từng khối ở địa chỉ là một số chẵn, không thể thao tác trên địa chỉ là số lẻ được.

Xét 1 struct như này:

**typedef** **struct**

{

**char** c;

**int** i;

} mstruct;

Như vậy với các struct của chúng ta, ví dụ nếu một biến “char c” nằm trên memory có địa chỉ chẵn, thì biến “int i” nằm kế tiếp thì sẽ có địa chỉ lẻ rất phức tạp để CPU thao tác với biến “int i” này. Vì vậy chúng ta có thêm 2 khái niệm sau:

**Data alignment**: sắp xếp data sao cho địa chỉ của các biến luôn là số chẵn và phù hợp với hệ thống.

**Data padding**: để làm được việc alignment như ở trên chúng ta cần phải “padding” (đệm) thêm một số byte vào sau biến “char c” để khi đó biến “int i” có thể ở địa chỉ chẵn.

Chúng ta xét một số ví dụ sau để hiểu rõ hơn về cách tính kích thước struct và các vấn đề về phân mảnh bộ nhớ.

**Ví dụ 1:**

Ta khai báo 1 struct A và giả xử thanh ghi của RAM có kích thước là 8 bytes (hệ thống 64 bit).

struct A {

   // sizeof(int) = 4

   int x;

   // Đệm mất 4 bytes

   // sizeof(double) = 8

   double z;

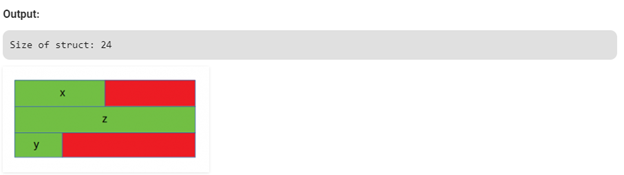
   // Đệm mất 0 bytes

   // sizeof(short int) = 2

   short int y;

   // Đệm mất 6 bytes

};



Như vậy đối với struct A kích thước của nó sẽ là 24 bytes, trong đó có 14 bytes được sử dụng và 10 bytes bị padding.

**Ví dụ 2:**

// C program to illustrate

// size of struct

#include <stdio.h>

**int** main**()**

{

**struct** B {

       // sizeof(double) = 8

**double** z;

       // sizeof(int) = 4

**int** x;

       // sizeof(short int) = 2

**short** **int** y;

       // Đệm mất 2 bytes

   };

   printf("Size of struct: %ld", **sizeof**(struct B));

**return** 0;

}



Như vậy đối với struct B kích thước của nó sẽ là 16 bytes, trong đó có 14 bytes được sử dụng và 2 bytes bị padding. Chúng ta thấy rằng việc sắp xếp thứ tự các phần tử của struct có thể giúp cho việc xử dụng tài nguyên RAM trở lên hiệu quả hơn, tránh bị tốn quá nhiều bytes cho quá trình padding.

**Ví dụ 3:**

// C program to illustrate

// size of struct

#include <stdio.h>

**int** main**()**

{

**struct** C {

       // sizeof(double) = 8

**double** z;

       // sizeof(short int) = 2

**short** **int** y;

       // Đệm mất 2 bytes

       // sizeof(int) = 4

**int** x;

   };

   printf("Size of struct: %ld", **sizeof**(struct C));

**return** 0;

}



Union là gì ?

# **UNION – STRUCT**

Union trong C là một kiểu dữ liệu đặc biệt có sẵn trong C cho phép lưu trữ các kiểu dữ liệu khác nhau trong cùng một vị trí bộ nhớ. Cấu trúc của Union là tất cả các thành phần của nó dùng chung một vùng nhớ có kích thước tương ứng với thành phần lớn nhất.

Cấu trúc của **Struct** là cung cấp một loạt vùng nhớ liền nhau cho từng thành phần.

Vậy nếu ta đặt một structure trong một union thì sao?

## Ví dụ:

**union** {

float Data4bytes;

**struct** {

char bytel;

char byte2;

char byte3;

char byte4;

} Byte;

} Data;

Data.Data4bytes = 26.04;

Đoạn lệnh trên tạo ra một union tên Data chiếm vùng nhớ 4 bytes (cả biến int và struct 4 char điều có chiều dài 4 bytes. Vùng nhớ được cấp cho các biến như sau.



Chương trình đầy đủ

#**include** <stdio.h>

union {

  float Data4bytes;

**struct** {

     char bytel;

     char byte2;

     char byte3;

     char byte4;

  }Byte;

} Data;

int main()

{

  Data.Data4bytes = 26.04;

  printf("4 Bytes:\nByte1:%d\nByte2:%d\nByte3:%d\nByte4:%d\n",Data.Byte.bytel,Data.Byte.byte2,Data.Byte.byte3,Data.Byte.byte4);

  return 0;

}

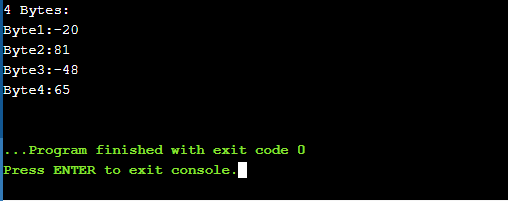
Như vậy ta đã có thể tách khối dữ liệu 4 bytes riêng từng byte 1 để xử lý. Đồng thời ta cũng có thể từ các dữ liệu nhỏ gộp thành một khối dữ liệu lớn.

# **Ứng dụng trong lập trình nhúng**

Như chúng ta đã biết các giao thức trong lập trình nhúng cũng như IoT đều sử sụng các mảng bytes để truyền dữ liệu Ví dụ như UART, I2C,... hay là HTTP, FTP,.. Tất cả đều cần dùng đến xử lý chuỗi và xử lý dữ liệu. Ví dụ chúng ta cần truyền một biến float qua UART nhưng UART chỉ nhận kiểu char hay bytes.

Và chúng ta có thể dựa vào tính chất này để quy đổi từ kiểu dữ liệu bất kì ra chuỗi bytes đơn thuần để thuận tiện cho xử lý dữ liệu cho lập trình nhúng.

Kết quả của chương trình:

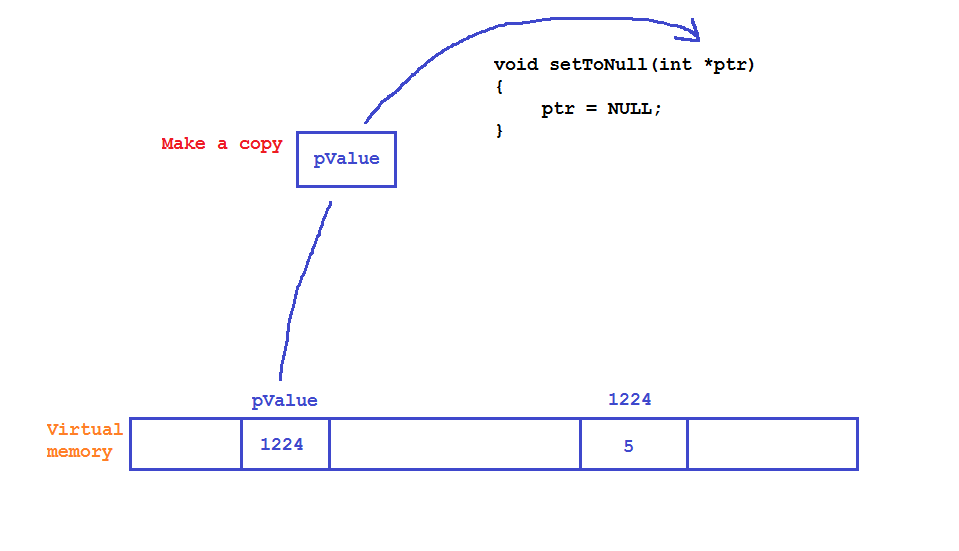


31 tháng trước · 7936 views

Con trỏ là gì ?

Con trỏ trong C cũng chỉ là biến, cũng có thể khai báo, khởi tạo và lưu trữ giá trị và có địa chỉ của riêng nó. Nhưng biến con trỏ không lưu giá trị bình thường, nó là biến trỏ tới 1 địa chỉ khác, tức mang giá trị là 1 địa chỉ trong RAM.

Nếu phỏng vấn có code thì người phỏng vấn sẽ đưa các hàm con có sử dụng con trỏ để thử thách ứng viên như sau:

* Con trỏ trong parameter của hàm.
* Tham số của hàm là tham chiếu vào con trỏ
* Khi chúng ta truyền đối số cho hàm là một địa chỉ, cái địa chỉ này cũng chỉ là bản copy của địa chỉ ban đầu. Về bản chất, truyền địa chỉ vào hàm là truyền đối số là giá trị (**pass by value**). Địa chỉ của đối số sẽ được copy và gán lại cho tham số con trỏ của hàm. Nếu bên trong hàm có câu lệnh thay đổi địa chỉ được truyền vào, chúng chỉ thay đổi bản sao của địa chỉ gốc. Để dễ hình dung hơn, chúng ta xem xét ví dụ sau:
* void setToNull(int \*ptr)
* {
* ptr = NULL; // (4)
* } // (5)
* int main()
* {
* int value = 5;
* int \*pValue = &value; // (1)
* cout << "pValue point to " << pValue << endl; // (2)
* setToNull(pValue); // (3)
* cout << "pValue point to " << pValue << endl; // (6)
* system("pause");
* return 0;
* }
* Xem tiếp...
* Có 6 bước để nói về đoạn chương trình trên:
* (1) Gán địa chỉ của biến value cho con trỏ pValue.  
  (2) In ra địa chỉ mà con trỏ pValue đang nắm giữ.  
  (3) Truyền giá trị của con trỏ đang nắm giữ cho hàm setToNull  
  (4) Sau khi con trỏ ptr trong hàm setToNull nhận được giá trị đầu vào, con trỏ ptr này được gán lại giá trị NULL.  
  (5) Ra khỏi phạm vi của hàm setToNull, con trỏ ptr bị hủy.  
  (6) In ra lại giá trị của con trỏ pValue. Lúc này, chúng ta có thể thấy giá trị của pValue không hề thay đổi, nó vẫn còn trỏ đến địa chỉ của biến value.
* Như vậy, giá trị địa chỉ được truyền vào hàm được nắm giữ bởi tham số con trỏ của hàm, từ đó chúng ta có thể sử dụng toán tử **dereference** để thao tác với vùng nhớ tại địa chỉ đó. Chúng ta cũng có thể cho tham số của hàm trỏ đến địa chỉ khác, nhưng không ảnh hưởng gì đến con trỏ gốc.
* [](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-8-con-tro-va-ham/0.png?raw=true)
* 0.png?raw=true959×551
* Trong một số trường hợp cụ thể, chúng ta muốn thay đổi địa chỉ của con trỏ đối số đang trỏ đến, chúng ta có thể sử dụng tham chiếu cho con trỏ đối số. Xét đoạn chương trình bên dưới:
* Con trỏ trỏ tới một biến được khai báo local trong hàm và hàm return biến đó sau đó hỏi kết quả return.
* Sử dụng con trỏ để truy cập vùng nhớ có địa chỉ dạng HEX (cần có từ khóa volatile để trình biên dịch không optimize biến).

Program stack nắm giữ những stack frames, đôi khi được gọi là activation records hoặc activation frames. Stack frame nắm giữ các parameters và local variables của một hàm. Phần heap quản lý bộ nhớ động (dynamic memory)

Hãy xem Figure 3-1 mô tả cách tổ chức stack và heap về mặt khái niệm. Mô tả này dựa trên đoạn code dưới đây:

**void** function2() {

Object \*var1 = ...;

**int** var2;

printf("Program Stack Example");

}

**void** function1() {

Object \*var3 = ...;

function2();

}

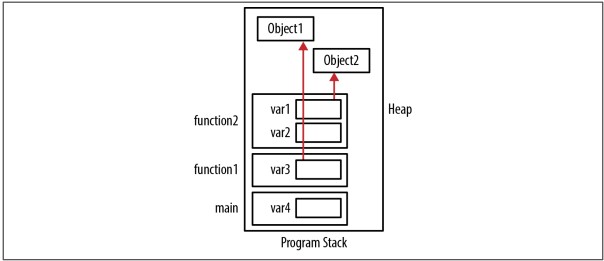
**int** main() {

**int** var4;

function1();

}

Theo những hàm được gọi, stack frame của chúng được đẩy (push) vào stack và stack sẽ phát triển “lên” (upward). Khi một hàm kết thúc (terminate), stack frame của nó được bốc (pop) ra khỏi program stack. Bộ nhớ sử dụng cho stack frame này không được xóa và có thể bị ghi đè ngay sau đó bởi những stack frame khác được push vào program stack.

Figure 3-1. Stack and heap

Khi bộ nhớ được cấp phát động, nó sẽ nằm ở vùng heap và có xu hướng phát triển “xuống” (downward). Heap sẽ bị phân mảng (fragment) do bộ nhớ được cấp phát (allocate) và giải phóng (deallocate) nhiều lần. Mặc dù heap thường phát triển xuống (đây là chiều phổ biến), nhưng bộ nhớ vẫn có thể cấp phát ở bất cứ đâu trong heap.

## Organization of a Stack Frame

Một stack frame bao gồm những phần tử sau:

* Địa chỉ trả về (return address): là địa chỉ trong chương trình, vị trí của hàm được trả về sau khi hoàn thành.
* Nơi lưu trữ dữ liệu cục bộ (storage for local data): bộ nhớ được cấp phát cho các biến cục bộ.
* Nơi lưu trữ tham số (storage for parameters): bộ nhớ được cấp phát cho các tham số của hàm.
* Con trỏ stack và base (stack and base pointers): những con trỏ được dùng khi chạy hệ thống (runtime system) để quản lý stack.

Lập trình viên C thường không quan tâm về con trỏ stack và base. Hai con trỏ này dùng để quản lý stack frame. Tuy nhiên, việc hiểu được chúng là gì và cách sử dụng sẽ giúp hiểu sâu hơn về program stack.

Con trỏ stack thường trỏ vào phần top của stack. Con trỏ stack base (con trỏ frame) thường tượng trưng và trỏ vào một địa chỉ trong stack frame, như là địa chỉ trả về (return address). Con trỏ này hỗ trợ truy cập các phần tử của stack frame. Hai con trỏ này đều không phải là con trỏ C. Chúng được dùng bởi runtime system để quản lý program stack. Nếu runtime system được viết bằng C, thì có thể những con trỏ này sẽ là con trỏ C thực sự.

Hãy xem xét quá trình tạo một stack frame cho hàm sau. Hàm này truyền vào một mảng số nguyên (array of integers), và một số nguyên là kích thước của mảng. Ba lệnh gọi hàm

printf

dùng để hiển thị địa chỉ các tham số và địa chỉ biến:

**float** average(**int** \*arr, **int** size) {

**int** sum;

printf("arr: %p\n",&arr);

printf("size: %p\n",&size);

printf("sum: %p\n",&sum);

**for**(**int** i=0; i<size; i++) {

sum += arr[i];

}

**return** (sum \* 1.0f) / size;

}

Khi thực thi, bạn nhận được kết quả tương tự như sau:

arr: 0x500

size: 0x504

sum: 0x480

Khoảng cách giữa parameter (

arr

,

size

) và local variable (

sum

) là do những phần tử khác của stack frame. Những phần tử này được dùng bởi runtime system để quản lý stack.

Khi stack frame được tạo, parameter được đẩy (push) vào theo thứ tự ngược lại với thứ tự khai báo của chúng, và nằm tiếp theo local variable. Điều này được mô tả ở Figure 3-2. Ở trường hợp này,

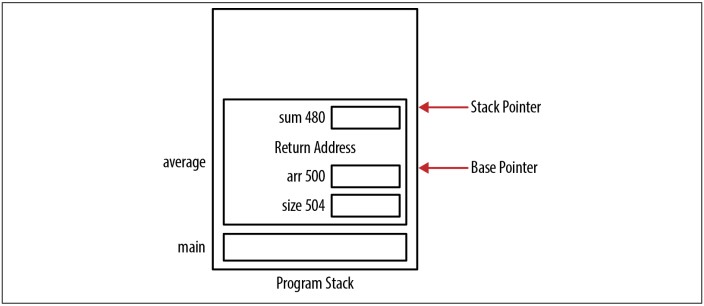
size

được đẩy vào sau

arr

. Thông thường, địa chỉ trả về (return address) cho hàm gọi (function call) được đẩy vào tiếp theo, và theo sau là các local variable.

Về mặt khái niệm, stack trong ví dụ này tăng “lên” (grow “up”). Tuy nhiên, những parameter, local variable và những stack frame mới khác được thêm vào tại địa chỉ thấp của bộ nhớ (lower memory address). Trong thực tế, hướng tăng của stack là tùy theo cách implement.

Figure 3-2. Stack frame example

Biến

i

 được dùng cho lệnh

for

 không nằm trong stack frame này. Ngôn ngữ C xem nó như một hàm “mini” và sẽ đẩy (push) và kéo (pop) nó theo cách phù hợp. Ở trường hợp này khối lệnh được đẩy vào program stack bên trên stack frame

average

khi xử lý tới nó và sẽ được kéo ra (pop off) khi nó kết thúc.

Mặc dù địa chỉ đúng có thể thay đổi khác nhau, nhưng thứ tự thường không. Đây là điều quan trọng nên hiểu, vì nó giúp ta giải thích được cách bộ nhớ cấp phát và thiết lập thứ tự liên quan của những parameter và variable. Điều này cũng hữu dụng khi debug lỗi liên quan tới con trỏ. Nếu ta không hiểu được stack frame được cấp phát như thế nào, thì sẽ không hiểu được ý nghĩa của phép gán địa chỉ.

Vì stack frame được đẩy vào program stack, hệ thống có thể bị hết bộ nhớ (run out of memory). Điều kiện này gọi là **stack overflow** và thường gây ra hậu quả là chương trình bị dừng bất thường. Hãy nhớ rằng mỗi **luồng (thread)**thường cấp phát program stack của riêng nó. Điều này dẫn đến nguy cơ xung đột nếu một hoặc nhiều thread truy cập vào cùng đối tượng (object) trong bộ nhớ.

# Passing and Returning by Pointer

Ở phần này, chúng ta sẽ tìm hiểu tác động của việc truyền con trỏ vào hàm và trả về con trỏ từ hàm. Truyền con trỏ (passing pointers) cho phép đối tượng tham chiếu (referenced object) có thể được truy cập bởi nhiều hàm mà không cần làm nó trở thành đối tượng toàn cục (global object). Có nghĩa là chỉ những hàm cần truy cập tới đối tượng sẽ có quyền truy cập và do đó không cần phải duplicate đối tượng này.

Nếu ta muốn dữ liệu (data) bị thay đổi trong một hàm, nó cần được truyền bởi con trỏ. Chúng ta có thể truyền dữ liệu bởi con trỏ và cấm thay đổi nó bằng cách truyền một con trỏ tới hằng số (a pointer to a constant), sẽ được nói đến ở phần **“Passing a Pointer to a Constant”**. Nếu dữ liệu cần thay đổi là một con trỏ, thì ta phải truyền một con trỏ tới con trỏ (a pointer to a pointer), sẽ được nói đến ở phần **“Passing a Pointer to a Pointer”**.

Parameter được truyền bằng giá trị, bao gồm cả con trỏ. Nghĩa là một bản copy của argument sẽ được truyền vào hàm. Truyền một con trỏ tới một argument sẽ có ích khi làm việc với cấu trúc dữ liệu lớn. Ví dụ, với một cấu trúc lớn tượng trưng cho nhân viên (employee). Nếu chúng ta truyền toàn bộ cấu trúc cho hàm, thì mỗi byte của cấu trúc sẽ cần được copy, dẫn đến làm chương trình chậm hơn và tốn nhiều bộ nhớ dùng cho stack frame. Truyền một con trỏ tới một object nghĩa là object không cần bị copy, và chúng ta có thể truy cập tới nó thông qua con trỏ.

## Passing Data Using a Pointer

Một trong nhưng lý do cơ bản để truyền dữ liệu (passing data) bằng con trỏ là để cho phép hàm có khả năng chỉnh sửa (modify) data. Đoạn code sau implement hàm hoán đổi (swap function), là hàm hoán đổi giá trị của 2 parameter với nhau. Đây là một hàm thông dụng được dùng cho các thuật toán sắp xếp. Ở đây, chúng ta sử dụng con trỏ integer và tham chiếu chúng để ảnh hưởng phép toán swap:

**void** swapWithPointers(**int**\* pnum1, **int**\* pnum2) {

**int** tmp;

tmp = \*pnum1;

\*pnum1 = \*pnum2;

\*pnum2 = tmp;

}

Đoạn code sau sử dụng hàm trên:

**int** main() {

**int** n1 = 5;

**int** n2 = 10;

swapWithPointers(&n1, &n2);

**return** 0;

}

Con trỏ

pnum1

và

pnum2

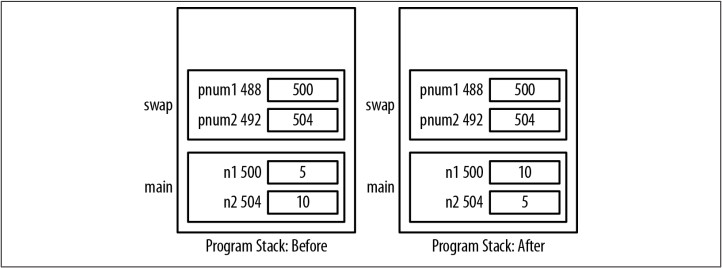
 được tham chiếu trong quá trình swap. Kết quả là giá trị của

n1

 và

n2

được chuyển đổi. Figure 3-3 mô tả cách tổ chức bộ nhớ. Ảnh trước (Before) là program stack lúc bắt đầu gọi hàm swap. Ảnh sau (After) là ngay trước lúc hàm return

Figure 3-3. Swapping with pointers

## Passing Data by Value

Nếu chúng ta không truyền bằng con trỏ, thì phép toán swap sẽ không thực hiện được. Trong hàm sau, hai số nguyên (integer) sẽ được truyền bằng giá trị (pass by value). Tiếp theo, trong hàm

main

, hai số nguyên sẽ được truyền cho hàm:

**void** swap(**int** num1, **int** num2) {

**int** tmp;

tmp = num1;

num1 = num2;

num2 = tmp;

}

**int** main() {

**int** n1 = 5;

**int** n2 = 10;

swap(n1, n2);

**return** 0;

}

Tuy nhiên, cách này sẽ vô dụng bởi vì số nguyên được truyền vào là giá trị, không phải con trỏ. Chỉ là bản copy của các argument lưu vào

num1

và

num2

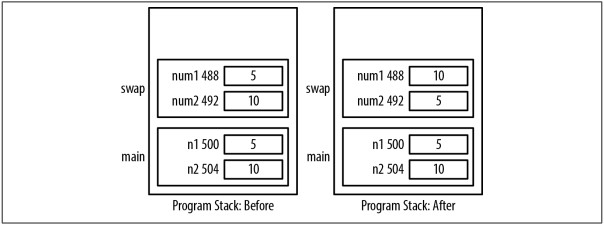
. Nếu ta thay đổi

num1

, thì argument

n1

vẫn không đổi. Khi ta thay đổi giá trị của các parameter, chúng ta không thay đổi được giá trị của các argument gốc. Figure 3-4 mô tả cách bộ nhớ được cấp phát cho parameter.

Figure 3-4. Pass by value

## Passing a Pointer to a Constant

Truyền con trỏ tới một hằng số (constant) là một kĩ thuật thông dụng trong C. Cách làm này rất hiệu quả vì chúng ta chỉ truyền địa chỉ của data và có thể tránh việc copy toàn bộ vùng nhớ, trong một số trường hợp data khá lớn. Tuy nhiên, với một con trỏ thông thường, data có thể bị sửa đổi và chúng ta không muốn thế. Khi xét đến vấn đề này thì việc truyền một con trỏ tới một constant chính là giải pháp.

Trong ví dụ này, chúng ta truyền một con trỏ tới một hằng số nguyên (a pointer to a constant integer) và một con trỏ tới một số nguyên (a pointer to an integer), chúng ta không thể thay đổi giá trị được truyền vào bởi con trỏ tới một constant:

**void** passingAddressOfConstants(const **int**\* num1, **int**\* num2) {

\*num2 = \*num1;

}

**int** main() {

const **int** limit = 100;

**int** result = 5;

passingAddressOfConstants(&limit, &result);

**return** 0;

}

Sẽ không có lỗi cú pháp nào xuất hiện, và hàm sẽ gán 100 cho biến

result

.

Ở phiên bản sau của hàm này, chúng ta thử thay đổi giá trị của cả hai số nguyên được tham chiếu như sau:

**void** passingAddressOfConstants(const **int**\* num1, **int**\* num2) {

\*num1 = 100;

\*num2 = 200;

}

Cách dùng dưới đây sẽ gây ra lỗi nếu chúng ta truyền biến hằng số

limit

cho hàm hai lần:

const **int** limit = 100;

passingAddressOfConstants(&limit, &limit);

Khi biên dịch nó sẽ báo lỗi cú pháp (syntax error) do kiểu biến không phù hợp (type mismatch) giữa parameter thứ hai và argument của nó. Ngoài ra, nó còn báo rằng chúng ta đang cố ý thay đổi một hằng số được tham chiếu bởi parameter thứ nhất.

Hàm kỳ vọng một con trỏ tới số nguyên (a pointer to an integer), nhưng thay vào đó là một con trỏ tới hằng số nguyên (a pointer to an integer constant) đã được truyền vào. Chúng ta không thể truyền địa chỉ của hằng số nguyên (address of an integer constant) cho một con trỏ tới một hằng số. Vì nó sẽ cho phép thay đổi giá trị hằng số.

Việc truyền trực tiếp địa chỉ của hằng số nguyên như sau cũng sẽ gây ra lỗi cú pháp:

passingAddressOfConstants(&23, &23);

Lỗi chỉ ra rằng yêu cầu một

<strong>lvalue </strong>

làm toán tử (operand) cho phép toán địa chỉ (address-of operator).

## Returning a Pointer

Muốn trả về con trỏ, ta chỉ việc khai báo kiểu trả về (return type) là một con trỏ tới kiểu dữ liệu tương ứng. Nếu bạn muốn hàm trả về một đối tượng (object), thì có 2 kỹ thuật thường dùng như sau:

* Cấp phát bộ nhớ bên trong hàm bằng hàm

malloc

 và trả về địa chỉ của nó. Người gọi có trách nhiệm giải phóng bộ nhớ được trả về.

* Truyền một object cho hàm để modify. Người gọi có trách nhiệm cấp phát và giải phóng bộ nhớ của object.

Đầu tiên, hãy xem trường hợp sử dụng hàm

malloc

 để cấp bộ nhớ trả về. Ví dụ dưới đây mô phỏng việc trả về một con trỏ tới một local object. Cách thứ hai không được khuyến khích sử dụng, nó sẽ được thảo luận ở phần **“Passing Null Pointers”**.

Trong ví dụ, chúng ta định nghĩa một hàm có parameter truyền vào là kích thước của mảng số nguyên và một giá trị để khởi tạo cho mỗi phần tử. Hàm này có nhiệm vụ cấp phát bộ nhớ cho một mảng số nguyên, khởi tạo mảng với giá trị được truyền, sau đó trả về địa chỉ của mảng:

**int**\* allocateArray(**int** size, **int** value) {

**int**\* arr = (**int**\*)malloc(size \* **sizeof**(**int**));

**for**(**int** i=0; i<size; i++) {

arr[i] = value;

}

**return** arr;

}

Sử dụng hàm đó như sau:

**int**\* vector = allocateArray(5,45);

**for**(**int** i=0; i<5; i++) {

printf("%d\n", vector[i]);

}

Figure 3-5 vẽ cách bộ nhớ được cấp phát cho hàm này. Ảnh trước (Before) cho thấy trạng thái của chương trình ngay trước khi return. Ảnh sau (After) là trạng thái của chương trình sau khi hàm return. Lúc này, biến

vector

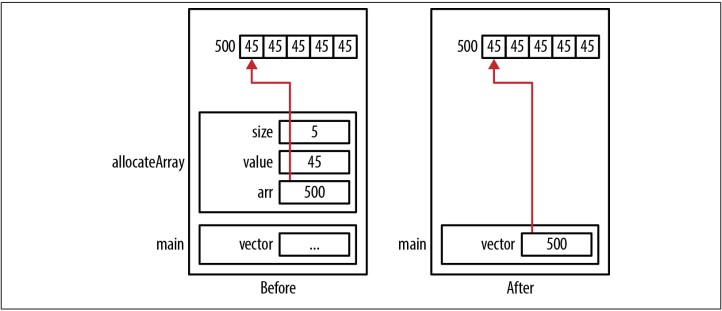
 chứa địa chỉ của bộ nhớ được cấp phát (500) trong hàm

allocateArray

. Khi hàm kết thúc, biến

arr

 bị xóa, bộ nhớ ở địa chỉ 500 được tham chiếu bởi con trỏ không bị mất. Người dùng phải nhớ free bộ nhớ này sau khi dùng xong.

Figure 3-5. Returning a pointer

Mặc dù ví dụ này hoạt động đúng, tuy nhiên nó tiềm ẩn vài vấn đề có thể xảy ra khi một hàm return con trỏ, bao gồm:

* Trả về một con trỏ chưa khởi tạo (uninitialized pointer)
* Trả về một con trỏ tới địa chỉ không hợp lệ (pointer to an invalid address)
* Trả về một con trỏ tới một biến cục bộ (pointer to a local variable)
* Trả về một con trỏ nhưng bị lỗi khi giải phóng (free) nó

Vấn đề cuối thường xảy ra ở hàm

allocateArray

. Trả về bộ nhớ cấp phát động từ hàm nghĩa là người gọi hàm chịu trách nhiệm trong việc giải phóng bộ nhớ của nó. Như cách làm dưới đây:

**int**\* vector = allocateArray(5,45);

...

free(vector);

Chúng ta phải free nó sau khi sử dụng xong. Nếu không thì chúng ta sẽ bị lỗi rò rỉ bộ nhớ (memory leak).

## Pointers to Local Data

Trả về con trỏ tới một dữ liệu cục bộ (local data) là một sai lầm dễ mắc phải nếu bạn không hiểu program stack hoạt động như thế nào. Trong ví dụ tiếp theo, chúng ta sẽ lấy và sửa lại hàm allocateArray ở phần **“Returning a Pointer”**. Thay vì cấp phát bộ nhớ động (dynamically allocating memory) cho mảng, chúng ta sẽ dùng một mảng cục bộ (local array):

**int**\* allocateArray(**int** size, **int** value) {

**int** arr[size];

**for**(**int** i=0; i<size; i++) {

arr[i] = value;

}

**return** arr;

}

Tuy nhiên, địa chỉ của mảng được trả về sẽ không còn hợp lệ một khi hàm return. Bởi vì stack frame của hàm bị lấy ra (pop off) khỏi stack. Mặc dù mỗi phần tử của mảng vẫn lưu giá trị 45, tuy nhiên giá trị này cũng có khả năng bị ghi đè (overwrite) nếu hàm khác được gọi. Điều này được thể hiện ở ví dụ sau. Trong này, hàm

printf

được gọi liên tục, khiến cho mảng bị lỗi corruption:

**int**\* vector = allocateArray(5,45);

**for**(**int** i=0; i<5; i++) {

printf("%d\n", vector[i]);

}

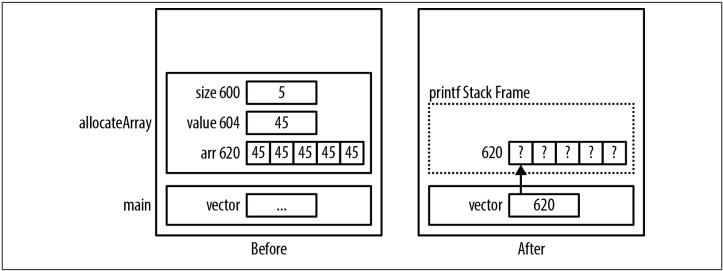
Figure 3-6 mô tả cách bộ nhớ được cấp phát như thế nào khi lỗi xảy ra. Ô nét chấm cho thấy chỗ từng là stack frame của hàm

allocateArray

 có thể trở thành stack frame của những hàm khác, như là hàm

printf

, có thể bị đẩy (push) vào program stack, gây corrupt bộ nhớ đang dùng của mảng. Nội dung thực sự của stack frame đó phụ thuộc theo cách làm (implementation-dependent).

Figure 3-6. Returning a pointer to local data

Một cách tiếp cận khác là khai báo biến

arr

 làm static. Nó sẽ nâng scope của biến ra khỏi phạm vi hàm và allocate bên ngoài stack frame, loại trừ khả năng bị những hàm khác ghi đè (overwrite) giá trị lên biến này:

**int**\* allocateArray(**int** size, **int** value) {

**static** **int** arr[5];

...

}

Tuy nhiên, cách này không phải lúc nào cũng nên dùng. Mỗi lần hàm allocateArray được gọi, nó sẽ dùng lại chính mảng này. Điều này vô hiệu hóa mọi lệnh gọi tới hàm trước đó. Ngoài ra, vì mảng static phải được khai báo với kích thước cố định, do đó sẽ hạn chế năng lực của hàm khi xử lý các kích thước khác nhau.

Nếu chỉ cần hàm trả về vài giá trị cần thiết và không gây tổn hại gì khi chia sẻ chúng. Ta có thể duy trì một danh sách giá trị này và trả về cái tương ứng. Sẽ hữu ích nếu chúng ta cần trả về tin nhắn kiểu trạng thái (a status type message), như một error number, là một giá trị không cần thay đổi.

## Passing Null Pointers

Trong phiên bản tiếp theo của hàm

allocateArray

, một con trỏ tới mảng được truyền vào cùng với kích thước mảng và một giá trị để khởi tạo cho mỗi phần tử của mảng. Hàm trả về con trỏ cho thuận tiện. Tuy nhiên phiên bản này của hàm không cấp phát bộ nhớ, việc này sẽ được làm ở phiên bản sau nữa:

**int**\* allocateArray(**int** \*arr, **int** size, **int** value) {

**if**(arr != **NULL**) {

**for**(**int** i=0; i<size; i++) {

arr[i] = value;

}

}

**return** arr;

}

Một thói quen tốt khi truyền một con trỏ vào hàm là luôn luôn kiểm tra khác **null** trước khi sử dụng nó.

Sử dụng hàm như sau:

**int**\* vector = (**int**\*)malloc(5 \* **sizeof**(**int**));

allocateArray(vector,5,45);

Nếu con trỏ là NULL, thì không hành động nào được thực hiện, do đó chương trình sẽ không bị dừng do lỗi bất thường.

## Passing a Pointer to a Pointer

Khi con trỏ (pointer) được truyền vào hàm, nó được truyền bằng giá trị (passed by value). Nếu chúng ta muốn điều chỉnh (modify) một con trỏ gốc (original pointer), chứ không phải bản copy của nó. Chúng ta phải truyền nó như một con trỏ tới một con trỏ (a pointer to a pointer). Trong ví dụ này, một con trỏ tới mảng integer được truyền vào hàm. Con trỏ này sẽ được gán bộ nhớ (assign memory) và khởi tạo (initialize). Hàm sẽ trả về bộ nhớ được cấp phát (allocated memory) cho parameter thứ nhất. Trong hàm này, đầu tiên chúng ta cấp phát bộ nhớ rồi khởi tạo nó. Địa chỉ của bộ nhớ được cấp phát này được gán cho một con trỏ int (a pointer to an

int

). Để điều chỉnh con trỏ trong một hàm, ta phải truyền địa chỉ của con trỏ. Do đó, phải khai báo parameter là con trỏ tới con trỏ kiểu int (a pointer to a pointer to an

int

). Khi gọi hàm, ta cần truyền vào địa chỉ con trỏ:

**void** allocateArray(**int** \*\*arr, **int** size, **int** value) {

\*arr = (**int**\*)malloc(size \* **sizeof**(**int**));

**if**(\*arr != **NULL**) {

**for**(**int** i=0; i<size; i++) {

\*(\*arr+i) = value;

}

}

}

Hãy test hàm bằng đoạn code sau:

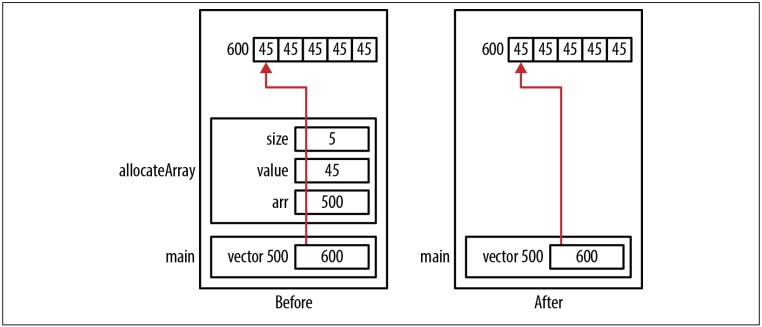
**int** \*vector = **NULL**;

allocateArray(&vector,5,45);

Parameter đầu tiên của hàm allocateArray() là một con trỏ tới con trỏ tới một integer (a pointer to a pointer to an integer). Khi gọi hàm, chúng ta cần truyền vào một giá trị kiểu này. Ta có thể truyền vào địa chỉ của biến vector. Địa chỉ trả về từ malloc được gán cho arr. Tham chiếu một con trỏ tới một con trỏ tới một integer (dereference a pointer to a pointer to an integer) cho kết quả là một con trỏ tới một integer (a pointer to an integer). Vì đây là địa chỉ của vector, do đó sẽ modify được vector.

Quá trình cấp phát bộ nhớ mô tả trong Figure 3-7. Ảnh trước (Before) vẽ stack sau khi malloc return và mảng được khởi tạo. Trong khi, ảnh sau (After) vẽ stack sau khi hàm return.

Để dễ xác định vấn đề như ***rò rỉ bộ nhớ*** (memory leak), hãy vẽ sơ đồ cấp phát bộ nhớ.

Figure 3-7. Passing a pointer to a pointer

Phiên bản dưới đây của hàm mô tả tại sao truyền một con trỏ đơn thuần sẽ không hiệu quả:

**void** allocateArray(**int** \*arr, **int** size, **int** value) {

arr = (**int**\*)malloc(size \* **sizeof**(**int**));

**if**(arr != **NULL**) {

**for**(**int** i=0; i<size; i++) {

arr[i] = value;

}

}

}

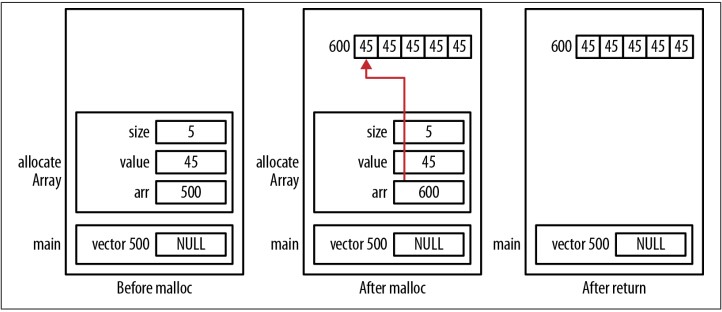
Cách sử dụng hàm:

**int** \*vector = **NULL**;

allocateArray(&vector,5,45);

printf("%p\n", vector);

Khi chương trình chạy, bạn sẽ thấy kết quả hiển thị là 0x0. Bởi vì khi vector được truyền vào hàm, chỉ là giá trị được copy vào parameter arr. Do đó modify arr không ảnh hưởng lên vector. Khi hàm return, giá trị lưu trong arr không được copy vào vector. Figure 3-8 vẽ quá trình cấp phát bộ nhớ. Ảnh **Before malloc** cho thấy trạng thái bộ nhớ ngay trước khi arr được gán giá trị mới. Ảnh **After malloc** vẽ trạng thái bộ nhớ sau khi hàm malloc thực thi trong hàm allocateArray và mảng được khởi tạo. Biến arr được modify để chỉ vào vùng địa chỉ mới trong heap. Ảnh **After return** là trạng thái bộ nhớ sau khi hàm return. Ngoài ra, chúng ta còn bị **memory leak** vì đã làm mất truy cập tới cụm bộ nhớ ở địa chỉ 600.

Figure 3-8. Passing pointers

### Viết hàm free của riêng bạn (Writing your own free function)

Hàm free() không kiểm tra con trỏ truyền vào có khác NULL hay không, và cũng không set giá trị NULL cho con trỏ trước khi return để tránh bị Dangling Pointer. Điều này thường gây ra lỗi ‘Segmentations faults’ hoặc ‘double free’. Do đó để an toàn người ta thường tự viết thêm hàm free riêng để giữ cho chương trình không bị corrupt.

Dựa trên những nền tảng đã thảo luận ở phần **“Passing and  Returning by Pointer”** , chúng ta sẽ tạo một hàm free riêng có khả năng gán NULL cho con trỏ sau khi free. Hàm này cần sử dụng con trỏ tới con trỏ như dưới đây:

**void** saferFree(**void** \*\*pp) {

**if** (pp != **NULL** && \*pp != **NULL**) {

free(\*pp);

\*pp = **NULL**;

}

}

Hàm

saferFree

gọi hàm

free

để thực sự deallocate memory. Nó khai báo sử dụng parameter là một con trỏ tới con trỏ kiểu

void

(a pointer to a pointer to

void

). Sử dụng con trỏ tới con trỏ cho phép ta sửa đổi (modify) con trỏ truyền vào. Sử dụng kiểu

void

cho phép truyền bất kỳ kiểu dữ liệu nào vào hàm. Tuy nhiên, chúng ta sẽ nhận một warning nếu không ép kiểu tường minh (explicitly cast) thành kiểu con trỏ tới

void

mỗi khi gọi hàm. Nếu ta nhớ ép kiểu tường minh thì warning sẽ biến mất.

Định nghĩa thêm macro

safeFree

như bên dưới, gọi hàm

safeFree

cùng với ép kiểu

void

và sử dụng toán tử địa chỉ (address-of operator).

#define safeFree(p) saferFree((void\*\*)&(p))

Sử dụng macro như sau:

**int** main() {

**int** \*pi;

pi = (**int**\*) malloc(**sizeof**(**int**));

\*pi = 5;

printf("Before: %p\n",pi);

safeFree(pi);

printf("After: %p\n",pi);

safeFree(pi);

**return** (EXIT\_SUCCESS);

}

Giả sử hàm

malloc

trả về bộ nhớ ở địa chỉ 1000, output print ra màn hình sẽ là 1000, tiếp sau là 0. Lần thứ hai gọi macro

safeFree

với giá trị NULL không làm ngưng chương trình, do hàm đã phát hiện và bỏ qua nó.

# Function Pointers

Con trỏ hàm là con trỏ chứa địa chỉ của hàm. Con trỏ có khả năng trỏ đến hàm đưa ra một tính năng quan trọng và hữu ích của C. Nó cung cấp một cách khác để thực thi các hàm theo một thứ tự không cụ thể tại thời điểm biên dịch và không phải sử dụng các câu lệnh điều kiện.

Tuy nhiên, một mối lo ngại về sử dụng con trỏ hàm là nguy cơ chương trình chạy chậm hơn. Bộ xử lý có thể không sử dụng được dự đoán nhánh (branch prediciton) kết hợp với đường ống (pipelining). Branch prediciton là một kỹ thuật nhờ đó bộ xử lý sẽ đoán xem chuỗi thực thi (multiple execution sequences) nào sẽ được thực hiện. Pipelining là một công nghệ phần cứng thường được sử dụng để cải thiện hiệu suất của bộ xử lý và đạt được bằng cách thực hiện lệnh chồng chéo (overlapping instruction execution). Trong sơ đồ này, bộ xử lý sẽ bắt đầu xử lý nhánh mà nó tin rằng sẽ được thực thi. Nếu bộ xử lý dự đoán thành công nhánh đúng, thì các lệnh (instruction) hiện tại trong đường ống (pipeline) sẽ không phải bị loại bỏ.

Sự chậm lại này có thể xảy ra hoặc không. Việc sử dụng con trỏ hàm trong các tình huống như tra cứu bảng (table lookup) có thể giảm thiểu các vấn đề về hiệu suất. Trong phần này, ta sẽ tìm hiểu cách khai báo (declare) con trỏ hàm, xem cách chúng được sử dụng để hỗ trợ các đường dẫn thực thi thay thế (alternate execution path) và khám phá các kỹ thuật khai thác tiềm năng của chúng.

## Declaring Function Pointers

Cú pháp khai báo (declare) một con trỏ tới một hàm (a pointer to a function) có thể gây nhầm lẫn khi bạn lần đầu tiên nhìn thấy nó. Cũng như nhiều khía cạnh của C, một khi bạn đã quen với ký hiệu, mọi thứ sẽ bắt đầu đâu vào đó. Hãy bắt đầu với một khai báo đơn giản. Dưới đây, chúng ta khai báo một con trỏ tới một hàm được truyền void và trả về void:

**void** (\*foo)();

Khai báo này trông giống như một nguyên mẫu hàm (function prototype). Nếu chúng ta loại bỏ bộ dấu ngoặc đơn đầu tiên, nó sẽ có vẻ là nguyên mẫu hàm cho hàm

foo

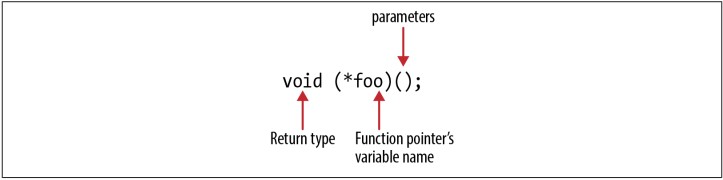
, hàm này được truyền

void

và trả về một con trỏ tới

void

. Tuy nhiên, dấu ngoặc đơn làm cho nó trở thành một con trỏ hàm có tên là foo. Dấu hoa thị (\*) cho biết đó là một con trỏ. Figure 3-9 mô tả cụ thể các phần của khai báo con trỏ hàm.

Figure 3-8. Passing pointers

*Khi con trỏ hàm được dùng, người lập trình phải cẩn thận đảm bảo rằng nó được dùng đúng. Bởi vì C không kiểm tra các tham số (parameter) được truyền có đúng không.*

Một số ví dụ khai báo con trỏ hàm như dưới đây:

**int** (\*f1)(**double**); // Passed a double and

// returns an int

**void** (\*f2)(**char**\*); // Passed a pointer to char and

// returns void

**double**\* (\*f3)(**int**, **int**); // Passed two integers and

// returns a pointer to a double

Đừng nhầm lẫn hàm trả về một con trỏ (function that return a pointer) với con trỏ hàm (function pointers). Dưới đây khai báo

f4

là một hàm trả về một con trỏ tới một số nguyên, trong khi

f5

là một con trỏ hàm trả về một số nguyên. Biến

f6

là một con trỏ hàm trả về một con trỏ tới một số nguyên:

**int** \*f4();

**int** (\*f5)();

**int**\* (\*f6)();

Khoảng trắng trong các biểu thức này có thể được sắp xếp lại để nó đọc như sau:

**int**\* f4();

**int** (\*f5)();

Rõ ràng

f4

là một hàm trả về một con trỏ tới một số nguyên. Tuy nhiên, khi sử dụng dấu ngoặc đơn với

f5

để liên kết rõ ràng dấu hoa thị “con trỏ” với tên hàm, làm cho nó trở thành một con trỏ hàm.

## Using a Function Pointer

Dưới đây là một ví dụ đơn giản về sử dụng con trỏ hàm, trong đó một hàm được truyền vào một số nguyên và trả về một số nguyên. Chúng ta cũng định nghĩa (define) một hàm

square

để bình phương một số nguyên và sau đó trả về giá trị bình phương. Để đơn giản hóa các ví dụ này, chúng ta bỏ qua khả năng tràn số nguyên (integer overflow).

**int** (\*fptr1)(**int**);

**int** square(**int** num) {

**return** num\*num;

}

Để sử dụng con trỏ hàm thực thi hàm

square

, chúng ta cần gán địa chỉ của hàm

square

cho con trỏ hàm, như code bên dưới. Cũng giống như tên mảng, khi chúng ta sử dụng tên của chính hàm đó, nó sẽ trả về địa chỉ của hàm đó. Chúng ta cũng khai báo một số nguyên mà chúng ta sẽ truyền vào hàm:

**int** n = 5;

fptr1 = square;

printf("%d squared is %d\n",n, fptr1(n));

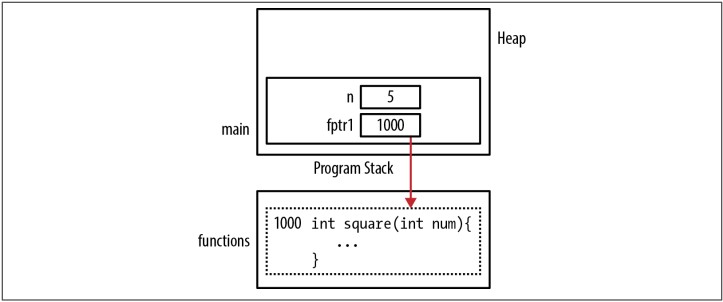
Khi thực hiện nó sẽ hiển thị: “5 squared is 25”. Chúng ta có thể sử dụng toán tử địa chỉ (address-of) với tên hàm như sau, nhưng nó không cần thiết và dư thừa. Trình biên dịch sẽ bỏ qua toán tử address-of một cách hiệu quả khi được sử dụng trong ngữ cảnh này.

fptr1 = &square;

Figure 3-10 minh họa cách cấp phát bộ nhớ cho ví dụ này. Ở đây đặt hàm

square

bên dưới program stack. Đây chỉ là mục đích minh họa. Các hàm được allocate trong một segment khác với segment sử dụng bởi program stack. Vị trí thực tế của hàm thường không được quan tâm.

Figure 3-10. Location of functions

Người ta thường khai báo một định nghĩa kiểu (type definition) cho con trỏ hàm để sử dụng thuận tiện hơn. Cách làm như dưới đây. Định nghĩa kiểu như vậy có vẻ hơi lạ. Thông thường, tên của định nghĩa kiểu là phần tử cuối cùng của khai báo:

**typedef** **int** (\*funcptr)(**int**);

...

funcptr fptr2;

fptr2 = square;

printf("%d squared is %d\n",n, fptr2(n));

## Passing Function Pointers

Việc truyền một con trỏ hàm khá dễ thực hiện. Chỉ cần sử dụng khai báo con trỏ hàm làm tham số (parameter) của hàm. Chúng ta sẽ làm ví dụ truyền một con trỏ hàm bằng cách sử dụng các hàm

add

,

sub

và

compute

như được khai báo bên dưới:

**int** add(**int** num1, **int** num2) {

**return** num1 + num2;

}

**int** subtract(**int** num1, **int** num2) {

**return** num1 - num2;

}

**typedef** **int** (\*fptrOperation)(**int**,**int**);

**int** compute(fptrOperation operation, **int** num1, **int** num2) {

**return** operation(num1, num2);

}

Hàm được sử dụng như sau:

printf("%d\n",compute(add,5,6));

printf("%d\n",compute(sub,5,6));

Output sẽ là 11 và –1. Địa chỉ của hàm

add

và

sub

được truyền đến hàm

compute

. Những địa chỉ này sẽ được sử dụng để gọi phép tính tương ứng. Ví dụ này cũng chỉ ra cách viết code linh hoạt hơn thông qua sử dụng con trỏ hàm.

## Returning Function Pointers

Để return con trỏ hàm ta cần khai báo kiểu trả về (return type) của hàm là một con trỏ hàm (function pointer). Chúng ta sẽ sử dụng lại hàm

add

và

sub

cùng với định nghĩa kiểu đã tạo ra trong phần **“Passing Function Pointers”**.

Chúng ta sẽ sử dụng hàm

select

sau đây để return một con trỏ hàm cho một phép toán dựa trên một ký tự input. Nó sẽ return một con trỏ tới hàm

add

hoặc hàm

subtract

, tùy thuộc vào opcode được truyền:

fptrOperation select(**char** opcode) {

**switch**(opcode) {

**case** '+': **return** add;

**case** '-': **return** subtract;

}

}

Hàm

evaluate

liên kết các hàm này lại với nhau. Hàm được truyền hai số nguyên (integer) và một ký tự (character) đại diện cho phép tính sẽ được thực hiện. Nó truyền opcode tới hàm

select

, hàm này trả về một con trỏ tới hàm để thực thi. Trong câu lệnh return, nó thực thi hàm này và trả về kết quả:

**int** evaluate(**char** opcode, **int** num1, **int** num2) {

fptrOperation operation = select(opcode);

**return** operation(num1, num2);

}

Những hàm

printf

sau sẽ in kết quả khi sử dụng hàm này:

printf("%d\n",evaluate('+', 5, 6));

printf("%d\n",evaluate('-', 5, 6));

Output sẽ là 11 và –1.

## Using an Array of Function Pointers

Chúng ta còn có thể sử dụng mảng con trỏ hàm (array of function pointers) để chọn hàm cần thiết cho việc evaluate. Sử dụng khai báo con trỏ hàm từ

typedef

làm kiểu của mảng. Mảng được khởi tạo với tất cả phần tử là

NULL

. Cách viết như dưới đây sẽ khởi tạo các phần tử trong mảng với cùng một giá trị. Nếu số lượng giá trị trong ngoặc nhọn nhỏ hơn kích thước của mảng, thì những giá trị này sẽ được sử dụng để khởi tạo mọi phần tử của mảng:

**typedef** **int** (\*operation)(**int**, **int**);

operation operations[128] = {**NULL**};

Ngoài ra, chúng ta có thể khai báo mà không cần dùng

typedef

như sau;

**int** (\*operations[128])(**int**, **int**) = {**NULL**};

Mục đích của mảng này là cho phép tra bằng ký tự danh mục (character index) để chọn hàm tương ứng tính toán. Ví dụ: ký tự ‘\*’ sẽ tra ra hàm nhân (multiplication) nếu nó tồn tại. Chúng ta sử dụng chỉ mục ký tự vì mỗi ký tự bằng chữ cũng là một số nguyên. 128 phần tử tương là ứng với 128 ký tự ASCII đầu tiên. Chúng ta sử dụng định nghĩa này cùng với các hàm

add

và

subtract

đã được develop ở phần **“Returning Function Pointers”**.

Khởi tạo mảng cho tất cả phần tử là

NULL

rồi, gán các hàm

add

và

subtract

cho các phần tử tương ứng với các dấu cộng và dấu trừ:

**void** initializeOperationsArray() {

operations['+'] = add;

operations['-'] = subtract;

}

Hàm

evaluate

trước đó được viết lại thành

evaluateArray

. Thay vì gọi hàm

select

để lấy con trỏ hàm, chúng ta sử dụng mảng con trỏ hàm

operations

với ký tự phép toán (operation character) làm chỉ mục (index):

**int** evaluateArray(**char** opcode, **int** num1, **int** num2) {

fptrOperation operation;

operation = operations[opcode];

**return** operation(num1, num2);

}

Kiểm tra hoạt động của hàm như sau:

initializeOperationsArray();

printf("%d\n",evaluateArray('+', 5, 6));

printf("%d\n",evaluateArray('-', 5, 6));

Kết quả thực thi chuỗi lệnh trên là 11 và –1. Ở phiên bản an toàn hơn của hàm

evaluateArry

, nên kiểm tra null các con trỏ hàm trước khi cố thực thi hàm đó.

## Comparing Function Pointers

Con trỏ hàm có thể được so sánh với nhau bằng cách toán tử đẳng thức (equality) và bất đẳng thức (inequality). Trong ví dụ sau, chúng ta sử dụng định nghĩa kiểu

fptrOperation

và hàm

add

đã tạo từ phần **“Passing Function Pointers”**. Hàm

add

được gán cho con trỏ hàm

fptr1

, sau đó so sánh với địa chỉ của hàm

add

:

fptrOperation fptr1 = add;

**if**(fptr1 == add) {

printf("fptr1 points to add function\n");

} **else** {

printf("fptr1 does not point to add function\n");

}

Khi chạy đoạn code này, output sẽ chứng minh rằng con trỏ trỏ đến hàm

add

.

Trong thực tế việc so sánh các con trỏ hàm thường được áp dụng cho mảng các con trỏ hàm tượng trưng cho các bước (step) của một tác vụ (task).

Ví dụ chúng ta có thể áp dụng một chuỗi hàm để thao tác với một mảng hàng tồn kho (array of inventory parts). Một tập các thao tác (a set of operations) có thể là sắp xếp các phần, tính toán tổng tích lũy của số lượng, sau đó hiển thị mảng và hiển thị tổng. Tập operation thứ hai có thể là hiển thị mảng, tìm giá đắt nhất và giá rẻ nhất, sau đó hiển thị hiệu số của chúng.

**typedef** <return\_type> (operation\*) (...);

operation task1[] = {

sort\_the\_parts,

calculate\_cumulative\_sum,

display\_array, // log operation

display\_sum,

}

operation task2[] = {

display\_array, // log operation

find\_max\_expensive,

find\_min\_expensive,

display\_differene,

};

**int** index = 0;

**while**(1)

{

// find log operation

**if**(task1[index] == display\_array)

{

task1[index] = **NULL**; // remove log operation

}

index++;

}

Mỗi operation có thể được định nghĩa bằng một mảng các con trỏ tới từng hàm. Thao tác ghi log (log operation) có thể tồn tại trong cả hai danh sách. Khả năng so sánh hai con trỏ hàm sẽ cho phép sự thay đổi động (dynamic modification) một operation bằng cách delete operation, chẳng hạn như ghi log, bằng cách tìm (find) và sau đó xóa (remove) hàm khỏi danh sách.

## Casting Function Pointers

Một con trỏ hàm có thể được ép kiểu sang loại khác (cast to another type). Điều này nên được thực hiện cẩn thận vì khi hệ thống chạy (runtime system), nó không xác minh được các tham số (parameter) được sử dụng bởi một con trỏ hàm là chính xác hay không. Ta cũng có thể ép kiểu con trỏ hàm thành một kiểu con trỏ hàm khác rồi quay lại. Con trỏ kết quả sẽ bằng với con trỏ ban đầu. Kích thước của các con trỏ hàm được sử dụng không nhất thiết phải giống nhau. Đoạn code sau minh họa việc này:

**typedef** **int** (\*fptrToSingleInt)(**int**);

**typedef** **int** (\*fptrToTwoInts)(**int**,**int**);

**int** add(**int**, **int**);

fptrToTwoInts fptrFirst = add;

fptrToSingleInt fptrSecond = (fptrToSingleInt)fptrFirst;

fptrFirst = (fptrToTwoInts)fptrSecond;

printf("%d\n",fptrFirst(5,6));

Kết quả in ra sẽ là 11.

*Chuyển đổi giữa con trỏ hàm (function pointer) và con trỏ dữ liệu (pointer to data) không đảm bảo sẽ hoạt động đúng.*

Việc sử dụng

void\*

 không đảm bảo sẽ hoạt động đúng với con trỏ hàm. Do đó, chúng ta không nên gán một con trỏ hàm cho

void\*

 như sau:

**void** (\*fptr)() = func;

**void**\* pv = fptr;

Tuy nhiên, khi chuyển đổi các con trỏ hàm, người ta thường thấy một loại con trỏ hàm “cơ sở” (base) như được khai báo bên dưới. Điều này khai báo

fptrBase

là một con trỏ hàm tới một hàm, được truyền

void

và trả về

void

:

**typedef** **void** (\*fptrBase)();

Đoạn code này mô tả việc sử dụng con trỏ base, ví dụ này được lặp lại từ ví dụ trước:

fptrBase basePointer;

fptrFirst = add;

basePointer = (fptrToSingleInt)fptrFirst;

fptrFirst = (fptrToTwoInts)basePointer;

printf("%d\n",fptrFirst(5,6));

Con trỏ base được dùng như một vật giữ chỗ (placeholder) để hoán chuyển giá trị con trỏ hàm.

*Luôn luôn chắc chắn rằng bạn sử dụng đúng danh sách đối số (argument list) cho con trỏ hàm. Sử dụng sai có thể gây ra lỗi hành vi không xác định (indeterminate behavior).*

# Summary

 Hiểu cấu trúc program stack và heap góp phần giúp chúng ta nắm vững và có thể mô tả chi tiết về cách thức hoạt động của một chương trình và cách hoạt động của con trỏ. Chúng ta cũng đã tìm hiểu về stack, heap và stack frame. Những khái niệm này giúp giải thích cơ chế truyền (pass) con trỏ đến một hàm và trả lại (return) con trỏ từ một hàm.

Ví dụ việc return một con trỏ tới một biến cục bộ (local variable) là tệ vì bộ nhớ được cấp phát cho biến cục bộ sẽ bị ghi đè bởi các lệnh gọi hàm tiếp theo. Truyền một con trỏ tới constant data là việc làm hiệu quả và ngăn hàm sửa đổi dữ liệu được truyền. Truyền một con trỏ tới một con trỏ (a pointer to a pointer) cho phép con trỏ đối số (argument pointer) được gán lại địa chỉ khác trong bộ nhớ. Nhờ có stack và heap đã giúp minh họa chi tiết chức năng này.

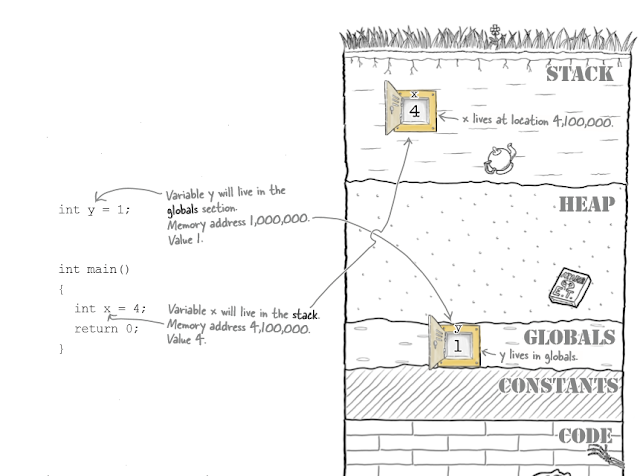
Con trỏ hàm cũng đã được giới thiệu và giải thích. Loại con trỏ này rất hữu ích để kiểm soát trình tự thực thi các hàm trong một ứng dụng bằng khả năng thay thế hàm cần thực thi dựa trên nhu cầu của ứng dụng.

* Ép kiểu con trỏ để truy cập bộ nhớ.

### Con trỏ là gì?

Câu trả lời, con trỏ là một biến nguyên bình thường, nó chứa một giá trị trong đó. Vậy giá trị nó lưu trong đó là gì? Câu trả lời là địa chỉ của một ô nhớ. Ta lập trình trên **ARM 32 bit**, thì con trỏ chứa giá trị 32 bit (8 số hex).  
Ủa, vậy tại sao lại có**char\***, **int\***, **void\***, **double\***, vv? Câu trả lời là: con trỏ chứa địa chỉ, con trỏ kiểu char trỏ tới "vùng nhớ" kiểu char (nghĩa là các ô nhớ ở đây chứa dữ liệu kiểu **char**), con trỏ kiểu **int**thì trỏ tới vùng nhớ kiểu **int**....

### Chúng ta đi sâu hơn về bộ nhớ (memory)

**[](https://4.bp.blogspot.com/-z4h-TktExp4/WkOtMnycLsI/AAAAAAAAA2E/I4QJhmEbUoQS4B-zEUPpIUjd1EK4MeipACLcBGAs/s1600/contro-pointer.png)**

Nhìn vào hình và ta sẽ giải thích như sau:  
1. Khi chúng ta khai báo một biến, máy tính sẽ tạo ra một vùng nhớ để lưu giá trị cho biến đó (hay còn gọi là cấp phát bộ nhớ).  
2. Nếu biến được khai báo trong hàm **main()**, thì những ô nhớ được cấp phát sẽ nằm trong vùng **stack** (như trong hình là biên x)  
3. Nếu biến được khai báo ngoài hàm **main()** và ngoài tất cả các hàm, thì những ô nhớ được cấp phát sẽ nằm trong vùng **global** của bộ nhớ (trong hình là biến y).  
Giả sử máy tính cấp phát cho biến x một "vị trí" (vị trí này chính là địa chỉ đấy - địa chỉ của vùng nhớ, tại sao mình không nói là ô nhớ, bởi vì tùy vào CPU mà kiểu int có thể được chứa ở nhiều ô nhớ) là 4,100,000. Chúng ta đã gán 4 vào x thì cái "vị trí" 4,100,000 sẽ lưu giá trị 4.  
Và...con trỏ là cái thứ lưu cái số 4,100,000 như ví dụ ở trên đấy! Nếu muốn trỏ tới thằng x, thì phải dùng con trỏ kiểu int, bởi vì giá trị của x là integer.

### Khai báo con trỏ

|  |  |
| --- | --- |
|  | Kiểu dữ liệu \*tên con trỏ; |

Kiểu dữ liệu ở đây bao gồm:  
- Kiểu dữ liệu có sẵn: int, char, void, long, ....  
- Kiểu dữ liệu do chúng ta tự định nghĩa: struct, union.  
- Kiểu dữ liệu là lớp (class) do chúng ta định nghĩa (c++).  
- Kiểu dữ liệu dẫn xuất, kiểu con trỏ hàm (nâng cao).  
Ví dụ:

|  |  |
| --- | --- |
|  | int \*p; |

- p ở đây là con trỏ, và con trỏ p này trỏ đến vùng nhớ kiểu int. Lưu ý \*p không phải là con trỏ mà là giá trị của vùng nhớ mà con trỏ p trỏ tới (giả sử p trỏ tới thằng x trong hình đầu thì \*p = 4 nhé!).

Tiếp ví dụ:

|  |  |
| --- | --- |
|  | int \*a, \*b; // thì a và b đều là con trỏ |
|  | int \*a, b; // a là con trỏ, b là biến integer |
|  | int\* a, b; // a là con trỏ, b là biến nguyên, khai báo này đúng, nhưng dễ gây nhầm lẫn |
|  | void \*a; // đúng! Có con trỏ kiểu void này nhé |

### Khởi tạo giá trị cho biến con trỏ

Khởi tạo (initialize) khác với khai báo (declared) các bạn nhé! Sau khi khai báo 1 biến, bạn gán giá trị lần đầu tiên cho biến đó thì gọi là khởi tạo! Mình khuyên các bạn nên khởi tạo biến mỗi khi khai báo, vì khi không khởi tạo, nhiều trình biên dịch sẽ tự khởi tạo một giá trị rác cho biến đó, hoặc nhiều khi gây lỗi.  
Khởi tạo cho con trỏ:

|  |  |
| --- | --- |
|  | tên con trỏ = địa chỉ vùng nhớ; |

- Toán tử lấy địa chỉ: toán tử & 1 ngôi (unary operator), toán tử này hoàn toàn khác với toán tử & 2 ngôi (bitwise).  
ví dụ:

|  |  |
| --- | --- |
|  | int bienInteger = 100; // khai báo và khởi tạo biến integer |
|  | int \*conTro; // khai báo con trỏ kiểu int |
|  | conTro = &bienInteger; // con trỏ conTro trỏ tới biến bienInteger |

- Lúc này **\*conTro** tương đương với **bienInteger**, mọi thao tác với **\*conTro** cũng chính là thao tác với **bienInteger**, hay mọi thao tác với **bienInteger** cũng chính là thao tác với **\*conTro**.  
- Câu lệnh **bienInteger = 5**; hoàn toàn tương đương với **\*conTro = 5;**.  
- **bienInteger++;** hoàn toàn tương đương với **(\*conTro)++;** (Khác với **\*conTro++** nhé!).

Lấy một ví dụ đơn giản thế này:

|  |  |
| --- | --- |
|  | int \*p; // kiểu dữ liệu của con trỏ p này là int\* |

Hiểu kiểu dữ liệu của con trỏ chưa nào? quá dễ hiểu! Vậy xem ví dụ sau:

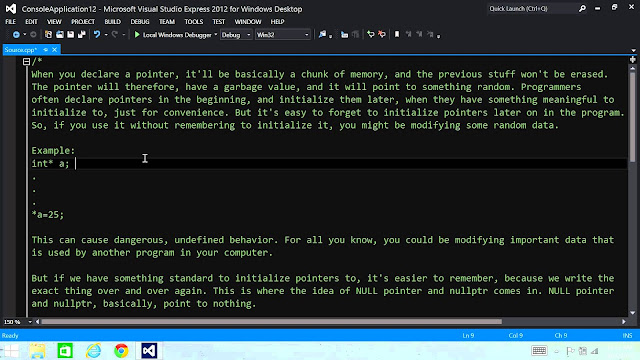
|  |  |
| --- | --- |
|  | void \*p; // thằng p này kiểu gì??? |

### **Các phép toán trên con trỏ**

1. Phép gán  
- Tất cả con trỏ đều có phép gán.  
- Như phần khởi tạo, phép gán yêu cầu vế trái là một con trỏ, và vế phải là một địa chỉ.  
- Phép gán yêu cầu sự tương xứng về kiểu dữ liệu, nếu không chúng ta phải ép kiểu, các bạn biết ép kiểu không?  
- Phép gán với con trỏ kiểu void không cần phải tương xứng kiểu dữ liệu.  
Xem ví dụ:

|  |  |
| --- | --- |
|  | int a = 100; |
|  | int \*p; |
|  | p = (int\*)0x0060FF08; // 0x0060FF08 giả sử là địa chỉ của biến a, |
|  | // đây là ép kiểu, |
|  | // vì 0x0060FF08 chỉ là một số nguyên, |
|  | // phải ép sang kiểu int\* mới gán được cho con trỏ. |
|  | printf("\*p: %d\n", \*p); // kết quả của \*p là 100, vì lúc này p trỏ tới a |

2. Phép so sánh  
- Phép so sánh ngang bằng dùng để kiểm tra 2 con trỏ có trỏ vào cùng 1 vùng nhớ hay không, hoặc kiểm tra 1 con trỏ có phải là đang trỏ vào NULL hay không (trong trường hợp cấp phát động, mở file, mở resource, vv).

**[](https://1.bp.blogspot.com/-on_ii1LXM8M/Wkpc9pR_2II/AAAAAAAAA3o/hZeV7Jz8Clkh7DUm3lrEJqi25AOmySStwCLcBGAs/s1600/pointer.jpg)**

- Phép so sánh lớn hơn nhỏ hơn: >, =, <= sử dụng để kiểm tra về độ thấp cao giữa 2 địa chỉ. Con trỏ nào nhỏ hơn thì trỏ vào địa chỉ thấp hơn.  
- Được quyền so sánh mọi con trỏ với 0, vì 0 chính là NULL.

|  |  |
| --- | --- |
|  | int a=197,\*p=&a; |
|  | double \*x; |
|  | p == &a; // kết quả = 1 |
|  | main == 0; // kết quả bằng 0 |
|  | p == 0; // kết quả bằng 0 |
|  | x == 0; // kết quả bằng 0 |

Lưu ý: chúng ta sẽ đi sâu về phép so sáng này sau, vì nâng cao nó khá khó!  
3. Phép cộng trừ và phép tăng giảm  
Bản chất của việc tăng giảm con trỏ là di chuyển con trỏ đi lên hoặc đi xuống:  
- Đương nhiên không phải di chuyển sang ô nhớ kế tiếp (byte kế tiếp), mà phụ thuộc vào kiểu dữ liệu của vùng nhớ con trỏ trỏ tới.  
Ví dụ:

|  |  |
| --- | --- |
|  | int a = 100, \*p = &a; // giả sử lúc này p đang trỏ tới địa chỉ 0x0060FF08 (địa chỉ của a) |
|  | p++; // p lúc này sẽ trỏ tới địa chỉ 0x0060FF0C (tăng lên 4 ô nhớ) |
|  | // bởi vì kiểu int lúc này có 4 byte |
|  | p -= 2; // p sẽ trỏ tới địa chỉ 0x0060FF04 |

Chú ý:  
- Không có phép tăng giảm con trỏ void và con trỏ hàm.  
- Không có phép cộng 2 con trỏ với nhau.  
- Phép trừ 2 con trỏ trả về độ lệch pha giữa 2 con trỏ

### Một số ví dụ hay và đương nhiên...khó!

Chương trình viết hoa một chuỗi được nhập vào từ bàn phím

|  |  |
| --- | --- |
|  | #include <stdio.h> |
|  | #include <conio.h> |
|  | #include <ctype.h> |
|  | int main() |
|  | { |
|  | char xau[100]; |
|  | char \*p = &xau; |
|  | printf("Nhap xau: "); |
|  | scanf("%[a-zA-Z]",xau); // lệnh này giới hạn kí tự nhập từ a-z và A-Z thôi |
|  | /\*while-loop style\*/ |
|  | while(\*p) |
|  | { |
|  | printf("%c",toupper(\*p)); |
|  | p++; |
|  | } |
|  | return 0; |
|  | } |

Chương trình in ra chuỗi bị đảo ngược

|  |  |
| --- | --- |
|  | #include <stdio.h> |
|  | #include <conio.h> |
|  | #include <ctype.h> |
|  | int main() |
|  | { |
|  | char xau[100]; |
|  | char \*p; |
|  | printf("Nhap xau: "); |
|  | scanf("%[a-zA-Z]",xau); |
|  |  |
|  | p = xau + strlen(xau) -1; // đưa con trỏ về cuối xâu |
|  | /\* for-loop style \*/ |
|  | for(; p >= xau; p--) |
|  | printf("%c", \*p); |
|  | return 0; |
|  | } |

Chương trình con lấy độ dài một chuỗi (khi học nhúng thì bạn nên viết chương trình con ra như vậy thay vì dùng hàm có sẵn, bạn phải khai báo thư viện, tốn bộ nhớ

|  |  |
| --- | --- |
|  | int strlen(char \*p) |
|  | { |
|  | int temp = 0; // biến đếm độ dài |
|  | while(\*p) // chừng nào gặp kí tự kết thúc chuỗi thì dừng (mã ASCII = 0) |
|  | { |
|  | temp++; |
|  | p++; |
|  | } |
|  | return temp; |
|  | } |

Con trỏ hàm và cách sử dụng con trỏ hàm.

Từ khóa volatile để làm gì ?

Trong lập trình nhúng (Embedded System), ta rất thường hay gặp khai báo biến với từ khóa volatile. Việc khai báo biến **volatile** là rất cần thiết để tránh những lỗi sai khó phát hiện do tính năng optimization của compiler.

**Cú pháp:**

**volatile** **int** **var**;

**int** **volatile** **var**;

Volatile đại diện cho các biến có thể thay đổi bất thường mà không thông qua nguồn source code.

**Một biến cần được khai báo dưới dạng biến volatile khi nào ?**

Khi mà giá trị của nó có thể thay đổi một cách không báo trước. Trong thực tế, có 3 loại biến mà giá trị có thể bị thay đổi như vậy :

* Memory-mapped peripheral registers (thanh ghi ngoại vi có ánh xạ đến ô nhớ)
* Biến toàn cục được truy xuất từ các tiến trình con xử lý ngắt (interrupt service routine)
* Biến toàn cục được truy xuất từ nhiều tác vụ trong một ứng dụng đa luồng.

**Ví dụ:**

Trong lập trình nhúng, chúng ta hay gặp đoạn code khi ta khai báo 1 biến đếm count, mỗi khi bấm nút xảy ra ngắt ngoài, chúng ta tăng biến đếm count. Tuy nhiên, khi chúng ta bật tính năng tối ưu code của compiler, nó sẽ hiểu rằng các biến như vậy dường như không thay đổi giá trị bởi phần mềm nên compiler có xu hướng loại bỏ biến count để có thể tối ưu kích cỡ file code chạy được sinh ra.

Từ khóa static để làm gì ?

# **Biến static trong khai báo biến cục bộ**

Khi 1 biến cục bộ được khai báo với từ khóa static. Biến sẽ chỉ được khởi tạo 1 lần duy nhất và tồn tại suốt thời gian chạy chương trình. Giá trị của nó không bị mất đi ngay cả khi kết thúc hàm. Tuy nhiên khác với biến toàn cục có thể gọi trong tất cả mọi nơi trong chương trình, thì biến cục bộ static chỉ có thể được gọi trong nội bộ hàm khởi tạo ra nó. Mỗi lần hàm được gọi, giá trị của biến chính bằng giá trị tại lần gần nhất hàm được gọi.

Ví dụ:

#include<stdio.h>

int in\_so\_thu\_tu(void)

{

   static int x = 0;

   x = x + 1;

   printf("%d\r\n",x);

}

int main() {

   in\_so\_thu\_tu ();         //giá trị của x tăng lên 1 đơn vị từ 0

   in\_so\_thu\_tu ();         //giá trị của x tăng lên 1 đơn vị từ 1

   in\_so\_thu\_tu ();         //giá trị của x tăng lên 1 đơn vị từ 2

   in\_so\_thu\_tu ();         //giá trị của x tăng lên 1 đơn vị từ 3

   in\_so\_thu\_tu ();         //giá trị của x tăng lên 1 đơn vị từ 4

   return 0;

}

Kết quả:

1

2

3

4

5

# **Biến static trong khai báo biến toàn cục và khai báo hàm**

Mỗi project thường sẽ được viết trên nhiều File vì mục đích phân chia module cũng như là để dễ bảo trì. Do có nhiều File nên rất có thể ở các File sẽ có sự trùng lặp trong cách đặt tên biến. Để tránh sự cố sai sót này người ta đưa ra khái niệm biến toàn cục tĩnh và hàm tĩnh.

* Biến toàn cục tĩnh sẽ chỉ có thể được truy cập và sử dụng trong File khai báo nó, các File khác không có cách nào truy cập được.
* Hàm tĩnh sẽ chỉ có thể gọi trong File khai báo nó, các File khác không có cách nào gọi hàm này được.

Ví dụ:

//-----------------

//A.c

// biến a này chỉ được sử dụng trong file A.c

**static** **int** a;

// hàm hienthi() này chỉ được sử dụng trong file A.c

**static** **void** hien\_thi() {};

**int** c;

//------------------

//B.c

// biến a này chỉ được sử dụng trong file B.c

**static** **int** a;

// hàm hienthi() này chỉ được sử dụng trong file B.c

**static** **void** hien\_thi() {};

**int** d;

Memory layout của một chương trình C.

# . Memory Layout

Memory layout của của một chương trình C/C++ gồm 5 phần chính: **Text Segment, Initialized Data Segment, Uninitialized Data Segment, Heap và Stack**

### 2.1 Text Segment

Text Segment ở vùng nhớ của địa chỉ thấp nhất, đây là phần chứa các đoạn mã lệnh của chương trình.

### 2.2 Initialized Data Segment (DS)

Initialized Data Segment ( Data Segment ) là nơi lưu trữ **global variables, static variables** với điều kiện các biến này được khởi tạo bởi programmer với **giá trị khác 0**.

int global = 100;

int foo() {

static int number = 10;

return 0;

}

Trong đoạn chương chính trên, biến global được khởi tạo với giá trị 100, và biến **static** number được khởi với giá trị 10 bởi programmer nên được lưu trữ vào Initialized Data Segment.

### 2.3 Uninitialized Data Segment (BSS)

Uninitialized Data Segment (BSS) là nơi lưu trữ **global variables, static variables** **không được khởi tạo** hoặc khởi tạo với **giá trị bằng 0.**

int global;

int foo() {

static int number = 0;

return 0;

}

Trong đoạn chương trình trên, biến global **không được khởi tạo giá trị** mặc định và biến **static** number **được khởi tạo với giá trị bằng 0** sẽ được lưu trữ vào Uninitialized Data Segment.

### 2.4 Heap (Dynamic Memory Allocation)

Trong C/C++ chúng ta có thể hoàn toàn control được quá trình cấp phát hoặc giải phóng bộ nhớ bằng các lệnh như malloc, calloc, relloc, free, new, delete, ... Vùng nhớ được cấp phát chính là HEAP, vùng nhớ này sẽ phình lên ( grows upward ) mỗi khi bạn cấp phát. Khi sử dụng xong các bạn phải delete vùng nhớ này. Nếu quên không delete sẽ gây ra hiện tượng Memory Leak.

struct MyClass {

int data[100];

MyClass() {std::cout << "constructed [" << this << "]\n";}

};

int main () {

std::cout << "1: ";

MyClass \* p1 = new MyClass; //cấp chát bộ nhớ trong HEAP

return 0;

}

### 2.5 Stack (Automatic Variable Storage)

Khác với HEAP, Stack là một vùng nhớ được cấp phát tự động và có cấu trúc LIFO (Last In First Out). Mỗi khi chương trình được gọi, thì các function frame sẽ được gọi và push vào trong stack. Function Frame có cấu trúc như ví dụ dưới đây.Khi main() được gọi, function frame của main() sẽ được push vào stack, và khi function foo() được gọi thì function frame của foo() sẽ được push vào. Cấu trúc của một function frame gồm bốn phần chính: Function Parameter, Return Address, Saved Previus Frame Pointer, Local Variable.

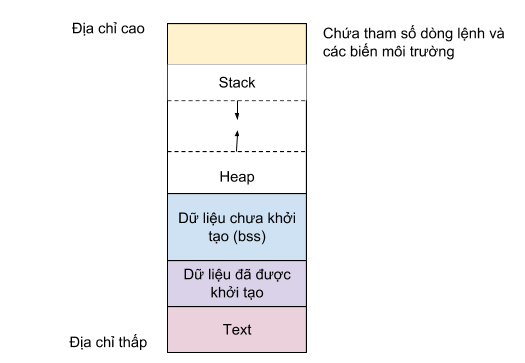
1. **Funtion Parameter** là các tham số truyền vào.
2. **Return Address** trong ví dụ trên, Return Address trả về địa chỉ p trong hàm main() chính là Local Variable.
3. **Saved Previus Frame Pointer** trỏ vào vị trí đầu tiên của function foo() tức là kết thúc của function main()
4. **Local Variable** là các biến local của function

* Lưu ý: Nếu chúng ta sử dụng hết vùng nhớ của Stack thì sẽ có một lỗi rất kinh điển đó là Stack OverFlow xảy ra.

## Sơ đồ bộ nhớ trong các chương trình C

Hiểu được sơ đồ, kiến trúc các vùng nhớ của một chương trình sẽ giúp bạn viết code một các hiệu quả và tối ưu. Mỗi vùng nhớ trong một chương trình lại có tính chất khác nhau. Về cơ bản trong một chương trình C, bộ nhớ được biểu hiện bởi các vùng sau:

* Vùng Text
* Vùng dữ liệu đã được khởi tạo
* Vùng dữ liệu chưa được khởi tạo
* Vùng Stack
* Vùng Heap

****

**Chương trình cơ sở**

Xét đoạn code sau:

#include <stdio.h>

int main()

{

    return 0;

}

Chương trình không chứa bất kỳ một biến nào, nên ta sẽ sử dụng nó để so sánh sự thay đổi kích thước trong vùng nhớ bằng lệnh size(1). Lệnh size(1) cho ta biết kích thước (byte) của các vùng text, DS, BSS.

Kiểm tra kích thước các vùng nhớ bằng cách lưu đoạn code trên vào 1 file source code "test.c" và làm như sau:

|  |
| --- |
| $ gcc test.c -o memory-layout-no-var  $ size memory-layout-no-var     text              data               bss   dec         hex    filename     1115            552               8     1675        68b    memory-layout-no-var |

Chú ý: Trong phạm vi bài viết này chúng tôi chỉ hướng dẫn cách kiểm tra kích thước các vùng nhớ trên hệ điều hành Linux (ubuntu, centos ...). Nếu các bạn đang dùng windown có thể dùng [linux online terminal](https://www.tutorialspoint.com/unix_terminal_online.php)[[1]](https://vimentor.com/en/lesson/so-do-bo-nho-trong-cac-chuong-trinh-c-1" \l "_msocom_1)

**Vùng Text**

Các tên khác:  text segment hay code segment.

Vùng này chứa mã máy của chương trình đã được biên dịch.

Vùng text segment của một đối tượng file có thể thực thi được thường là vùng nhớ chỉ đọc, điều này là để tránh việc chương trình vô tình bị sửa đổi không như mong muốn.

Thông thường vùng này là vùng nhớ chia sẻ nên ta chỉ cần một phiên bản sao chép tồn tại  trong bộ nhớ cho những chương trình thực thi thường xuyên.

**Vùng dữ liệu đã được khởi tạo**

Các tên khác: Initialized data segment, data segment (DS)

Để cho ngắn gon ta sẽ sử dụng "DS" như tên của vùng này.

DS là một phần trong bộ nhớ ảo của chương trình, nơi chứa những biến toàn cục, biến tĩnh hoặc các biến hằng đã được khởi tạo trong code.

Trong ví dụ sau các biến a và b sẽ được chứa trong vùng nhớ này.

#include <stdio.h>

int a = 9; /\* Khởi tạo giá trị cho biến toàn cục a \*/

static int b = 2; /\* Khởi tạo giá trị cho biến toàn cục b \*/

const int c = 8;  /\* Biến hằng c được chứa trong vùng nhớ khởi tạo chỉ đọc \*/

int main()

{

    b = a + 1;

    printf("%d %d %d\n", a, b, c);

    return 0;

}

Chú ý rằng DS không phải là vùng nhớ chỉ đọc, bởi vì giá trị của các biến có thể được thay đổi khi chạy runtime.

Thực ra vùng nhớ này có thể chia ra làm hai vùng nhỏ hơn, vùng khởi tạo chỉ đọc dùng để chứa các biến hằng, và vùng khởi tạo đọc-ghi để chứa các biến toàn cục và biến tĩnh thông thường.

Ta lưu đoạn code trên vào file "ds.c", thực hiện biên dịch bằng gcc và kiểm tra kích thước của chương trình trong các vùng nhớ bằng lệnh size để thấy được sự thay đổi giá trị của các vùng nhớ so với chương trình cơ sở ở trên.

|  |
| --- |
| $ gcc ds.c -o memory-layout-ds  $ size memory-layout-ds     text              data             bss    dec              hex          filename     1250            568             8       1826            722         memory-layout-ds |

Nhận thấy chỉ có kích thước của vùng text (do lượng code logic tăng lên) và data thay đổi so với chương trình cơ sở. điều này chứng tỏ các biến a, b, c ở trên đã được lưu vào vùng dữ liệu đã được khởi tạo.

**Vùng dữ liệu chưa được khởi tạo**

Các tên khác: Uninitialized Data Segment hay vùng BSS

Dữ liệu trong vùng này được khởi tạo bởi nhân của hệ điều hành về giá trị 0 trước khi chương trình được thực sự thực thi.

Vùng này chứa các biến toàn cục và các biến tĩnh được khởi tạo với các giá trị 0 (zero) hoặc không được khởi tạo với bất kỳ giá trị nào trong source code.

Trong ví dụ sau các biến global và i sẽ được chứa trong vùng nhớ bss.

#include <stdio.h>

int global; /\* Biến toàn cục chưa khởi tạo được chứa trong bss\*/

int main(void)

{

    static int i; /\* Biến tĩnh chưa khởi tạo được chứa trong bss \*/

    return 0;

}

Lưu chương trình vào file bss.c và thực hiện như sau để kiểm tra sự thay đổi kích thước cách vùng nhớ so với chương trình cơ sở.

|  |
| --- |
| $ gcc bss.c -o memory-layout-bss  $ size memory-layout-bss     text              data              bss   dec       hex     filename     1115            552              16    1683     693     memory-layout-bss |

Trong trường hợp này chỉ có vùng bss thay đổi kích thước so với chương trình cơ sở, vậy nên kết luận các biến chưa được khởi tạo "global" và "i" được lưu trong vùng bss là chính xác.

**Vùng nhớ Stack**

Vùng nhớ stack thường nằm liền kề với vùng heap. Hai vùng nhớ này phát triển theo chiều đối nhau. Ví trí đỉnh của stack được đánh dấu bằng stack pointer, vị trí đỉnh của heap được đánh đấu bằng heap pointer. Khi hai con trỏ này gặp nhau cũng là lúc bộ nhớ dành cho chúng bị cạn kiệt.

Tuy nhiên bạn không cần quá lo lắng về vấn đề ở trên vì ngày nay phần cứng của bộ nhớ (RAM vật lý) đã được tăng lên, hơn thế nữa người ta cũng áp dụng những kỹ thuật đánh địa chỉ ảo (bộ nhớ ảo), làm cho hai vùng này lớn hơn rất nhiều.

Vùng nhớ stack tuân theo cấu trúc LIFO (Last In First Out)và có một thanh ghi stack pointer để lưu giữ vị trí đỉnh của nó.

Các loại dữ liệu được chứa trong vùng nhớ này bao gồm:

* Biến cục bộ, biến tạm.
* Địa chỉ trả về sau mỗi lời gọi hàm.
* Các tham số truyền vào các hàm.

Các thuật toán đệ quy được thực hiện dựa trên hoạt động của stack và chúng được áp dụng rất nhiều trong thực tế.

Lỗi có thể gặp với vùng nhớ này chính là "stack overflow" khi mà ta sử dụng đệ quy không hồi kết hoặc đệ quy quá sâu. Nó cũng có thể xảy ra khi ta cấp phát một lượng bộ nhớ quá lớn, vượt quá giới hạn của stack.

Các biến tạo trong vùng nhớ này có thể tự động được thu hồi, nên ta không cần các kỹ thuật thu hồi bộ nhớ như khi ta cấp phát động trong heap.

**Vùng nhớ Heap**

Ngoài các đặc tính đã được đề cập ở trên vùng nhớ heap còn có một số đặc tính sau:

Các biến chứa trong vùng nhớ này phải được thu hồi thủ công bằng cách sử dụng các hàm delete, delete [] và free trong C/C++.

Việc cấp phát bộ nhớ cho một biến trong vùng này là châm hơn so với việc cấp phát trong stack. Có thể giải thích lý do này là vì vùng nhớ trong stack là luôn luôn liên tục nên việc cấp phát dễ dàng hơn nhiều.

Lỗi phân mảnh bộ nhớ hay xảy ra với bộ nhớ này qua thời gian sử dụng lâu, vì ta đã thực hiện nhiều thao tác cấp phát và thu hồi bộ nhớ trong nó, để lại nhiều vùng nhớ trống vô cùng nhỏ, khó xử dụng được.

Trong C/C++ có thể cấp phát trong vùng nhớ này qua các hàm new, malloc, calloc … Việc cấp phát có thể không thành công do lượng cấp phát vượt quá giới hạn của heap.

Vùng nhớ này thường được chỉa sẻ bởi các thư viện chia sẻ (shared libraries) hoặc các mô đun được tải động trong các tiến trình.

Hiểu về cách thức hoạt động, tính chất của các vùng nhớ là vô cùng quan trong với các lập trình viên viết các ngôn ngữ bậc thấp như C/C++, Assembly.

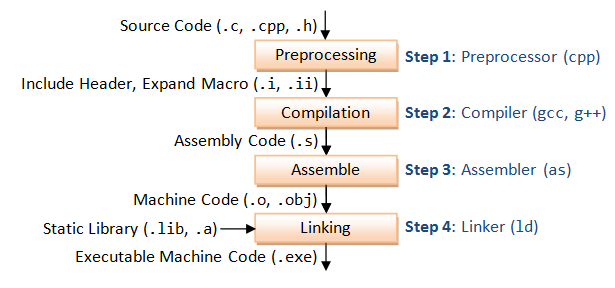
Tuy nhiên nó cũng có ích với bất cứ lập trình viên nào để giúp họ cải thiện được tốc độ, năng lực của chương trình, giúp tiết kiệm tài nguyên phần cứng và chương trình chạy hiệu quả hơn. Bạn có thể tìm hiểu thêm về các cấu trúc dữ liệu stack và heap để hiểu sâu hơn về cách hoạt động của hai vùng nhớ quan trọng này

Process để build một chương trình C như thế nào ?

**I. ĐỊNH NGHĨA**

Quy trình dịch là quá trình chuyển đổi từ ngôn ngữ bậc cao (NNBC) (C/C++, Pascal, Java, C#…) sang ngôn ngữ đích (ngôn ngữ máy) để máy tính có thể hiểu và thực thi. Ngôn ngữ lập trình C là một ngôn ngữ dạng biên dịch. Chương trình được viết bằng C muốn chạy được trên máy tính phải trải qua một quá trình biên dịch để chuyển đổi từ dạng mã nguồn sang chương trình dạng mã thực thi. Quá trình được chia ra làm 4 giai đoạn chính:

* Giai đoàn tiền xử lý (Pre-processor)
* Giai đoạn dịch NNBC sang Asembly (Compiler)
* Giai đoạn dịch asembly sang ngôn ngữ máy (Asember)
* Giai đoạn liên kết (Linker)



**II. HOẠT ĐỘNG**

1. Giai đoạn tiền xử lý – Preprocessor  
Giai đoạn này sẽ thực hiện:

* Nhận mã nguồn
* Xóa bỏ tất cả chú thích, comments của chương trình
* Chỉ thị tiền xử lý (bắt đầu bằng #) cũng được xử lý

*Ví dụ*: chỉ thị #include cho phép ghép thêm mã chương trình của một tệp tiêu để vào mã nguồn cần dịch. Các hằng số được định nghĩa bằng #define sẽ được thay thế bằng giá trị cụ thể tại mỗi nơi sử dụng trong chương trình.

2. Cộng đoạn dịch Ngôn Ngữ Bậc Cao sang Assembly

* Phân tích cú pháp (syntax) của mã nguồn NNBC
* Chuyển chúng sang dạng mã Assembly là một ngôn ngữ bậc thấp (hợp ngữ) gần với tập lệnh của bộ vi xử lý.

3. Công đoạn dịch Assembly

* Dich chương trình => Sang mã máy 0 và 1
* Một tệp mã máy (.obj) sinh ra trong hệ thống sau đó.

4. Giai đoạn Linker

* Trong giai đoạn này mã máy của một chương trình dịch từ nhiều nguồn (file .c hoặc file thư viện .lib) được liên kết lại với nhau để tạo thành chương trình đích duy nhất
* Mã máy của các hàm thư viện gọi trong chương trình cũng được đưa vào chương trình cuối trong giai đoạn này.
* Chính vì vậy mà các lỗi liên quan đến việc gọi hàm hay sử dụng biến tổng thể mà không tồn tại sẽ bị phát hiện. Kể cả lỗi viết chương trình chính không có hàm main() cũng được phát hiện trong liên kết.

**Kết thúc quá trình tất cả các đối tượng được liên kết lại với nhau thành một chương trình có thể thực thi được (executable hay .exe) thống nhất.**

Các bạn có thể tham khảo hướng dẫn liên kết 2 source file chương trình C bằng Command line qua video dưới đây.

Nếu một chương trình C sử dụng một thư viện libabc thì bước add thư viện vào chương trình C sẽ là bước nào trong các bước build chương trình C trên.

## Nhắc lại về kiến thức cũ

Để build 1 chương trình, sử dụng cú pháp như sau:

gcc -Wall -o <file thực thi> <các file nguồn>

* -Wall: chỉ định gcc xuất ra tất cả các warning trong quá trình build.
* -o: chỉ định gcc build source file thành file thực thi (excutable).
* <file thực thi>: là tên file output sau quá trình build và link các thư viện.
* <các file nguồn>: các file nguồn đầu vào (.c/.cpp) để build.

Hoặc

gcc -c <các source cần build>

Để build source file thành các file object (chưa build và link ra file thực thi).

gcc -Wall -o main main.c

Hoặc

gcc -c main.c

gcc -Wall -o main main.o

Và để chạy chương trình sử dụng lệnh.

./main

## Các bước để build 1 chương trình C/C++

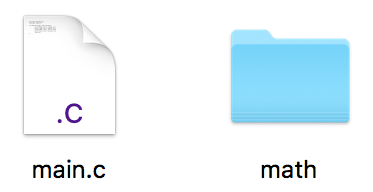


Quá trình build 1 chương trình trải qua 4 bước như dưới đây:

* Bước 1: Preprocessing - bộ tiền xử lý thay thế giá trị của các directive: #define, #include, #if, ... trong file mã nguồn .h và .cpp.
* Bước 2: Compilation - trình biên dịch chuyển mã nguồn thành mã assembly.
* Bước 3: Assemble - chuyển từ mã assembly sang mã nhị phân (mã máy - machine code).
* Bước 4: Linking - tiến hành liên kết các module và các thư viện liên kết thành file thực thi.

## Build module C/C++ với GCC

Giả sử trong Project có 1 module **math**, chức năng tính toán cộng trừ nhân chia.



Trong thư mục **math** có 2 file là **math.h** định nghĩa prototype của các hàm và **math.c** hiện thực các prototype trong **math.h**.

**File math.h**

#ifndef \_\_MATH\_H\_\_

#define \_\_MATH\_H\_\_

int add(int a, int b);

int sub(int a, int b);

int mul(int a, int b);

int div(int a, int b);

#endif

**File math.c**

#include "math.h"

int add(int a, int b)

{

return a + b;

}

int sub(int a, int b)

{

return a - b;

}

int mul(int a, int b)

{

return a \* b;

}

int div(int a, int b)

{

return a / b;

}

**File main.c**

#include <stdio.h>

#include "math/math.h"

int main(int argc, char\* argv[])

{

printf("Add: %d\n", add(5, 6));

printf("Sub: %d\n", sub(10, 6));

printf("Mul: %d\n", mul(5, 40));

printf("Div: %d\n", div(60, 6));

return 0;

}

Trong file main.c sử dụng module math, vì vậy trước chi build **main.c** phải build **math.c** sau đó link **math.o** (được build từ math.c).

* Bước 1: build module math
  + gcc -c math/math.c
  + Sau khi build sẽ được file **math.o**
* Bước 2: link **math.o** và **main.c**
  + gcc -Wall -o main math.o main.c
  + Sau khi chạy xong sẽ tạo được file thực thi **main**.

Thực thi file main bằng lệnh:

./main

Kết quả:

Add: 11

Sub: 4

Mul: 200

Div: 10

Đây là cách build module C/C++ sử dụng GCC, từ cách làm này có thể build và link các thư viện ngoài ví dụ như FFmpeg, OpenCV, FreeType, SDL, ...

## Link thư viện với GCC

### **Sơ lược thư viện liên kết tĩnh - Static Library**

Trong Linux thư viện liên kết tĩnh có phần mở rộng là **.a**, khi link thư viện liên kết tĩnh với mã để tạo ra file thực thi thì GCC sẽ nạp tất cả mã của thư viện liên kết tĩnh vào file thực thi, vì vậy kích thước của file thực thi lúc này sẽ tăng lên.

### **Sơ lược thư viện liên kết động - Dynamic Library**

Trong Linux thư viện liên kết động có phần mở rộng là **.so**, khác với thư viện liên kết tĩnh, khi link thư viện liên kết động mã thực thi sẽ không link vào file thực thi. Khi thực thi chương trình mới tìm các thư viện liên kết động và nạp vào bộ nhớ. Vì vậy mà file thực thi sau khi link xong sẽ không gia tăng kích thước từ thư viện.

Cả 2 thư viện này đều chứa mã nhị phân và có những file header **.h** chứa các prototype của các hàm trong thư viện.

Đa số các thư viện lớn như FFmpeg, OpenCV, SDL sau khi build xong sẽ tạo thư viện liên kết động và cả thư viện liên kết tĩnh.

Dưới đây là 1 đoạn mã sử dụng OpenCV

#include <iostream>

#include <opencv2/core/core.hpp>

#include <opencv2/highgui/highgui.hpp>

using namespace std;

using namespace cv;

int main(int argc, char\* argr[])

{

cout << Mat::zeros(4, 4, CV\_8UC1);

return 0;

}

Vậy làm sao build được chương trình này?

1. Cần chỉ định cho gcc đường dẫn chứa các file header của thư viện sử dụng **-I**
2. Cần chỉ định cho gcc đường dẫn chứa các file library của thư viện sử dụng **-L**
3. Cần link các module thư viện liên kết.

Ví dụ thư viện OpenCV sẽ có:

* /usr/local/include: đường dẫn chứa header của thư viện.
* /usr/local/lib: đường dẫn chứa lib của thư viện.

Và các module của OpenCV:

ImageMagick libcairo-gobject.a libopencv\_calib3d.dylib libpcre32.dylib

cairo libcairo-gobject.dylib libopencv\_core.3.1.0.dylib libpcrecpp.o.dylib

cmake libcairo-script-interpreter.2.dylib libopencv\_core.3.1.dylib libpcrecpp.a

dtrace libcairo-script-interpreter.a libopencv\_core.dylib libpcrecpp.dylib

gio libcairo-script-interpreter.dylib libopencv\_features2d.3.1.0.dylib libpcreposix.o.dylib

girepository-1.0 libcairo.2.dylib libopencv\_features2d.3.1.dylib libpcreposix.a

glib-2.0 libcairo.a libopencv\_features2d.dylib libpcreposix.dylib

gobject-introspection libcairo.dylib libopencv\_flann.3.1.0.dylib libpixman-1.0.dylib

itc14.0.0 libfontconfig.1.dylib libopencv\_flann.3.1.dylib libpixman-1.a

libMagick++-6.916.6.dylib libfontconfig.a libopencv\_flann.dylib libpixman-1.dylib

libMagick++-6.Q16.a libfontconfig.dylib libopencv\_highgui.3.1.0.dylib libpng.a

libMagick++-6.916.dylib libfreetype.6.dylib libopencv\_highgui.3.1.dylib libpng.dylib

libMagick++-6.916.la libfreetype.a libopencv\_highgui.dylib libpng16.16.dylib

libMagickCore-6.916.2.dylib libfreetype.dylib libopencv\_imgcodecs.3.1.0.dylib libpng16.a

libMagickCore-6.916.a libfreetype.la libopencv\_imgcodecs.3.1.dylib libpng16.dylib

libMagickCore-6.Q16.dylib libfribidi.0.dylib libopencv\_imgcodecs.dylib libpostproc.54.0.100.dylib

libMagickCore-6.916.la libfribidi.dylib libopencv\_imgproc.3.1.0.dylib libpostproc.54.dylib

libMagickWand-6.916.2.dylib libgio-2.0.0.dylib libopencv\_imgproc.3.1.dylib libpostproc.a

libMagickWand-6.Q16.a libgio-2.0.a libopencv\_imgproc.dylib libpostproc.dylib

libMagickWand-6.916.dylib libgio-2.0.dylib libopencv\_m1.3.1.0.dylib libswresample.2.1.100.dylib

libMagickWand-6.Q16.la libgirepository-1.0.1.dylib libopencv\_m1.3.1.dylib libswresample.2.dylib

libSDL2-2.0.0.dylib libgirepository-1.0.a libopencv\_ml.dylib libswresample.a

libSDL2.a libgirepository-1.0.dylib libopencv\_objdetect.3.1.0.dylib libswresample.dylib

libSDL2.dylib libglib-2.0.0.dylib libopencv\_objdetect.3.1.dylib libswscale.4.1.100.dylib

libSDL2.la libglib-2.0.a libopencv\_objdetect.dylib libswscale.4.dylib

libSDL2\_test.a libglib-2.0.dylib libopencv\_photo.3.1.0.dylib libswscale.a

libSDL2main.a libgmodule-2.0.0.dylib libopencv\_photo.3.1.dylib libswscale.dylib

libass.5.dylib libgmodule-2.0.a libopencv\_photo.dylib libtc18.6.dylib

libass.a libgmodule-2.0.dylib libopencv\_shape.3.1.0.dylib libtclstub8.6.a

libass.dylib libgobject-2.0.0.dylib libopencv\_shape.3.1.dylib libtiff.5.dylib

libavcodec.57.48.101.dylib libgobject-2.0.a libopencv\_shape.dylib libtiff.a

libavcodec.57.dylib libgobject-2.0.dylib libopencv\_stitching.3.1.0.dylib libtiff.dylib

libavcodec.a libgthread-2.0.0.dylib libopencv\_stitching.3.1.dylib libtiffxx.5.dylib

libavcodec.dylib libgthread-2.0.a libopencv\_stitching.dylib libtiffxx.a

libavdevice.57.0.101.dylib libgthread-2.0.dylib libopencv\_superres.3.1.0.dylib libtiffxx.dylib

libavdevice.57.dylib libharfbuzz-gobject.o.dylib libopencv\_superres.3.1.dylib libtk8.6.dylib

libavdevice.a libharfbuzz-gobject.a libopencv\_superres.dylib libtkstub8.6.a

libavdevice.dylib libharfbuzz-gobject.dylib libopencv\_ts.a libxvidcore.4.dylib

libavfilter.6.47.100.dylib libharfbuzz-icu.o.dylib libopencv\_video.3.1.0.dylib libxvidcore.a

libavfilter.6.dylib libharfbuzz-icu.a libopencv\_video.3.1.dylib node\_modules

libavfilter.a libharfbuzz-icu.dylib libopencv\_video.dylib pkgconfig

libavfilter.dylib libharfbuzz.o.dylib libopencv\_videoio.3.1.0.dylib python2.7

libavformat.57.41.100.dylib libharfbuzz.a libopencv\_videoio.3.1.dylib sqlite3.7.15.1

libavformat.57.dylib libharfbuzz.dylib libopencv\_videoio.dylib tcl8

Gõ lệnh để build như sau:

g++ -Wall -o main main.cpp -I/usr/local/include -L/usr/local/lib -lopencv\_calib3d -lopencv\_core -lopencv\_features2d -lopencv\_flann -lopencv\_highgui -lopencv\_imgcodecs -lopencv\_imgproc -lopencv\_ml -lopencv\_objdetect -lopencv\_photo -lopencv\_shape -lopencv\_stitching -lopencv\_superres -lopencv\_ts -lopencv\_video -lopencv\_videoio -lopencv\_videostab

Vì OpenCV được viết bằng C++ nên sử dụng **g++** để build thay vì dùng **gcc**:

Sau khi gõ dòng lệnh ./main để thực thi chương trình, kết quả như sau:

[ 0, 0, 0, 0;

0, 0, 0, 0;

0, 0, 0, 0;

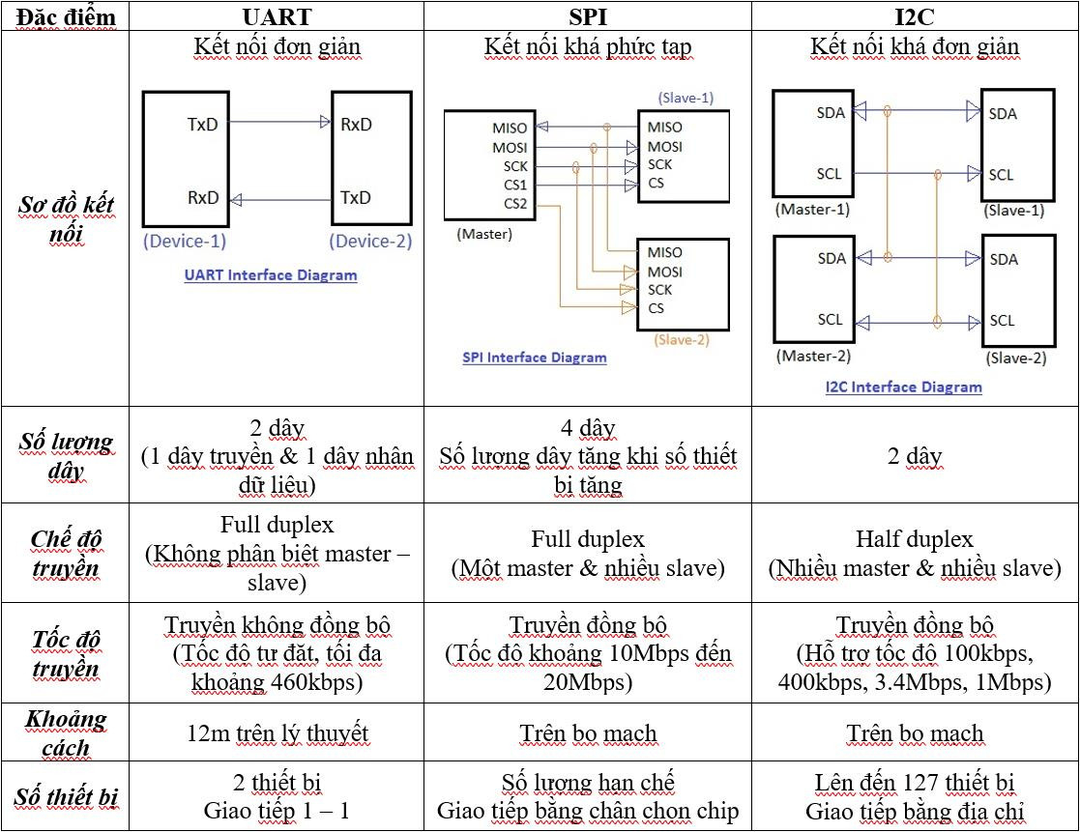
0, 0, 0, 0 ]

1 nhược điểm rất lớn là cần gõ trên Terminal khá nhiều nếu project có rất nhiều file **.c**. Để khắc phục nhược điểm này có thể viết makefile.

Nếu sử dụng IDE để thao tác thì việc link các thư viện này hoàn toàn tương tự, có thể xem clip link thư viện OpenCV trong Xcode:

#### Embedded MCU

I2C và SPI gồm có bao nhiêu dây giao tiếp ?



Số địa chỉ tối đa trong một mạng I2C là bao nhiêu ?

Khác biệt giữa I2C và SPI như thế nào ?

Khác biệt giữa UART và I2C như thế nào ?

Hàm blocking và non-blocking là như thế nào ?

Blocking và Non-Blocking trong lập trình chủ yếu được đề cập khi muốn miêu tả về cách một chương trình thực hiện các dòng lệnh của nó. Chúng ta có thể hiểu một cách đơn giản, nếu chương trình được thực hiện theo mô hình Blocking có nghĩa là các dòng lệnh được thực hiện một cách tuần tự. Khi một dòng lệnh ở phía trước chưa được hoàn thành thì các dòng lệnh phía sau sẽ chưa được thực hiện và phải đợi khi mà thao tác phía trước hoàn tất, và nếu như các dòng lệnh trước là các thao tác cần nhiều thời gian xử lý như liên quan đến IO (input/output) hay mạng (networking) thì bản thân nó sẽ trở thành vật cản trở ( blocker ) cho các lệnh xử lý phía sau mặc dù theo logic thì có những việc ở phía sau ta có thể xử lý được luôn mà không cần phải đợi vì chúng không có liên quan gì đến nhau.

Mô hình blocking tồn tại từ lịch sử, khi mà máy tính chỉ có thể xử lý đơn nhiệm trên một lõi (core) của bộ vi xử lý (chip). Nhưng theo thời gian, công nghệ ngày một trưởng thành với những thành tựu về phần cứng, máy tính giờ có thể làm nhiều việc cùng một lúc thì người ta cần phải suy nghĩ đến việc làm sao tận dụng được tối đa tài nguyên xử lý của máy tính và tránh lãng phí nó. Từ đó mà bất cứ chỗ nào có phần xử lý Blocking không cần thiết, người ta cần thay vào một giải pháp xử dụng tài nguyên khôn ngoan hơn, đó là Non-Blocking.

Trong mô hình Non-Blocking, các dòng lệnh không nhất thiết phải lúc nào cũng phải thực hiện một cách tuần tự (sequential) và đồng bộ (synchronous) với nhau. Ở mô hình này nếu như về mặt logic dòng lệnh phía sau không phụ thuộc vào kết quả của dòng lệnh phía trước, thì nó cũng có thể hoàn toàn được thực hiện ngay sau khi dòng lệnh phía trước được gọi mà không cần đợi cho tới khi kết quả được sinh ra. Những dòng lệnh phía trước miêu tả ở trên còn có thể gọi là được thực hiện theo cách không đồng bộ (Asynchronous), và đi theo mỗi dòng lệnh thường có một callback (lời gọi lại) là đoạn mã sẽ được thực hiện ngay sau khi có kết quả trả về từ dòng lệnh không đồng bộ. Để thực hiện mô hình Non-Blocking, người ta có những cách để thực hiện khác nhau, nhưng về cơ bản vẫn dựa vào việc dùng nhiều Thread (luồng) khác nhau trong cùng một Process (tiến trình), hay thậm chí nhiều Process khác nhau (inter-process communication – IPC) để thực hiện. Và mẫu thiết kết (design pattern) tên là event-loop là một trong những mẫu thiết kế nổi tiếng để thực hiện cơ chế Non-Blocking mà nếu có điều kiện trong tương lai mình sẽ viết bài để giới thiệu cho các bạn.

Trong bài viết này mình sẽ đưa ra một ví dụ để các bạn hiểu rõ hơn về Blocking và Non-Blocking, bao gồm hình ảnh minh họa cũng như chương trình đơn giản viết bằng Java. Ví dụ này mô tả quá trình lấy dữ liệu từ 3 lời gọi hàm khác nhau và sau đó in kết quả khi trả về từ hàm ra màn hình. Lời gọi hàm trong ví dụ chỉ là một đoạn code đơn giản mô phỏng một việc làm trong một thời gian nhất định, trong thực tế việc này có thể thao tác disk IO như đọc dữ liệu từ file hay database, hoặc thao tác liên quan đến kết nối mạng như gọi webservice … 3 hàm ở trên mình giả sử sẽ là 3 việc thực tế không liên quan gì đến nhau, và mình sẽ chỉ ra cùng là làm 3 việc thì cơ chế Blocking sẽ khác với Non-Blocking như thế nào.

Trước tiên các bạn hãy nhìn vào hình ảnh minh họa về Blocking và Non-Blocking ở dưới đây

Phần phía trên miêu tả sự hoạt động theo cơ chế Blocking mà ở đây mặc dù không có sự liên đới giữa 3 việc, nhưng các công việc tiếp sau luôn phải chờ công việc phía trước thực sự xong rồi mới có thể bắt đầu thực hiện. Các bước sẽ được mô tả như dưới đây

1. Hàm sentayho.com.vn() được gọi để lấy dữ liệu, vì nó là Blocking nên trước khi công việc này hoàn thành các việc tiếp sau sẽ phải đợi
2. Hàm printData(d1) được gọi để in dữ liệu lấy về từ sentayho.com.vn(), tương tự nó cũng là Blocking
3. Hàm sentayho.com.vn() được gọi để lấy dữ liệu, mặc dùng là nó không liên quan gì đến hai dòng lệnh trên, nhưng đến tận bây giờ nó mới được thực hiện và là Blocking nên chiếm một khoảng thời gian xử lý nữa
4. Hàm printData(d2) được gọi để in dữ liệu lấy về từ sentayho.com.vn(), là Blocking
5. Hàm sentayho.com.vn() được gọi để lấy dữ liệu, là Blocking
6. Hàm printData(d3) được gọi để in dữ liệu lấy về sentayho.com.vn(), là Blocking

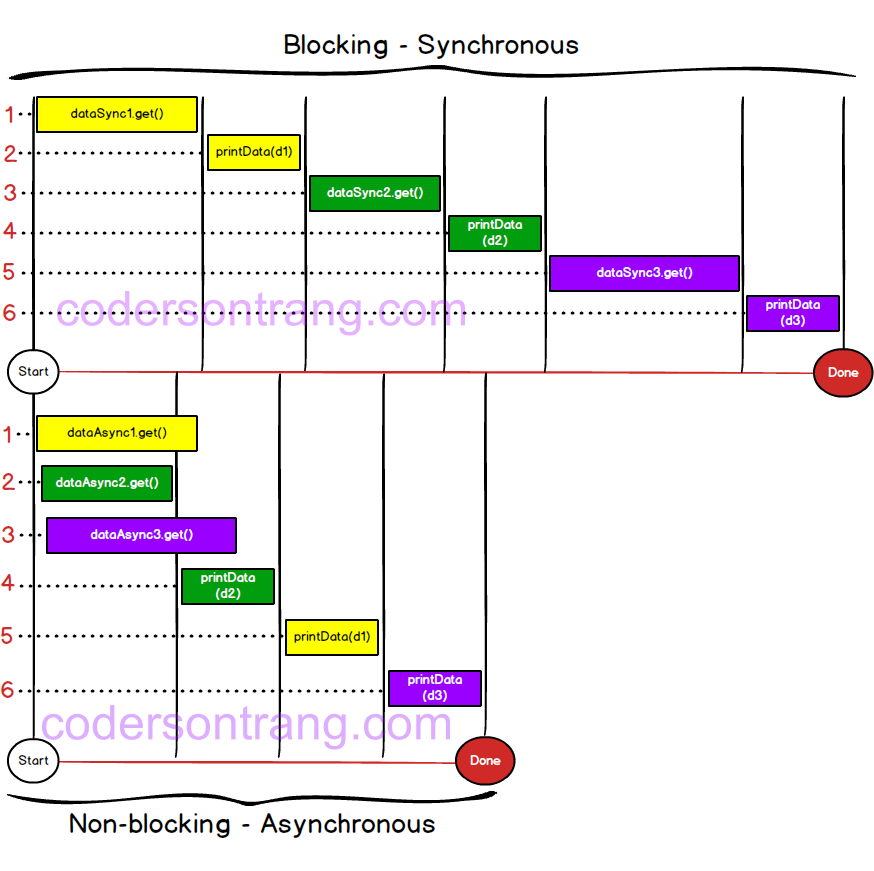
Ở phần này, mọi thao tác đều là blocking nên thời gian để thực hiện xong hết các thao tác sẽ bằng tổng thời gian của từng thao tác.

Phía dưới là phần thể hiện việc làm tất cả những việc trên, các thao tác in dự liệu printData(d1), printData(d2), printData(d3) vẫn là các thao tác Blocking nhưng ở đây có sự tham gia của Non-Blocking trong các thao tác lấy dữ liệu sentayho.com.vn(), sentayho.com.vn(), sentayho.com.vn(). Các thao tác Non-Blocking sẽ được bắt đầu gần như ngay lập tức và không cần phải chờ các thao tác phía trước thực hiện xong. Sau khi có kết quả các thao tác Non-Blocking sẽ gọi lại callback để in kết quả trả về ra màn hình. Cụ thể sẽ được diễn giải như ở dưới đây:

1. Hàm sentayho.com.vn() được gọi để lấy dự liệu, vì nó là Non-Blocking nên quá trình thực thi sẽ không phải dừng ở đây mà tiếp tục thực hiện dòng lệnh tiếp sau gần như ngay lập tức, tất nhiên vẫn phải sau khi đăng ký một callback để in ra dữ liệu trả về từ sentayho.com.vn().
2. Như nói ở trên, ngay sau đó, hàm sentayho.com.vn() được gọi cùng với đăng ký callback. Vì là Non-Blocking nên quá trình cũng giống như trên.
3. Tiếp theo hàm sentayho.com.vn() cũng được thực hiện tương tự. Đến đây, 3 hàm gọi để lấy dữ liệu gần như được thực hiện đồng thời mà không cần phải chờ nhau.
4. Trong khi hàm sentayho.com.vn() và sentayho.com.vn() đang thực hiện thì hàm sentayho.com.vn() đã lấy được dữ liệu về, lúc này callback được gọi để in dữ liệu đó ra màn hình, trong callback lúc này printData(d2) được gọi và nó là Blocking.
5. Trong thời gian printData(d2) đang thực hiện, sentayho.com.vn() đã hoàn tất việc lấy dữ liệu, callback của nó được gọi tuy nhiên vì printData(d2) là Blocking và đang thực hiện nên việc thực hiện printData(d1) sẽ phải chờ.
6. Cũng tương tự như trên, sentayho.com.vn() cũng hoàn tất việc lấy dữ liệu, callback của nó được gọi, lần này printData(d3) không những phải chờ printData(d2) như trên mà nó còn phải chờ thêm cả printData(d1) bởi vì printData(d1) cũng là Blocking. Sau khi cả printData(d2) và printData(d1) được hoàn thành thì printData(d3) được thực hiện và toàn bộ quá trình hoàn tất.

Bây giờ nhìn lại hình vẽ một lần nữa ta có thể thấy Non-Blocking rút ngắn thời gian thực hiện chương trình hơn là Blocking, việc rút ngắn thời gian này không phải vì các công việc được thực hiện nhanh hơn mà vì nhiều việc được thực hiện cùng một lúc hơn.

Sau đây là đoạn code demo cho việc thực thi với Blocking và Non-Blocking được viết bằng Java. Chúng ta sẽ tạo một Java project đơn giản trên IntelliJ IDE như hình dưới đây



**Mục lục**[[Ẩn](https://sentayho.com.vn/non-blocking-la-gi.html)]

* [1 DataSync.java](https://sentayho.com.vn/non-blocking-la-gi.html#DataSyncjava)
* [2 MainSync.java](https://sentayho.com.vn/non-blocking-la-gi.html#MainSyncjava)
* [3 DataAsync.java](https://sentayho.com.vn/non-blocking-la-gi.html#DataAsyncjava)
* [4 MainAsync.java](https://sentayho.com.vn/non-blocking-la-gi.html#MainAsyncjava)

### **DataSync.java**

public class DataSync { private int id; private long simulationDuration; DataSync(int id, long simulationDuration){ sentayho.com.vn = id; sentayho.com.vnlationDuration = simulationDuration; } public String get(){ try { sentayho.com.vnp(this.simulationDuration); } catch (InterruptedException e) { e.printStackTrace(); } return “data-“+id; } }

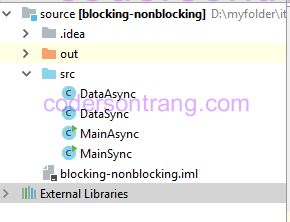
Lớp DataSync thể hiện nguồn dữ liệu có thể lấy về theo cơ chế Blocking, một nguồn dữ liệu được mô tả gồm có

* **id** : định danh nguồn dữ liệu, như ở trên miêu tả, chúng ta có ba nguồn được định danh là 1, 2, 3
* **simulationDuration**: tính bằng mili giây, giả lập quãng thời gian cần để lấy được dữ liệu về từ nguồn dữ liệu qua phương thức get().

### **MainSync.java**

public class MainSync { public static void main(String[] args) { long startTime, endTime; DataSync dataSync1 = new DataSync(1, 5000); //5s DataSync dataSync2 = new DataSync(2, 3000); //3s DataSync dataSync3 = new DataSync(3, 6000); //6s startTime = sentayho.com.vnentTimeMillis(); sentayho.com.vntln(“Start”); String d1 = sentayho.com.vn(); printData(d1); String d2 = sentayho.com.vn(); printData(d2); String d3 = sentayho.com.vn(); printData(d3); sentayho.com.vntln(“Done”); endTime = sentayho.com.vnentTimeMillis(); sentayho.com.vnt(“Execution time (ms): “+(endTime- startTime)); } private static void printData(String data){ try { sentayho.com.vnp(1000); } catch (InterruptedException e) { e.printStackTrace(); } sentayho.com.vntln(“Synchronously printing “+data); } }

sentayho.com.vn bao gồm phương thức main() là điểm bắt đầu của chương trình. Đầu tiên ba nguồn dữ liệu Blocking được khởi tạo lần lượt là dataSync1, dataSync2 và dataSync3 với ba giá trị thời gian khác nhau là 5 giây, 3 giây và 6 giây. Như vậy nguồn dữ liệu số 2 sẽ thực hiện nhanh nhất rồi đến số 1 và cuối cùng là số 3. Sau đó ở mỗi nguồn dữ liệu sẽ được gọi phương thức get() để lấy dự liệu về theo cơ chế Blocking. Dữ liệu sẽ được in ra ngay sau khi được trả về từ nguồn dữ liệu qua phương thức printData(). Phương thức printData() cũng là Blocking và ta mô phỏng thời gian để thực hiện công việc này trong quãng thời gian 1 giây. Ở phía cuối ta cũng in khoảng thời gian tính bằng mili giây để toàn bộ chương trình hoàn tất.

Và khi chạy chương trình ta thấy được thứ tự các câu lệnh được thực hiện giống như mô tả trên hình minh họa phần đầu cũng như tổng thời gian hoàn tất là 17001 mili giây như hình dưới đây:

### **DataAsync.java**

import sentayho.com.vnlier; public class DataAsync implements Supplier { private int id; private long simulationDuration; DataAsync(int id, long simulationDuration){ sentayho.com.vn = id; sentayho.com.vnlationDuration = simulationDuration; } @Override public String get() { try{ sentayho.com.vnp(simulationDuration); }catch (Exception e){} return “data-“+id; } }

Lớp DataAsync thể hiện nguồn dữ liệu có thể lấy về theo cơ chế Non-Blocking, và tương tự như lớp DataSync bên trên, ở đây cũng có id và simulationDuration.

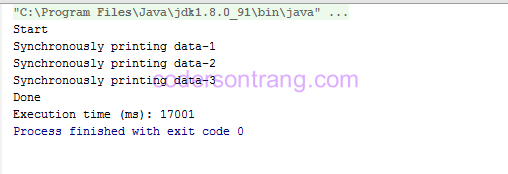
### **MainAsync.java**

import sentayho.com.vnurrent.\*; public class MainAsync { public static void main(String[] args) { long startTime, endTime; CountDownLatch latch = new CountDownLatch(3); DataAsync dataAsync1 = new DataAsync(1, 5000); DataAsync dataAsync2 = new DataAsync(2, 3000); DataAsync dataAsync3 = new DataAsync(3, 6000); startTime = sentayho.com.vnentTimeMillis(); sentayho.com.vntln(“Start”); try{ sentayho.com.vnlyAsync(dataAsync1).thenAccept(d1 -> { printData(d1); sentayho.com.vntDown(); }); sentayho.com.vnlyAsync(dataAsync2).thenAccept(d2 -> { printData(d2); sentayho.com.vntDown(); }); sentayho.com.vnlyAsync(dataAsync3).thenAccept(d3 -> { printData(d3); sentayho.com.vntDown(); }); sentayho.com.vnt(); sentayho.com.vntln(“Done”); endTime = sentayho.com.vnentTimeMillis(); sentayho.com.vnt(“Execution time (ms): “+(endTime- startTime)); }catch (Exception e){ } } private static void printData(String data){ try { sentayho.com.vnp(1000); } catch (InterruptedException e) { e.printStackTrace(); } sentayho.com.vntln(“Synchronously printing “+data); } }

Cũng giống như lớp MainSync ở trên, ta cũng khởi tạo 3 nguồn dữ liệu nhưng lần này là Non-Blocking, các định danh cũng thời gian giả lập để thực hiện lấy dữ liệu về không có gì thay đổi. Tiếp đó để thực hiện việc lấy dữ liệu theo cơ chế Non-Blocking, ta sử dụng CompletableFuture của Java 8 để nhận vào nguồn dữ liệu qua hàm supplyAsync(). supplyAsync() sẽ thực hiện hàm get() theo cách Non-Blocking từ tham số đầu vào là một sentayho.com.vnlier, chính vì vậy mà ta thấy vì sao lớp DataAsync ở trên phải implement sentayho.com.vnlier. Và đồng thời ta cũng có thể đăng ký callback cho mỗi lời gọi Non-Blocking này qua phương thức thenAccept() mà ở đây cụ thể là in giá trị trả về qua phương thức printData().

Ở đây chúng ta có một CountDownLatch được sử dụng, bởi vì các lời gọi hàm là Non-Blocking nên các lệnh phía sau sẽ thực hiện mà không cần chờ các lệnh phía trước nó hoàn thành. Chính vì vậy mà khi 3 nguồn dữ liệu chưa kịp trả về kết quả, thread thực hiện phương thức main() sẽ chạy hết chương trình trước mà không kịp in các dữ liệu trả về qua các callback. Đó chính là lý do vì sao chúng ta sử dụng CountDownLatch ở đây, mục đích chính là để chờ khi tất cả callback được hoàn tất thì ta mới kết thúc chương trình.

Khi tiến hành chạy chương trình, thứ tự các kết quả được in ra sẽ giống như mô tả ở hình vẽ tại phần đầu và tổng thời gian thực hiên chương trình lần này chỉ là 7171 mili giây thay vì 17001 mili giây khi thực hiện với cơ chế Blocking.



Ngày nay khi mà các thế hệ phần cứng ngày một trưởng thành với khả năng xử lý song song, thì việc các ứng dụng cần đáp ứng khả năng sử dụng tài nguyên một cách tối ưu là điều rất cần thiết. Non-Blocking là mô hình mà các ứng dụng sẽ luôn hướng đến mọi lúc có thể. Trong một số ngôn ngữ truyền thống như Java, mỗi một dòng lệnh đa phần sẽ là Blocking trong Thread gọi nó, các lập trình viên có thể tạo một cơ chế Non-Blocking trong chương trình của mình bằng việc chủ động sử dụng các API để tạo Thread khác, CompletableFuture… hoặc cao hơn là lập trình với giao thức **Reactive Stream (RxJava)**. Trong các nền tảng hiện đại ra đời sau như NodeJS, mọi dòng code đa phần sẽ đều là Non-Blocking, giúp cho các lập trình viên dễ dàng hơn rất nhiều trong việc sử dụng tối ưu tài nguyên, tránh lãng phí khi không cần thiết phải đợi chờ các thao tác xử lý đa phần là liên quan đến IO và Network, cũng như tránh các vấn đề phức tạp khi các lập trình viên phải tự mình tạo ra và quản lý các luồng xử lý không đồng bộ với nhau.

Cuối cùng hi vọng bài viết này sẽ giúp một phần nào đó cho các bạn có thể hình dung ra và phân biệt được mô hình lập trình Blocking, Non-Blocking và có được sự so sánh giữa chúng.

Good luck

Bạn có biết RTOS không ?

**RTOS là gì:**

RTOS (Real-Time operating system) hay được gọi là hệ điều hành thời gian thực mà cho phép ứng dụng của bạn chạy đa tác vụ và có thể đáp ứng được “deadline” theo thời gian thực. Lưu ý rằng việc đáp ứng được “deadline” không nhất nhiết có nghĩa là phải nhanh mà ở đây là mang tính “đúng thời điểm” và chính xác (cần là có ngay).

# **Khi nào cần sử dụng RTOS:**

·       Chạy các dự án lớn đòi hỏi xử lý nhiều công việc nhưng vẫn phải đáp ứng được về mặt thời gian

·       Các ứng dụng về viễn thông và IOT, các thiết bị liên quan đến y tế…

# **Tại Sao phải dùng RTOS:**

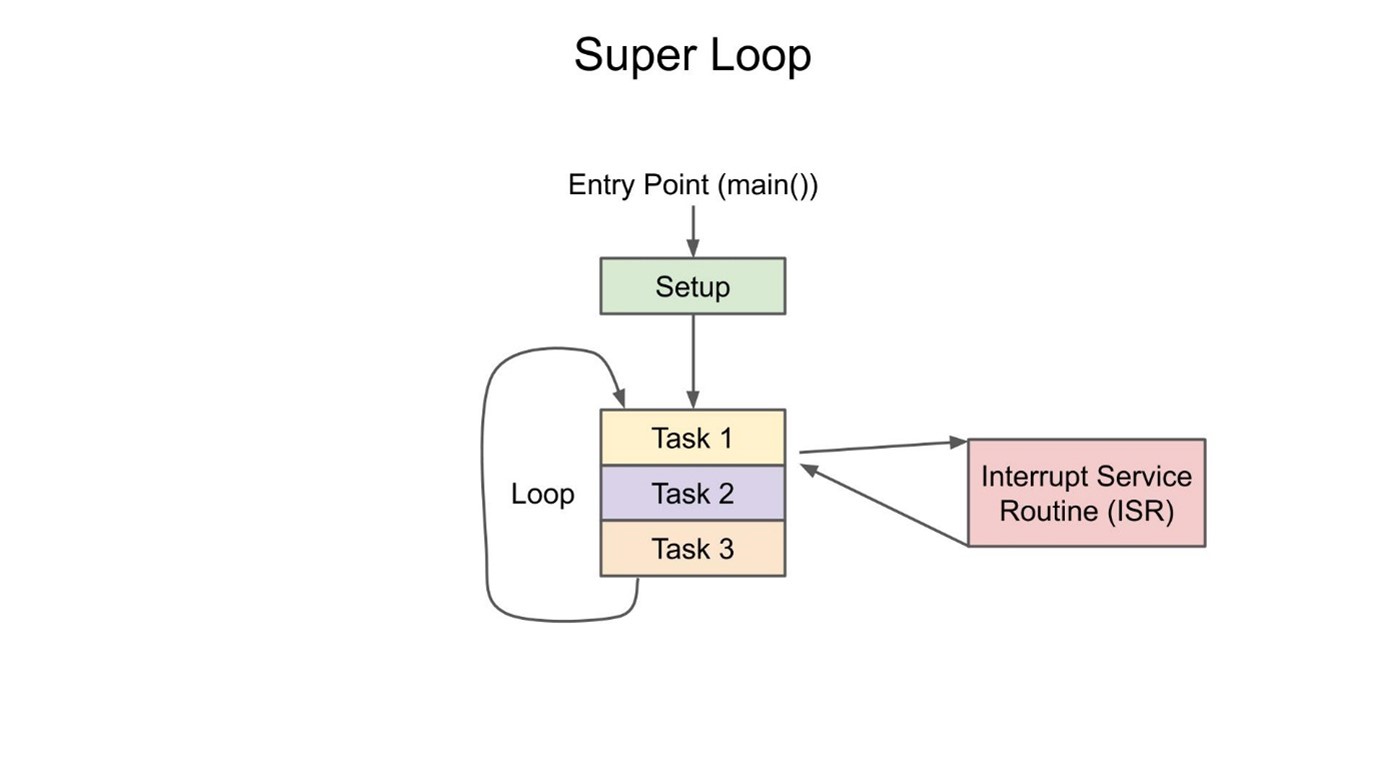
·       Giúp chương trình của bạn dễ quản lý và phát triển vì nó giúp phân “chia” 1 vấn đề phức tạp thành các phần nhỏ hơn “để trị”.

·       Tăng tính linh động và dễ bảo trì

·       Dễ dàng hơn trong việc chia sẻ tài nguyên của CPU.

# **So Sánh “Siêu Vòng Lặp” và đa tác vụ:**

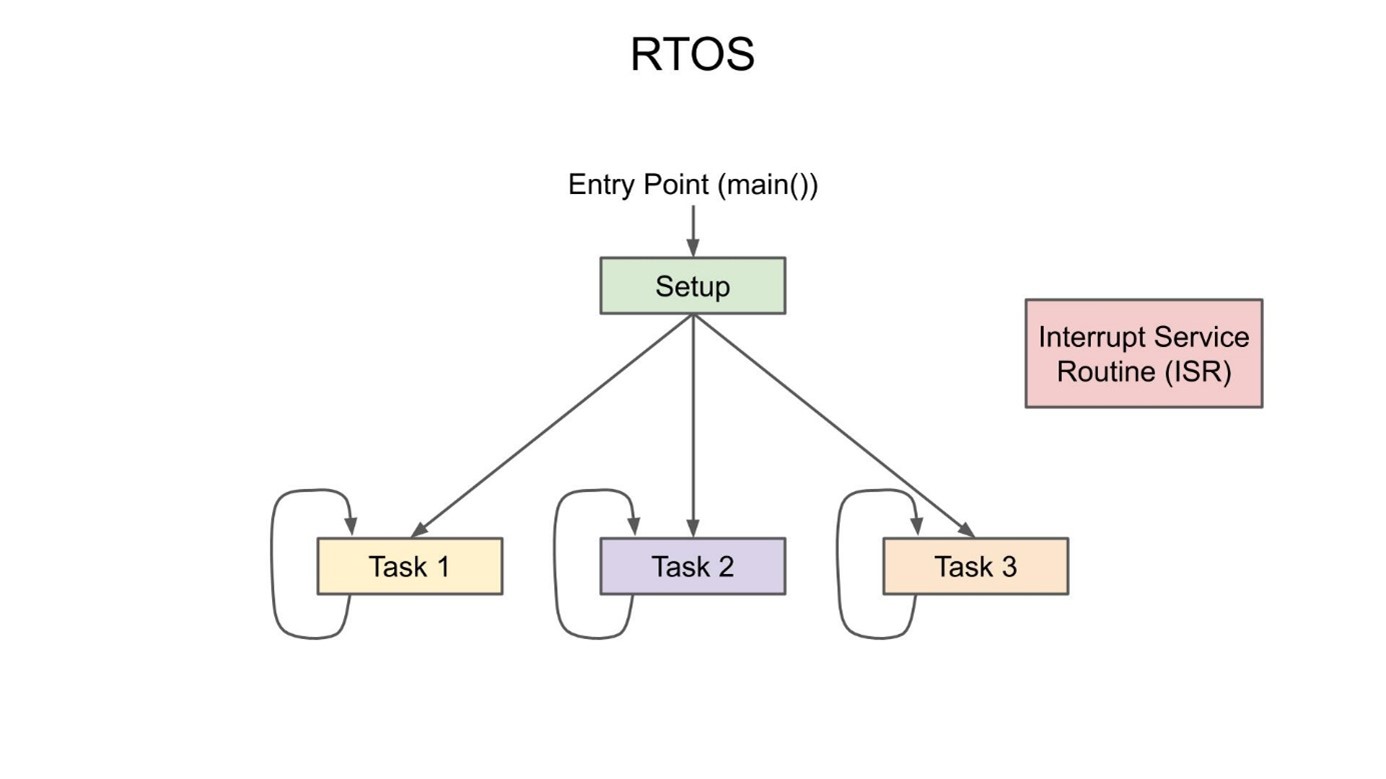
Nếu các bạn đã từng viết ứng dụng về nhúng thì chắc hẳn đã quá quen thuộc với kiến trúc của “siêu vòng lặp”. Ở đây, chúng không sử dụng hệ điều hành và cấu trúc khá đơn giản. Sau khi khai báo biến, thư viện, … thì trong hàm main() sau khi đã khởi tạo xong thì thì chúng sẽ thực hiện nhiều tác vụ định kỳ trong vòng lặp while(1) (void loop đối với arduino).



Và tất nhiên đây là cách phổ biến để xây dựng một chương trình, vì nó dễ thực hiễn và gỡ lỗi. Bạn thậm chỉ có thể thêm các ngắt để giúp cho chương trình được

“nuột” hơn.

Nhưng vấn đề chỉ thực sự xảy ra khi có quá nhiều nhiệm vụ cần phải hoàn thành trong 1 thời gian nhất định nhưng vẫn phải “xếp hàng” chờ cho nhiệm vụ trước hoàn thành xong. Đây là lúc mà RTOS có thể ra tay trợ giúp. Thay vì thực hiện mọi thứ tuần tự thì về cơ bản bạn có thể thực hiện mọi thứ gần như đồng thời.



Khi sử dụng RTOS thì task vụ vấn bị gián đoạn bởi trình phục vụ ngắt. Sau đó nó vẫn sẽ tiếp tục thực thi task vụ bị ngắt trước đó.

Ở trong FreeRtos thì người ta coi “task” là “thread” nhưng bạn cần phải phân biệt chúng 1 cách rạch ròi:

·       **Task**: tập hợp các lệnh được tải vào bộ nhớ. Có thể hiểu đơn giản đây là 1 số đơn vị công việc hoặc mục tiêu cần phải hoàn thành.

·       **Thread**: là 1 đơn vị của CPU với bộ đếm chương trình và bộ nhớ ngăn xếp của riêng nó.

Chốt lại: Có nên sử dụng RTOS không? Câu trả lời là có nếu bạn cần chạy nhiều tác vụ để hoàn thành mục tiêu có đòi hỏi coi về thời gian đáp ứng. Nếu không thì tốt nhất vẫn nên sử dụng cấu trúc “siêu vòng lặp”.

Đặc biệt việc sử dụng RTOS cũng vô cùng có ích trong làm việc nhóm khi ta có thể phân chia 1 vấn đề phức tạp thành các task nhỏ, và mỗi thành viên có thể quản lý 1 số task nhất định để dự án được phát triển đồng thời (nhưng có yêu cầu phải thống nhất về việc giao tiếp giữa các task).

Làm thế nào trao đổi dữ liệu giữa 2 thread trong RTOS ?

# Vấn đề đồng bộ / trao đổi dữ liệu giữa các task

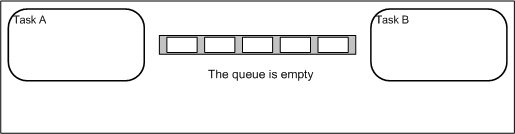
Mỗi task thực hiện tác vụ riêng của mình, chúng có không gian bộ nhớ riêng và hoàn toàn độc lập với nhau.

* Vậy làm thế nào để phối hợp chúng trong một chương trình lớn hơn, nếu xét mỗi task như là một module?
* Nếu task này muốn giao tiếp/trao đổi thông tin với task kia thì làm thế nào?
* Nếu 2 task cùng mức ưu tiên, cùng chạy song song và cùng muốn truy cập đến một tài nguyên chung thì làm sao để tránh xung đột?

Tất cả chỉ gói gọn trong hàng đợi (queue), semaphore và mutex

# Hàng đợi (queue)

Hoạt động theo cơ chế FIFO (First In - First Out) dùng trong FreeRTOS để trao đổi thông điệp (message) giữa các task với nhau.



## Ví dụ

Sử dụng lại ví dụ esp-rtos-basic-task, nhưng 2 task sửa lại theo mục đích sau:

* task\_led: nháy led, đếm biến counter, đếm xong thì gửi giá trị biến đếm này qua queue cho task\_printf in ra giá trị.
* task\_printf: chỉ chờ có thông điệp mới trên queue thì lấy thông điệp này ra khỏi queue và in giá trị này!

Mã nguồn chương trình:

git clone https://github.com/esp8266vn/esp-rtos-basic-queue-sem-mutex.git

cd esp-rtos-basic-queue-sem-mutex

Để sử dụng được queue phải #include "freertos/queue.h"

## Tạo queue

Cú pháp:

xQueueHandle xQueueCreate( UBaseType\_t uxQueueLength,

UBaseType\_t uxItemSize )*;*

* uxQueueLength: kích thước/số phần tử tối đa trên queue
* uxItemSize: kích thước của mỗi phần tử, tùy thuộc vào kiểu dữ liệu của phần tử trên queue.

Trường hợp này, biến counter là uint32\_t. Giả sử cần tối đa 10 phần tử trên queue. Khai báo và tạo như sau:

xQueueHandle xCounterQueue;

*// ...*

xCounterQueue = xQueueCreate( 10, **sizeof**(**uint32\_t**));

## Gửi thông điệp lên queue

Theo cú pháp:

BaseType\_t **xQueueSend**(

QueueHandle\_t xQueue,

**const** **void** \* pvItemToQueue,

TickType\_t xTicksToWait

);

* xQueue: queue handle (xCounterQueue)
* pvItemToQueue: trỏ đến phần tử/thông điệp cần gửi lên queue (&counter)
* xTicksToWait: thời gian chờ tối đa (timeout - tính bằng tick) trong trường hợp queue bị đầy không gửi thêm vào được.

task\_led sẽ làm nhiệm vụ này:

**void** **task\_led**(**void** \*pvParameters)

{

**uint32\_t** counter = 0;

**for**(;;){

vTaskDelay(100);

GPIO\_OUTPUT\_SET(LED\_GPIO, led\_state);

led\_state ^=1;

counter++;

**if**( xQueueSend( xCounterQueue,( **void** \* ) &counter, 10 ) != pdPASS )

{

*// Failed to send msg to queue, TODO handle*

}

}

}

## Nhận thông điệp từ queue

Theo cú pháp:

BaseType\_t **xQueueReceive**(

QueueHandle\_t xQueue,

**void** \*pvBuffer,

TickType\_t xTicksToWait

);

* xQueue: queue handle (xCounterQueue)
* pvBuffer: trỏ đến nơi sẽ chứa thông điệp lấy từ queue xuống.
* xTicksToWait: timeout trong trường hợp không có phần từ nào trên queue (đang trống)

task\_printf sẽ thực hiện nhiệm vụ này:

**void** **task\_printf**(**void** \*pvParameters)

{

**uint32\_t** receiveCounter;

printf("task\_printf started\n");

**for**(;;){

**if**( xQueueReceive( xCounterQueue, &( receiveCounter ), 100 ) )

{

printf("task\_printf, received counter = %u\n", receiveCounter);

}

}

}

## Chạy chương trình

make clean

make

make flash

#### **Kinh nghiệm C & Embedded**

Bạn biết về Link List, Ring Buffer, Queue không ?

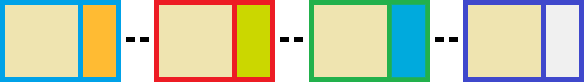
Danh sách liên kết là một trong những cấu trúc phổ biến và thường được sử dụng để quản lý dữ liệu trong chương trình. Các cấu trúc dữ liệu này rất phong phú để tùy vào nhu cầu mà lập trình viên sẽ ứng dụng như:

* Linked list - danh sách liên kết.
* Stack - ngăn xếp.
* Queue - hàng đợi.

## Linked list - danh sách liên kết

Trong các ứng dụng, [danh sách liên kết](https://www.iostream.vn/article/danh-sach-lien-ket-don-elmLe) là 1 cấu trúc dữ liệu linh hoạt (không bị cố định kích thước như mảng) và được sử dụng rất nhiều từ Snake (rắn săn mồi) cho đến danh sách các dòng dữ liệu khi làm việc với cơ sở dữ liệu.

Gần giống với mảng, danh sách liên kết là một chuỗi các phần tử chứa dữ liệu trong nó, nhưng khác với mảng, mỗi phần tử trong danh sách (còn được gọi là node) chứa trong nó dữ liệu cần lưu và cả dữ liệu để liên kết đến node sau nó.

Linked list - danh sách liên kết

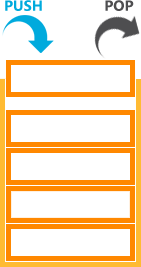
Với tính cơ động của mình, danh sách liên kết được sử dụng nhiều để tổ chức các dữ liệu mà người lập trình không biết trước kích thước. Ngoài ra danh sách liên kết còn được sử dụng để tổ chức dữ liệu một cách thứ tự, chẳng hạn như **Stack** và **Queue** là những cấu trúc dữ liệu có thứ tự có thể hiện thực được từ danh sách liên kết. Ngôn ngữ C++ cũng hiện thực 1 lớp dữ liệu để [xử lý danh sách liên kết](https://www.iostream.vn/article/stl-list-trong-c-s0lmL) hiệu quả và tiện lợi.

## Stack - ngăn xếp

Khi xếp 1 chồng đĩa trong bếp, thông thường đĩa trên cùng được đặt cuối cùng và khi lấy sẽ lấy từ đĩa trên cùng đó. Đó là một hình ảnh trong thực tế của **Stack**.

Khi lấy đĩa từ trên cùng, hành động đó được gọi là pop, ngược lại khi để đĩa lên trên cùng được gọi là push. Một điều cần phải lưu ý khi hiện thực hóa cấu trúc dữ liệu này là **Stack** chỉ có thể truy cập được phần tử trên cùng của nó (chỉ có thể lấy cái đĩa trên cùng của chồng đĩa). Ngoài ra, **Stack** có một đặc điểm rất hay đó là thứ tự của các phần tử được đưa vào **Stack** sẽ là **Vào trước – Ra sau**. Vì những đặc điểm này của **Stack** cho nên chúng có một số ứng dụng sau:

* Dùng để đảo ngược 1 chuỗi các ký tự.
* Dùng để hiện thực hóa tính năng undo của phần mềm.
* Dùng để quản lý những dữ liệu mà chúng ta chỉ cần quan tâm đến phần tử được push vào sau cùng của **Stack**(ví dụ như quản lý State Game, trạng thái của nhân vật trong trò chơi, …).

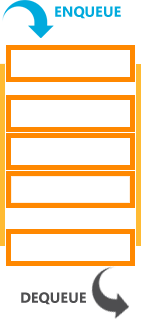
Stack - ngăn xếp

Đối với các ngôn ngữ lập trình hiện đại, Stack cũng được hỗ trợ sẵn vì nhu cầu cao của loại cấu trúc dữ liệu này, có thể thấy được C++ cũng hiện thực 1 bộ [STL trong đó có Stack](https://www.iostream.vn/article/stack-ngan-xep-uinHf) cho nhiều loại cấu trúc dữ liệu hữu ích.

## Queue - hàng đợi

Khi xếp hàng để mua hàng, người đến trước sẽ được thanh toán trước, người đến sau sẽ phải thanh toán sau. Đó là hình ảnh thực tế của **Queue**. Khi một người đi ra khỏi hàng đợi, được gọi là dequeue; khi một người bước vào xếp hàng, được gọi là enqueue. Khi quản lý dữ liệu bằng cấu trúc này, nó sẽ tạo ra sự thứ tự trong dữ liệu đưa vào, vì đặc điểm này mà **Queue** có một số ứng dụng sau:

* Cấu trúc hàng đợi thông điệp.
* Cấu trúc hàng đợi cho người chơi trong các trò chơi có tính năng tự tìm đối thủ.
* Cấu trúc các bộ đệm truyền tải dữ liệu.

Queue - hàng đợi

Ứng dụng Ring Buffer như thế nào ?

# **I. Ring buffer là gì? Nguyên lý hoạt động.**

Có thể hiểu 1 cách đơn giản Ring\_buffer là 1 bộ đệm hình tròn (hình nhẫn) dùng để chứa dữ liệu, nó cũng như 1 bộ đệm thẳng nhưng điểm đặc biệt của bộ đệm này là có thể chứa vô số dữ liệu nếu ta biết phối hợp việc đọc, ghi dữ liệu bộ đêm 1 cách nhịp nhàng với số ô nhớ giới hạn, còn bộ đệm thẳng thì chứa càng nhiều dữ liệu thì sẽ càng tốn bộ nhớ. Cùng tìm hiểu nguyên lý hoạt động của nó xem vì sao nó có thể làm được như vậy nhé

**Nguyên Lý hoạt động:**



bộ đệm này được thực hiện dựa trên một mảng. Kèm theo đó là 2 con trỏ Write và Read.

·       Mỗi khi nhận lệnh ghi, con trỏ pWrite sẽ ghi data vào bộ đệm, sau đó sẽ tăng lên 1 đơn vị pWrite++.

·       Mỗi khi nhận lệnh đọc, cỏn trỏ pRead sẽ tăng lên một. Sau đó đọc giá trị từ bộ đệm ra.

·       Khi 1 con trỏ tới được cuối mảng, nó sẽ cuộn lại vị trí đầu tiên. Đó là lý do vì sao gọi đây là bộ đệm vòng.

·       Nếu (pRead+1) == pWrite, chứng tỏ bộ đệm đang trống, không có gì để đọc.

·       Nếu pWrite == pRead, chứng tỏ bộ đệm đang đầy. không thể ghi thêm.

·       Các bạn cũng có thể thay đổi việc đọc/ghi buffer trước, hoặc thay đổi con trỏ trước cho phù hợp với ứng dụng của bạn. Khi đó thì cẩn kiểm tra lại các điều kiện bộ đệm trống và tràn.

Nó hoạt động trên nguyên lý FIFO (First in First Out). Việc sử dụng bộ đệm này khiến ta dễ dàng kiểm soát dữ liệu hơn khi sử dụng những giao thức truyền thông như UART

# 

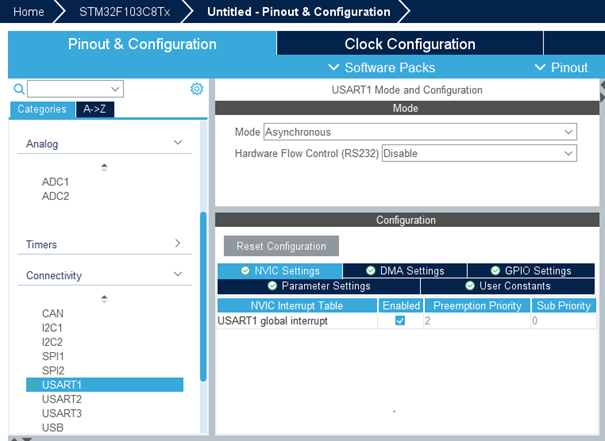
# **II. Lập trình Ring buffer trên STM32**

Ở ví dụ này, ta sẽ cùng nhau xử lý dữ liệu từ Ringbuffer theo ý muốn.

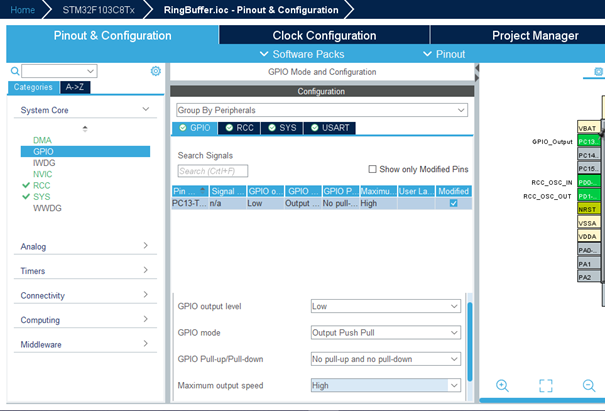
## 1. Khởi tạo Project trên CubeMx

Bước 1: Tạo mới Project, chọn MCU, cấu hình clock, debug, …

Bước 2: Kích hoạt UART và ngắt của nó, chọn mức ưu tiên chính = 2. Các thông số còn lại giữ nguyên.



Bước 3: Cấu hình GPIO PC13 để nháy led.



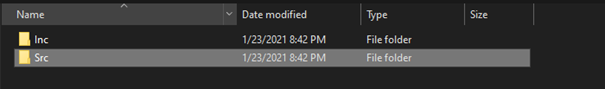
Bước 4: Đặt tên project và Gen code.

## ****2. Lập trình trên Keil C.****

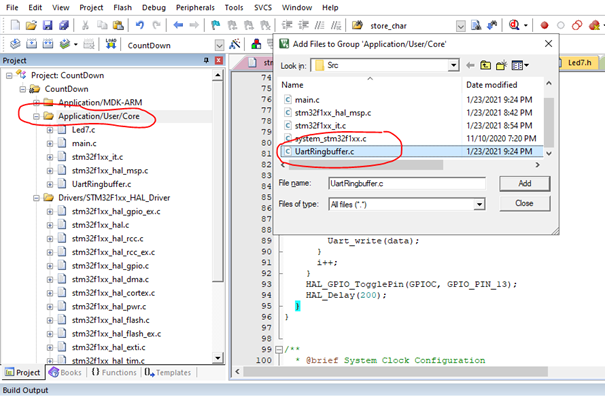
Bước 1: dowload thư viện Ring\_buffer theo link dưới đây.

https://github.com/stingdau1206/Ring\_Buffer.git

Bước 2: Copy file .c vào Src và file .h vào Inc trong thư mục chứa project



Bước 3: Nhấn đúp vào ../User/Core Add file .c của thư viện



Bước 4: vào file “stm32f1xx\_it.c” khai báo hàm ngắt, hàm này sẽ đẩy dữ liệu vào buffer khi có ngắt nhận dữ liệu.

**extern** **void** Uart\_isr **(UART\_HandleTypeDef \*huart)**;

Đồng thời tại hàm USART1\_IRQHandler(void) ta sẽ xử lý như sau.

void USART1\_IRQHandler(void)

{

 /\* **USER** CODE BEGIN USART1\_IRQn 0 \*/

           Uart\_isr(&huart1);

 /\* **USER** CODE END USART1\_IRQn 0 \*/

 //HAL\_UART\_IRQHandler(&huart1);

 /\* **USER** CODE BEGIN USART1\_IRQn 1 \*/

 /\* **USER** CODE END USART1\_IRQn 1 \*/

}

Bước 5: Include thư viện ring\_buffer trong file main.

#include "UartRingbuffer.h"

Bước 6: khởi tạo ring buffer.

Bước 7: Lập trình để Read và write dữ liệu từ buffer. Ở đây ta chỉ gửi đi dữ liệu có số thứ tự chẵn

int main(void)

{

 /\* USER CODE BEGIN 1 \*/

int i = 0;

 HAL\_Init();

 SystemClock\_Config();

 MX\_GPIO\_Init();

 MX\_DMA\_Init();

 MX\_USART1\_UART\_Init();

//HAL\_UART\_Receive\_DMA(&huart1, Rx\_data, 4);

 // khởi tạo ring buffer

 Ringbuf\_init ();

**while** (1)

 {

**if** (IsDataAvailable())

   {

     int data = Uart\_read();

**if** (i % 2 == 0)

                 {

                       Uart\_write(**data**);

                 }

                 i++;

   }

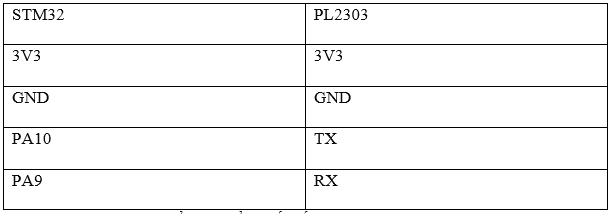
   HAL\_GPIO\_TogglePin(GPIOC, GPIO\_PIN\_13);

   HAL\_Delay(200);

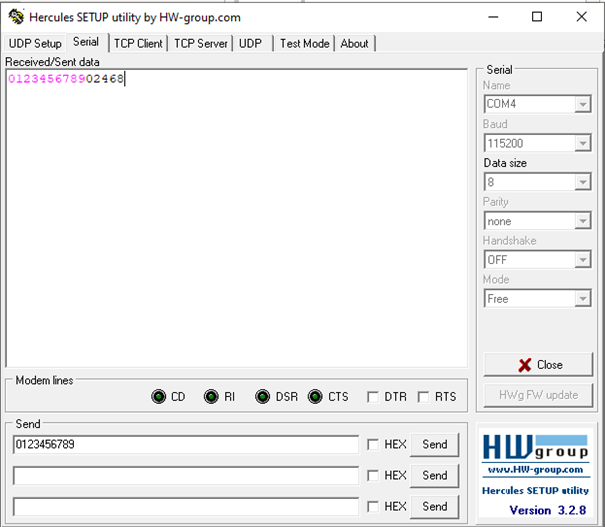
 }

}

Bước 8: Kết nối với module USB to COM theo sơ đồ dưới đây



Bước 9: nạp code và chạy mô phỏng trên hercules. Và đây là kết quả



﻿

Ta sẽ thấy sau khi nhận được dữ liệu, vđk chỉ gửi lại những dữ liệu có vị trí chẵn.

Tuy chỉ là 1 ví dụ đơn giản, nhưng ta có thể thấy được việc dữ liệu được lưu vào buffer giúp ta có thể dễ dàng bóc tách và xử lý, đồng thời nếu ta không xử lý ngay thì nó vẫn ở đó nếu khong bị tràn buffer, và ta có thể xử lý vào lúc nào tùy thích.

Lệnh nháy led Trong bài này chỉ là tạo hiệu ứng mỗi lần nháy là mỗi lần gửi dữ liệu chứ cũng không có vai trò gì quan trọng cả :v

## III. Giải thích hàm trong thư viện.

**void** Ringbuf\_init**(void)**;

Khởi tạo ring buffer

**int** Uart\_read**(void)**;

Đọc 1 byte từ buffer

Kết quả trả về là dữ liệu đọc được

**void** Uart\_write**(int c)**;

Ghi 1 byte vào buffer, tham số truyền vào là byte cần ghi

**void** Uart\_sendstring**(const char \*s)**;

Ghi 1 chuỗi vào buffer, tham số truyền vào là chuỗi cần ghi

**int** IsDataAvailable**(void)**;

Kiểm tra dữ liệu trong data có sẵn để đọc không, return true nếu có sẵn dữ liệu, ngược lạ thì false

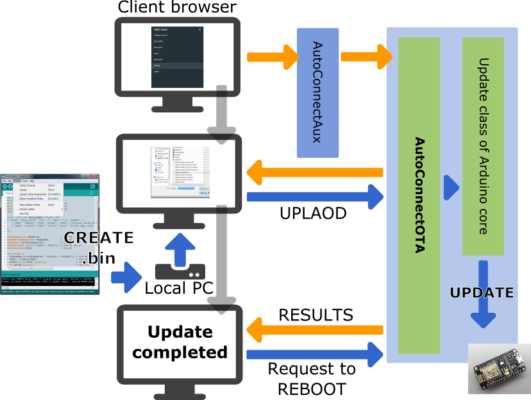
Trong khi sử dụng Ring Buffer mình cần lưu ý điều gì ?

Bạn biết về FOTA không ? Hãy trình bày FOTA mà bạn đã từng làm. Phần này hãy đảm bảo là bạn biết về FOTA, vì người pv sẽ hỏi sâu vào phần này. Dưới đây là những câu hỏi người phỏng vấn sẽ hỏi bạn.

## FOTA là gì?

FOTA (Firmware Over The Air) là thuật ngữ chỉ việc nạp firmware từ xa mà không cần sử dụng mạch nạp, chỉ cần có 1 kết nối không dây là có thể sử dụng được. FOTA được sử dụng rộng rãi trong các [sản phẩm IOT.](https://khuenguyencreator.com/lap-trinh-iot/)

Các bạn đã từng cập nhật hệ điều hành điện thoại hay chưa? Đó chính là FOTA đó. Hệ điều hành sẽ được cập nhật thông qua internet, điện thoại sau khi cập nhật xong sẽ restart và chạy hệ điều hành mới.



## Tại sao chúng ta phải sử dụng FOTA

FOTA đem lại rất nhiều lợi ích khi [lập trình IOT](https://khuenguyencreator.com/)

* Giúp việc nạp firmware trong quá trình R&D dễ dàng hơn, bởi nhiều khi sản phẩm IOT bị đóng kín, không có các chân để giao tiếp với bên ngoài mà chỉ giao tiếp qua các chuẩn không dây
* Thuận tiện trong việc bảo trì, nâng cấp firmware.
* Cập nhật các tính năng mới của sản phẩm và app

Tuy nhiên khi FOTA chúng ta sẽ gặp những vấn đề sau:

* Tốn bộ nhớ khi sử dụng
* Cần bảo mật các giao thức nạp hoặc sử dụng user dành riêng cho việc nạp

## Lập trình ESP32 FOTA nạp firmware từ xa trong mạng local wifi

Trong bài này chúng ta sử dụng ESP32 FOTA, nạp chương trình blynk led bằng wifi cho ESP32, sử dụng webserver được tạo ra bởi ESP32.



### Thiết lập trên server ESP32

Thiết lập style, màu sắc của web page



Thiết lập giao diện login

Để đổi tài khoản và mật khẩu login các bạn sửa câu lệnh: "if(form.userid.value=='admin' && form.pwd.value=='khuenguyencreator')"

Trong bài này ta sử dụng:

* ID: admin
* Password: khuenguyencreator

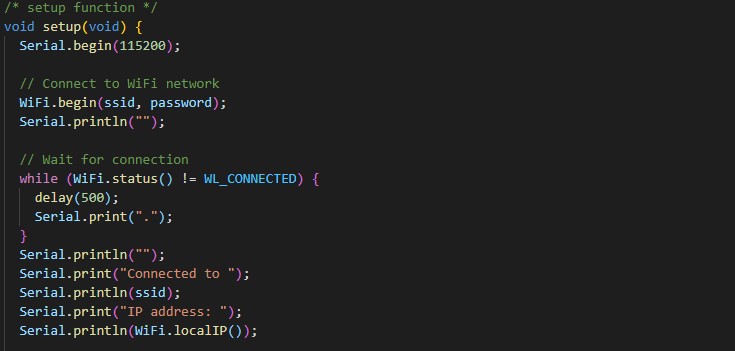


Thiết lập giao diện upload

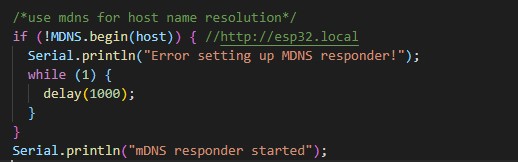


### Cập nhập firmware trên ESP32

Kết nối với wifi, quá quen thuộc với bạn rồi.



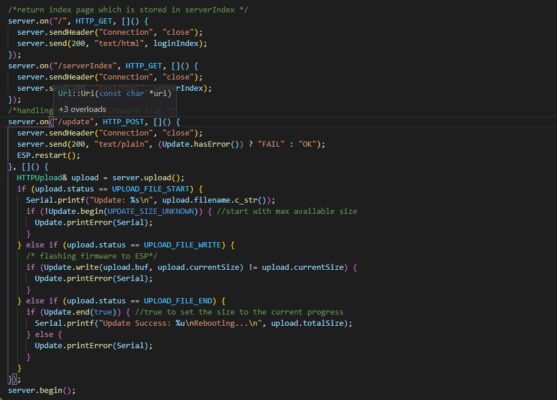
Sử dụng mDNS để truy cập đơn giản hơn, không cần sử dụng ip



Nạp firmware với file upload lên. Phần này chúng ta sẽ thêm các cách xử lý cho ESP32 mỗi event sảy ra

* Event khi vào trang login
* Event khi đăng nhập, vào trang upload
* Event khi upload firmware và nạp vào esp32

Để nạp firmware vào ESP32 ta sử dụng thư viện Update, với các hàm Update.begin , Update.buf , Update.end



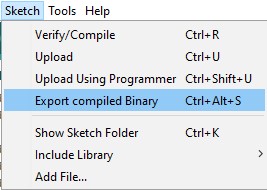
Trong loop chúng ta bật hàm xử lý các event. Và delay 1ms.

### Full code

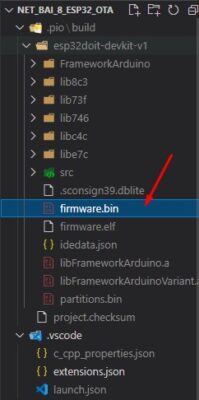
|  |  |
| --- | --- |
| 002  003  004  005  006  007  008  009  010  011  012  013  014  015  016  017  018  019  020  021  022  023  024  025  026  027  028  029  030  031  032  033  034  035  036  037  038  039  040  041  042  043  044  045  046  047  048  049  050  051  052  053  054  055  056  057  058  059  060  061  062  063  064  065  066  067  068  069  070  071  072  073  074  075  076  077  078  079  080  081  082  083  084  085  086  087  088  089  090  091  092  093  094  095  096  097  098  099  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153 | #include <Arduino.h>  #include <WiFi.h>  #include <WiFiClient.h>  #include <WebServer.h>  #include <ESPmDNS.h>  #include <Update.h>    **const** **char**\* host = "esp32";  **const** **char**\* ssid = "hellovtag";  **const** **char**\* password = "12345678";    WebServer server(80);    /\* Style \*/  String style =  "<style>#file-input,input{width:100%;height:44px;border-radius:4px;margin:10px auto;font-size:15px}"  "input{background:#f1f1f1;border:0;padding:0 15px}body{background:#3498db;font-family:sans-serif;font-size:14px;color:#777}"  "#file-input{padding:0;border:1px solid #ddd;line-height:44px;text-align:left;display:block;cursor:pointer}"  "#bar,#prgbar{background-color:#f1f1f1;border-radius:10px}#bar{background-color:#3498db;width:0%;height:10px}"  "form{background:#fff;max-width:258px;margin:75px auto;padding:30px;border-radius:5px;text-align:center}"  ".btn{background:#3498db;color:#fff;cursor:pointer}</style>";    /\* Login page \*/  String loginIndex =  "<form name=loginForm>"  "<h1>ESP32 FOTA</h1>"  "<input name=userid placeholder='User ID'> "  "<input name=pwd placeholder=Password type=Password> "  "<input type=submit onclick=check(this.form) class=btn value=Login></form>"  "<script>"  "function check(form) {"  "if(form.userid.value=='admin' && form.pwd.value=='khuenguyencreator')"  "{window.open('/serverIndex')}"  "else"  "{alert('Error Password or Username')}"  "}"  "</script>" + style;    /\* Server Index Page \*/  String serverIndex =  "<script src='<https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js>'></script>"  "<form method='POST' action='#' enctype='multipart/form-data' id='upload\_form'>"  "<input type='file' name='update' id='file' onchange='sub(this)' style=display:none>"  "<label id='file-input' for='file'>   Choose file...</label>"  "<input type='submit' class=btn value='Update'>"  "<br><br>"  "<div id='prg'></div>"  "<br><div id='prgbar'><div id='bar'></div></div><br></form>"  "<script>"  "function sub(obj){"  "var fileName = obj.value.split('\\\\');"  "document.getElementById('file-input').innerHTML = '   '+ fileName[fileName.length-1];"  "};"  "$('form').submit(function(e){"  "e.preventDefault();"  "var form = $('#upload\_form')[0];"  "var data = new FormData(form);"  "$.ajax({"  "url: '/update',"  "type: 'POST',"  "data: data,"  "contentType: false,"  "processData:false,"  "xhr: function() {"  "var xhr = new window.XMLHttpRequest();"  "xhr.upload.addEventListener('progress', function(evt) {"  "if (evt.lengthComputable) {"  "var per = evt.loaded / evt.total;"  "$('#prg').html('progress: ' + Math.round(per\*100) + '%');"  "$('#bar').css('width',Math.round(per\*100) + '%');"  "}"  "}, false);"  "return xhr;"  "},"  "success:function(d, s) {"  "console.log('success!') "  "},"  "error: function (a, b, c) {"  "}"  "});"  "});"  "</script>" + style;    /\* setup function \*/  **void** setup(**void**) {    Serial.begin(115200);      // Connect to WiFi network    WiFi.begin(ssid, password);    Serial.println("");      // Wait for connection  **while** (WiFi.status() != WL\_CONNECTED) {      delay(500);      Serial.print(".");    }    Serial.println("");    Serial.print("Connected to ");    Serial.println(ssid);    Serial.print("IP address: ");    Serial.println(WiFi.localIP());      /\*use mdns for host name resolution\*/  **if** (!MDNS.begin(host)) { //[http://esp32](http://esp32/)      Serial.println("Error setting up MDNS responder!");  **while** (1) {        delay(1000);      }    }    Serial.println("mDNS responder started");      /\*return index page which is stored in serverIndex \*/    server.on("/", HTTP\_GET, []() {      server.sendHeader("Connection", "close");      server.send(200, "text/html", loginIndex);    });    server.on("/serverIndex", HTTP\_GET, []() {      server.sendHeader("Connection", "close");      server.send(200, "text/html", serverIndex);    });    /\*handling uploading firmware file \*/    server.on("/update", HTTP\_POST, []() {      server.sendHeader("Connection", "close");      server.send(200, "text/plain", (Update.hasError()) ? "FAIL" : "OK");      ESP.restart();    }, []() {      HTTPUpload& upload = server.upload();  **if** (upload.status == UPLOAD\_FILE\_START) {        Serial.**printf**("Update: %s\n", upload.filename.c\_str());  **if** (!Update.begin(UPDATE\_SIZE\_UNKNOWN)) { //start with max available size          Update.printError(Serial);        }      } **else** **if** (upload.status == UPLOAD\_FILE\_WRITE) {        /\* flashing firmware to ESP\*/  **if** (Update.write(upload.buf, upload.currentSize) != upload.currentSize) {          Update.printError(Serial);        }      } **else** **if** (upload.status == UPLOAD\_FILE\_END) {  **if** (Update.end(**true**)) { //true to set the size to the current progress          Serial.**printf**("Update Success: %u\nRebooting...\n", upload.totalSize);        } **else** {          Update.printError(Serial);        }      }    });    server.begin();  }    **void** loop(**void**) {    server.handleClient();    delay(1);  } |
| Full Code | |

### Kết quả

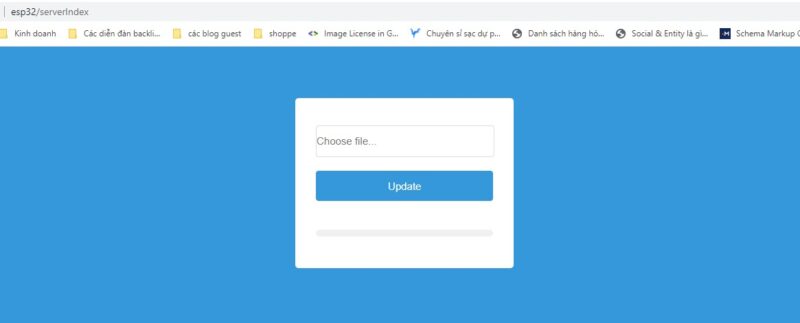
Sau khi nạp code vào ESP32, chung ta sử dụng Arduino, mở example blink led. Tạo bin file như sau:



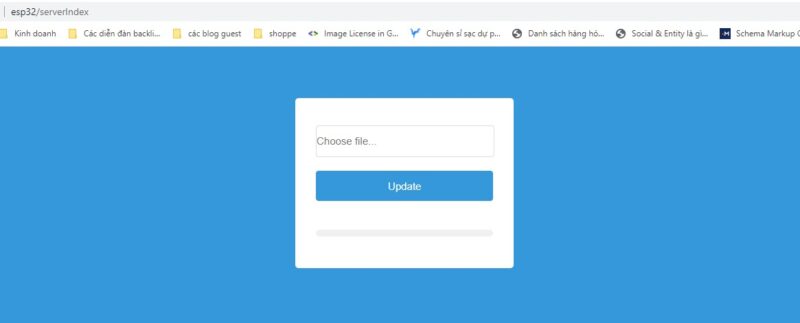
Nếu sử dụng platformio, các bạn nhấn build project, bin file sẽ nằm trong thư mục .pio theo đường dẫn



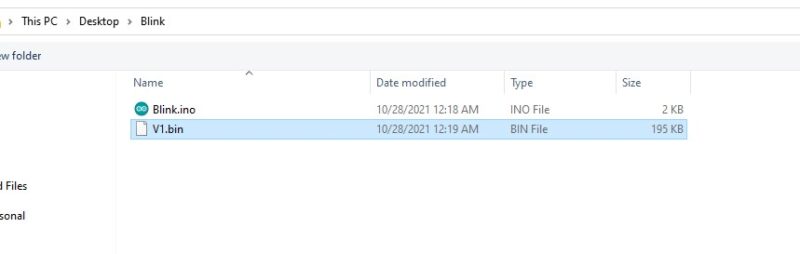
Để upload firmware, chúng ta mở trình duyệt, vào ip hoặc mDNS của ESP32.



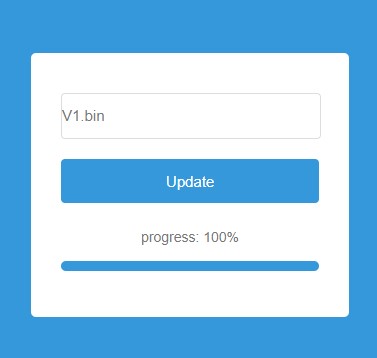
Login vào đúng account đã tạo



Nhấn update, tìm tới file bin đã tạo ra.



Nhấn update, kết quả ESP32 sẽ reboot và chạy firmware của blink, nháy led liên tục.





**Lưu ý:** Vì firmware blink không có fota, vậy nên chúng ta không thể fota cho esp32 sau khi nạp code blink. Để có thể sử dụng fota tiếp thì bắt buộc bản firmware bạn nạp vào cũng cần có chức năng fota

## Kết

FOTA là chức năng quan trọng trong lập trình IOT, tuy rằng FOTA sử dụng mạng nội bộ không được sử dụng nhiều trong các sản phẩm thực tế, tuy nhiên chúng vẫn có nhiều ứng dụng. Thông thường firmware sẽ được lưu tại 1 server trên internet và ESP32 sẽ kết nối và download firmware từ đó. Giúp lập trình viên quản lí version dễ dàng hơn, chi tiết chúng ta sẽ học trong bài sau.

Cám ơn các bạn đã đọc bài viết, cùng vào hội [Anh Em Nghiện Lập Trình](https://www.facebook.com/groups/nghienlaptrinh26) để cùng trao đổi nhé

* Nếu update firmware thất bại thì thế nào ?
* Update boot bằng cách nào ?
* Làm thế nào để đảm bảo update firmware thành công hoặc đảm bảo lỗi thì hệ thống vẫn hoạt động được bình thường ?
* Các lỗi thường gặp khi update firmware ?
* Làm thế nào tăng tính bảo mật cho việc update firmware từ xa ?

# Cập nhật firmware

## Giới thiệu OTA

Cập nhật firmware OTA (Over the Air) là tiến trình tải firmware mới vào ESP module thay vì sử dụng cổng Serial. Tính năng này thực sự rất hữu dụng trong nhiều trường hợp giới hạn về kết nối vật lý đến ESP Module.

OTA có thể thực hiện với:

* [Arduino IDE](https://arduino.esp8266.vn/network/ota.html#arduino-ide0)
* [Web Browser](https://arduino.esp8266.vn/network/ota.html#web-browser0)
* [HTTP Server](https://arduino.esp8266.vn/network/ota.html#http-server0)

Sử dụng OTA với tùy chọn dùng **Arduino IDE** trong quá trình phát triển, thử nghiệm, 2 tùy chọn còn lại phù hợp cho việc triển khai ứng dụng thực tế, cung cấp tính năng cập nhật OTA thông qua web hay sử dụng HTTP Server.

Trong tất cả các trường hợp, thì Firmware hỗ trợ OTA phải được nạp lần đầu tiên qua cổng Serial, nếu mọi thứ hoạt động trơn tru, logic ứng dụng OTA hoạt động đúng thì có thể thực hiện việc cập nhật firmware thông qua OTA.

Sẽ không có đảm bảo an ninh đối với quá trình cập nhật OTA bị hack. Nó phụ thuộc vào nhà phát triển đảm bảo việc cập nhật được phép từ nguồn hợp pháp, đáng tin cậy. Khi cập nhật hoàn tất, ESP8266 sẽ khởi động lại và thực thi code mới. Nhà phát triển phải đảm bảo ứng dụng thực trên module phải được tắt và khởi động lại 1 cách an toàn. Nội dung bên dưới cung cấp bổ sung các thông tin về an ninh, và an toàn cho tiến trình cập nhật OTA.

### Bảo mật

Khi ESP8266 được phép thực thi OTA, có nghĩa nó được kết nối mạng không dây và có khả năng được cập nhập Sketch mới. Cho nên khả năng ESP8266 bị tấn công sẽ nhiều hơn và bị nạp bởi mã thực thi khác là rất cao. Để giảm khả năng bị tấn công cần xem xét bảo vệ cập nhật của bạn với một mật khẩu, cổng sử dụng cố định khác biệt, v.v…

Kiểm tra những tính năng được cung cấp bởi thư viện [ArduinoOTA](https://github.com/esp8266/Arduino/tree/master/libraries/ArduinoOTA) thường xuyên, có thể được nâng cấp khả năng bảo vệ an toàn:

void setPort(uint16\_t port);

void setHostname(const char**\*** hostname);

void setPassword(const char**\*** password);

Một số chức năng bảo vệ đã được xây dựng trong và không yêu cầu bất kỳ mã hóa nào cho nhà phát triển. [ArduinoOTA](https://github.com/esp8266/Arduino/tree/master/libraries/ArduinoOTA) và espota.py sử dụng [Digest-MD5](https://en.wikipedia.org/wiki/Digest_access_authentication) để chứng thực việc tải firmware lên. Đơn giản là đảm bảo tính toàn vẹn của firmware bằng việc tính [MD5](https://en.wikipedia.org/wiki/MD5).

Hãy phân tích rủi ro cho riêng ứng dụng của bạn và tùy thuộc vào ứng dụng mà quyết định những chức năng cũng như thư viện để thực hiện. Nếu cần thiết, có thẻ xem xét việc thực hiện các phương thức bảo vệ khỏi bị hack, ví dụ như cập nhật OTA chỉ cho tải lên chỉ theo lịch trình cụ thể, kích hoạt OTA chỉ được người dùng nhấn nút chuyên dụng “Cập nhật”, v.v…

### An toàn

Quá trình OTA tiêu tốn nguồn tài nguyên và băng thông của ESP8266 khi tải lên. Sau đó, ESP8266 được khởi động lại và một Sketch mới được thực thi. Cần phân tích và kiểm tra ảnh hưởng của quá trình này tới các chức năng cũ và sketch mới của ESP module.

Nếu ESP được đặt ở xa và điều khiển một vài thiết bị, ta nên chú ý tới hoạt động của thiết bị nếu thiết bị ngừng hoạt động đột xuất do quá trình cập nhật. Do đó, ta cần phải xác định được trạng thái làm việc an toàn của thiết bị trước quá trình cập nhật. Ví dụ, module được dùng để điều khiển hệ thống tưới nước tự động trong vườn. Nếu trong quá trình hoạt động mà hệ thống điều khiển bị tắt đột ngột và các van bị mở, thì cả vườn sẽ bị ngập nước.

Các hàm sau đây được cung cấp bởi thư viện [ArduinoOTA](https://github.com/esp8266/Arduino/tree/master/libraries/ArduinoOTA) và được dùng để xử lý ứng dụng trong quá trình cập nhật OTA hoặc để xử lý khi OTA gặp lỗi:

void onStart(OTA\_CALLBACK(fn));

void onEnd(OTA\_CALLBACK(fn));

void onProgress(OTA\_CALLBACK\_PROGRESS(fn));

void onError(OTA\_CALLBACK\_ERROR (fn));

### Yêu cầu căn bản

Bộ nhớ Flash phải có đủ dung lượng để lưu cả sketch cũ (đang vận hành trên hệ thống) và sketch mới (cập nhật OTA).

Hệ thống File và EEPROM cũng cần dung lượng để lưu trữ. Xem [`flash layout`\_](https://arduino.esp8266.vn/network/ota.html#id8) .

Hàm

ESP**.**getFreeSketchSpace();

được dùng để kiểm tra dung lượng trống cho sketch mới.

Để tìm hiểu tống quát về memory layout, nơi mà new sketch được lưu và quá trình copy nó trong quá trình cập nhật OTA,xem [Update process - memory view](https://arduino.esp8266.vn/network/ota.html#update-process-memory-view).

Các phần sau trình bày chi tiết và các phương pháp cụ thể để thưc hiện OTA.

## Arduino IDE

Thưc hiện OTA với module Arduino IDE chỉ được thưc hiện trong các trường hợp điển hình sau: -trong lúc firmware đang hoạt động để thay thế việc sử dụng cổng nối tiếp, - khi cập nhật ít cho modules, -chỉ khi module Arduino IDE và máy tính đang ờ trong cùng một mạng.

### Yêu cầu

ESP và máy tính phải được kết nối cùng một mạng.

### Ví dụ ứng dụng

Các hướng dẫn sau trình bày cách cấu hình OTA trên board NodeMCU 1.0 (ESP-12E Module). Ta có thể dùng các board khác nếu chúng thỏa các yêu cầu trình bảy ở trên. Hướng dẫn này thích hợp với các hệ điều hành hỗ trợ Arduino IDE. Các hình ảnh hướng dẫn thực hiện trên Win7 và bạn có thể gặp một số khác biệt nhỏ (như tên của các cổng nối tiếp), nếu dùng Linux hoặc MacOS if you are using Linux and MacOS.

1. Trước khi bắt đầu, ta phải cài đặt đầy đủ các phần mềm sau:

Arduino IDE 1.6.7 hoặc bản mới nhất - [Latest version](https://www.arduino.cc/en/Main/Software)

esp8266/Arduino platform package 2.0.0 hoặc bản mới nhất - và làm theo hướng dẫn: [See link](https://github.com/esp8266/Arduino#installing-with-boards-manager)

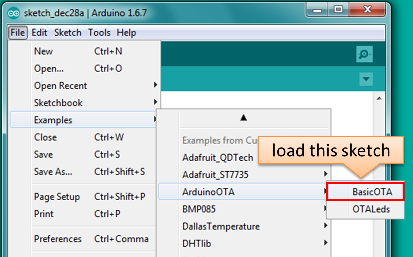
Python 2.7 - [Download Python](https://www.python.org/)

**Chú ý**: Người dùng Windows nên chọn “Add python.exe to Path” (xem hình bên dưới – lựa chọn này không phải là lựa chọn mặc định)

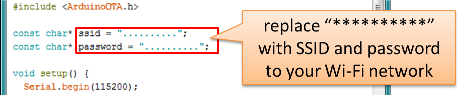


1. Bây giờ, ta chuẩn bị sketch và cấu hình cho việc upload qua cổng nối tiếp:

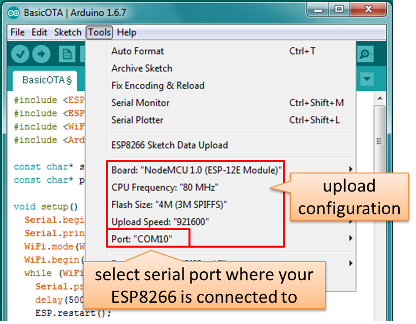
Mở Arduino IDE và load sketch BasicOTA.ino. Vào File > Examples > ArduinoOTA.



Cập nhật SSID và password trong sketch để module có thể truy cập WIFI.

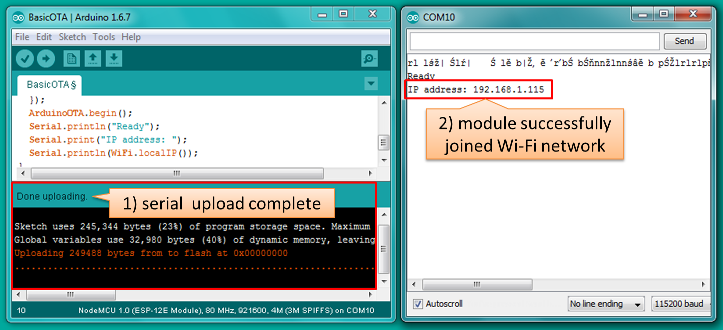


Cấu hình thông số upload như bên dưới (bạn cần điều chỉnh cấu hình cho phù hơp nếu dùng module khác):



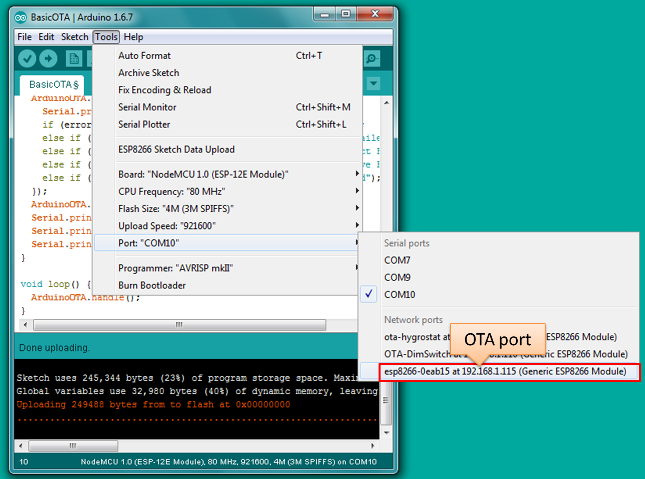
**Chú ý**: Tùy vào phiên bản và board ESP bạn sử dụng, bạn có thể thấy Upload Using: trong menu. Lựa chọn này sẽ không hoạt động và không ảnh hưởng tới lựa chọn của bạn. Chức năng này tương thích với các phiên bản OTA cũ và bỉ gỡ bỏ ở platform package version 2.2.0.

1. Upload sketch (Ctrl+U). Khi hoàn thành, mở Serial Monitor (Ctrl+Shift+M) và kiểm tra module đã trup cập được WIFI chưa:



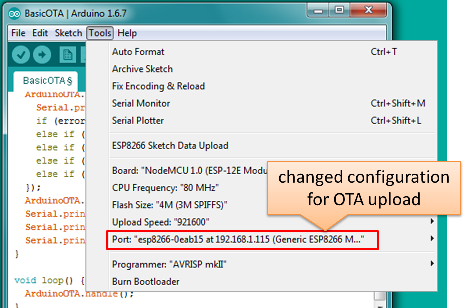
**Chú ý**: ESP module nên được reset sau khi upload xong after. Nếu không thì các bước tiếp theo sẽ không thưc hiện được. Reset có thể được tự động thưc hiện sau khi ta mở màn hình nối tiếp như ở hình trên. Nó phụ thuộc vào cách ta kết nối DTR và RTS từ USB-Serial tới ESP. Nếu phần mềm không reset tự động, thì ta phải ấn nút rest trên board. Để hiểu rõ hơn, cần tìm hiểu hàm ESP.restart().

1. Khi module được kết nối tới mạng, sau vài giây, cổng esp8266-ota sẽ mở trên Arduino IDE. Chọn port với địa chỉ IP hiện trên cửa sổ Serial Monitor ở bước trước.



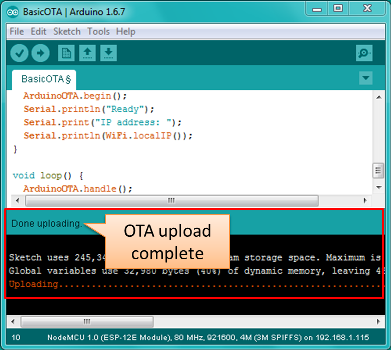
**Chú ý**: Nếu cổng OTA không hiện lên, thoát Arduino IDE, và mở lại. Kiểm tra lại port OTA. Nếu vẫn tiếp tục không hiển thị cổng OTA, kiểm tra tưòng lửa của máy và các cài đặt trên router.

5.Chọn cổng OTA:



**Chú ý**: Mục Upload Speed ở hình trên: Gĩư gía trị này không đổi do nó liên quan tới cổng nối tiếp.

1. Nếu bạn đã thực hiện các bước trên thành công, bạn có thể upload (Ctrl+U) sketch qua OTA.



**Chú ý**: Để có thể upload sketch nhiều lần qua OTA, bạn cần nhúng (embed) lịch OTA vào phầm mềm. Dùng BasicOTA.ino để làm ví dụ.

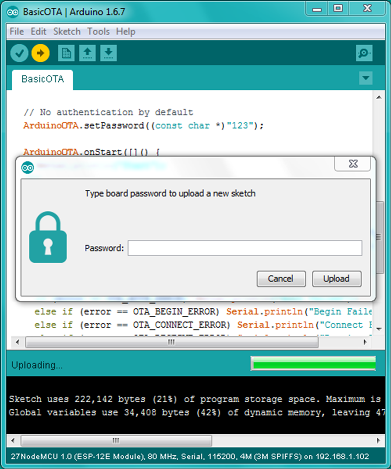
### Bảo mật Password

Bảo vệ quá trình upload OTA với password là một quá trình khá đơn gỉản. Những việc bạn cần làm là dán đoạn code sau vào đoạn code của bạn:

ArduinoOTA**.**setPassword((const char **\***)"123");

Trong đó "123" làn một password ví dụ. Thay "123" bằng password của bạn.

Trước khi thực hiện việc này, ta cùng tìm hiểu cách sử dụng sketch BasicOTA.ino. BasicOTA.ino có thể truy xuất bằng đường dẫn: File > Examples > ArduinoOTA. Mở BasicOTA.ino, bỏ các comment có sẵn trong file và upload sketch. Để việc troubleshooting dễ hơn, không sửa đổi sketch ví dụ, gồm cả sketch OTA password (như với password 123 lúc nãy). Sau đó upload lại sketch một lần nữa (dùng OTA). Sau khi biên dịch và upload xong, cửa sổ sẽ hiện lên yêu cầu nhập password như hình:

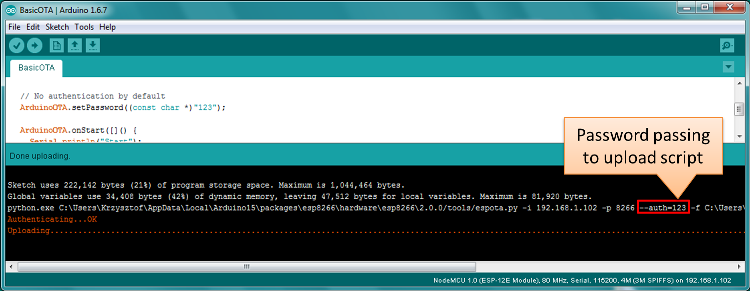


Nhập password và upload như bình thường. Nếu nhập đúng, và chạy thành công,kết quả là thông báo``Authenticating...OK`` trên màn hình



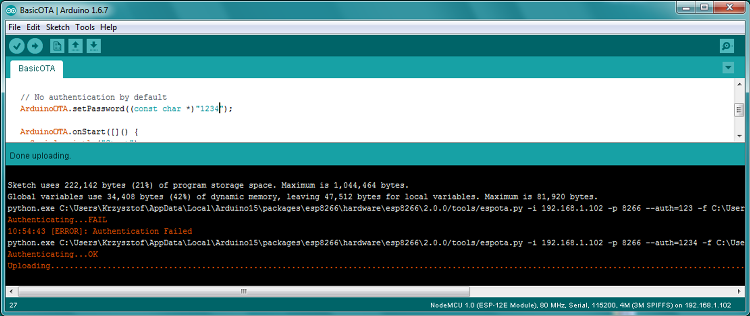
Lần này bạn sẽ không được nhắc yêu cầu phải nhập lại password cho lần sau. Arduino IDE sẽ nhớ password cho bạn. Bạn sẽ nhận được thông báo nhắc nhập lại password khi bạn mở lại IDE hoặc nếu bạn thay đổi password trong sketch và upload sketch và sau đó upload lại một lần nữa.

Cần lưu ý là password cũ có thể bị lộ, nếu IDE không được đóng sau lần upload cuối cùng. Việc này có thể được thực hiện bằng cách cho phép Show verbose ouput during: upload trong File -> Preferences và upload lên module.



Từ hình trên ta có thể thấy password trên log, do nó được đưa tới **espota.py** upload script

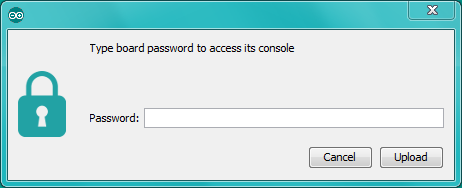
Password cũng có thể bị thay đổi trong khi đang upload như ở hình dưới dây:



Ơ hình trên ta thấy, trong khi upload, Arduino IDE dùng password cũ, do đó việc upload bị thất bại (upload failed) và bị report bởi IDE. Chỉ khi IDE yêu cầu bạn nhập password mới và khi nhập đúng thì việc upload mới thành công.

## Troubleshooting

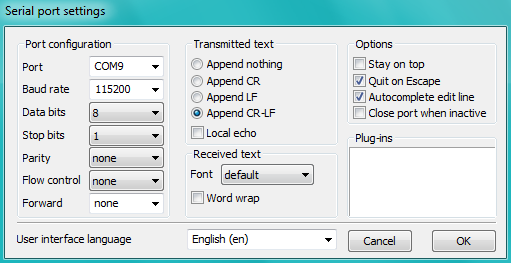
Nếu việc cập nhật OTA thất bại, bước đầu tiên bạn cần làm là kiểm tra tin nhắn báo lỗi (error message) hiện trên cửa sổ upload của Arduino IDE. Nếu việc này không gíup được bạn, hãy upload lại khi kiểm tra các thông tin của ESP hiện trên serial port, Serial Monitor từ IDE sẽ không hữu ích trong trường hợp này. Khi cố sử dụng nó, màn hình sau sẽ hiển thị:



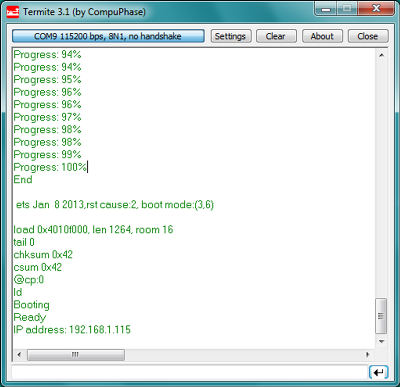
Đây là cửa sổ cho Arduino Ýun và không được thực hiện trên ESP8266. Cửa sổ xưất hiện do IDE cố mở Serial Monitor dùng cổng mạng mà bạn đã chọn cho OTA upload.

Thay vào đó, bạn cần một Serial monitor ngoài. Nếu bạn đang sử dụng Windows, hãy sử dụng [Terminal](https://www.compuphase.com/software_termite.htm) . Đây là một cổng RS232 đơn gỉan mà không đòi hỏi điều khiển dòng RTS và DTR. Hai dạng điều khiển dòng này có thể gây ra vấn đề nếu ta đang sử dụng các đường dây cắm theo thứ tự để thay đổi GPIO và chân RESET trên ESP để upload.

Chọn cổng COM và baud rate trên cổng ngoài nếu bạn dùng Arduino Serial Monitor. Chọn cài đặt chuẩn cho Termite như hình sau:

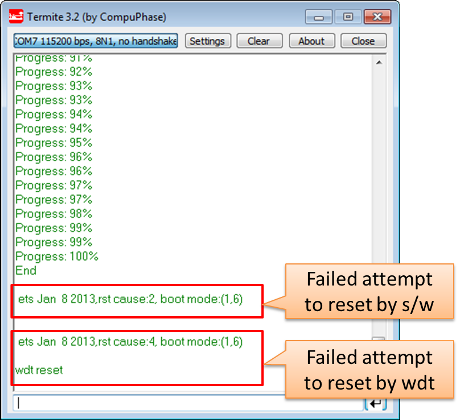


Sau đó chạy OTA từ IDE và theo dõi kết quả hiển thị trên cổng. Quá trình ArduinoOTA thành công dùng BasicOTA.ino sketch như hình dưới (Địa chỉ IP tùy thuộc vào cấu hình mạng của máy bạn)



Nếu upload thất bại, bạn sẽ thấy các lỗi báo về từ IDE, từ các lỗi ngoại lệ (exception) và stack trace (như thiếu cấu trúc try ... catch hoặc throws trong lập trình Java/Android)

Nhưng khác với ở các trường hợp trên khi lỗi được hiển thị trên màn hình compile của IDE, lỗi lần này được hiển thị như hình sau:



Trong trường hợp này, phần lớn các module ESP đều không được reset sau lần upload đầu dùng serial port (như các thao tác đã trình bày ở phần Arduino IDE ở trên)

Các nguyên nhân phổ biến gây lỗi OTA như sau:

* Không đủ dung lượng bộ nhớ trên chip (ví dụ như ESP01 với 512KB bộ nhớ flash không đủ cho OTA).
* Quá nhiều khai báo bộ nhớ cho SPIFFS nên sketch mới sẽ không phù hợp gĩư sketch và SPIFFS (xem lại Giới thiệu về OTA, quá trình update).
* Quá ít khai báo bộ nhớ trong Arduino IDE để3 bạn lựa chọn (tức là ít hơn kích thước vật lý).
* Không reset module ESP sau lần upload đầu dùng serial port.

## Web Browser

Việc cập nhật được mô tả trong phần này được thực hiện với một web browser. Việc này có thể hữu ích trong các trường hợp sau: \* Sau khi cài đặt ứng dụng nếu tải trực tiếp từ Arduino IDE không thuận tuận hoặc không thể tải được do sự cố nào đó \* Sau khi cài đặt ứng dụng nếu người dùng không thể update OTA bằng các server update ngoài. \* Để update một phần nhỏ lên thiết bị sau khi cài đặt khi cài đặt update server không thực tế.

### Yêu cầu

Module ESP và máy tính phải được kết nối cùng một mạng.

### Tổng quan

Cập nhật với web browser được thực hiện bằng lớp ESP8266HTTPUpdateServer cùng với 2 lớp: ESP8266WebServer và ESP8266mDNS

Code:

setup()

MDNS**.**begin(host);

httpUpdater**.**setup(**&**httpServer);

httpServer**.**begin();

MDNS**.**addService("http", "tcp", 80);

loop()

httpServer**.**handleClient();

### Ưng dụng ví dụ:

Để thực hiện ví dụ, bạn cần:

* Sketch ví dụ WebUpdater.ino trong thư viện ESP8266HTTPUpdateServer.
* NodeMCU 1.0 (ESP-12E Module).

Bạn cũng có thể dùng module khác nếu nó thõa yêu cầu trên

1. Để bắt đầu, bạn cần phải có đủ các phần mềm sau đây:

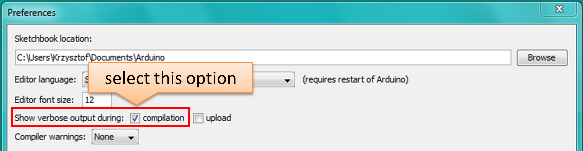
Arduino IDE và phiên bản 2.0.0-rc1 (of Nov 17, 2015): [Download](https://github.com/esp8266/Arduino#installing-with-boards-manager)

**Host software tùy thuộc vào hệ điều hành bạn sử dụng:**

Avahi với Linux: [Download Avahi](http://avahi.org/) Bonjour với Windows [Download Bonjour](http://www.apple.com/support/bonjour/) Mac OSX và iOS: đã có hỗ trợ sẵn trên Hệ điều hành

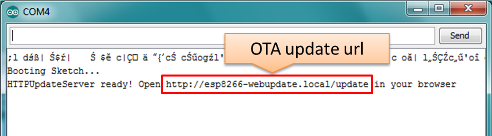
1. Chuẩn bị sketch và cấu hình cho lần upload đầu với cổng nối tiếp.

* Mở Arduino IDE và load sketch WebUpdater.ino (nàm ở File > Examples > ESP8266HTTPUpdateServer).
* Update SSID and password vào sketch để cho module có thể truy cập WIFI.
* Mở File > Preferences, tìm “Show verbose output during:” và ấn chọn “compilation”.



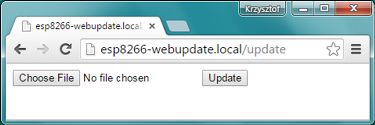
Chú ý: Việc chọn cài đặt compilation này cũng được yêu cầu tương tự ở 5 bước sau. Bạn có thể không chọn "compilation" ở các bước sau.

1. Upload sketch (Ctrl+U). Khi thực hiện xong, mở Serial Monitor (Ctrl+Shift+M) và kiểm tra nếu bạn thấy các tin nhắn như hình sau, bao gồm cả URL cho cập nhật OTA



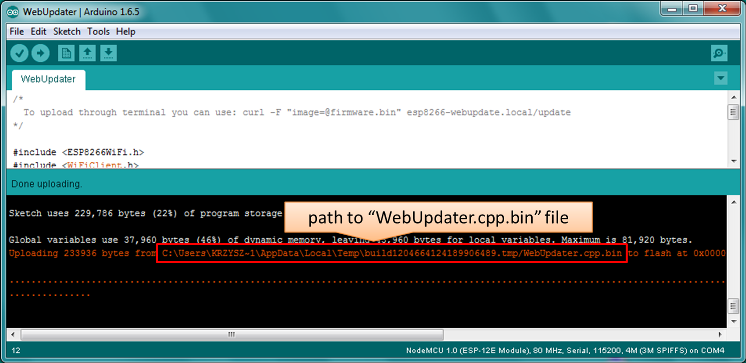
Chú ý: Những tin nhắn này chỉ hiển thị khi module đã truy cập được WIFI và sẵn sàng upload OTA. Cũng giống như với module Arduino IDE, sau khi upload xong thì ta phải reset lại module.

* 1. Sau đó mở web browser lên và paste URL ở trên (i.e. <http://esp8266-webupdate.local/update>). Sau khi thưc hiện xong, browser sẽ có dạng như hình dưới. Bạn có thể chọn file để update.

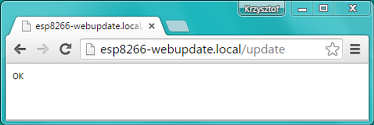


Chú ý: Khi đã nhập <http://esp8266-webupdate.local/update> mà không thưc hiện được, hãy thay esp8266-webupdate với module địa chỉ IP. Ví dụ, nếu IP của module là 192.168.1.100 thì URL phải là <http://192.168.1.100/update>. Phương pháp này hữu hiệu trong trường hợp host software cài đặt ở bước 1 không hoạt động. Nếu vẫn không thưc hiện được nữa và Serial Monitor cũng không có tin nhắn cảnh báo nào thì bạn có thể tự tìm lỗi bằng cách mở URL được cung cấp trên Google Chrome, ấn F12 và kiểm tra nội dụng trong tab "Console" và "Network". Google sẽ cung cấp nhiều tính năng hỗ trợ ở các tab này.

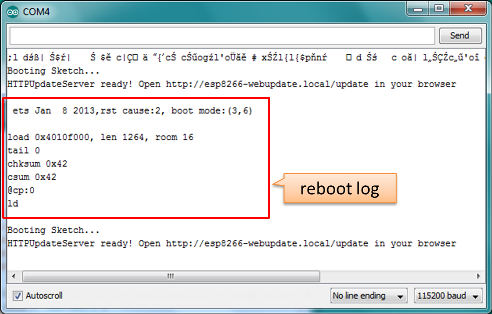
* 1. Để có được file, ta chuyển folder làm việc tới folder của Arduino IDE để lưu kết quả biên dịch. Ta có thể kiểm tra đường dẫn tới file sau khi xem kết quả biên dịch như ở hình dưới.



Bây giờ chọn "Choose File" trong web browser, đến nơi lưu file làm việc như đã trình bày ở bước 5, tìm file "WebUpdater.cpp.bin" và upload lên. Nếu upload thành công, ta sẽ có kết quả như hình dưới.



Kết quả reboot có thể thấy trên cửa sổ Serial Monitor như hình dưới:



Sau khi reboot, ta sẽ nhận được tin nhắn HTTPUpdateServer ready! Open http:// esp8266-webupdate.local /update in your browser như ở bước 3, bởi vì module được load lại bằng code cũ (dùng serial port), sau đó dùng OTA.

Sau khi bạn đã quen với các bước thưc hiện này, ta có thể điều chỉnh WebUpdater.ino sketch và xuất ra các tin nhắn theo ý muốn, sau đó biên dịch chương trình, sau đó upload file biên dịch dùng Web browser. Kết quả sẽ hiển thị trên Serial Monitor.

Ta cũng có thể lên kế hoạch update OTA trong sketch, như hướng dẫn Tổng quan ở trên. Nếu thưc hiện đúng, ta có thể dùng web browser upload sketch mới như là bản cập nhật mới, bên cạnh sketch cũ đã update.

Trong trường hợp update OTA thất bại sau khi đã sửa đổi sketch, ta có thể khôi phục lại module bằng cách load lại sketch cũ đã upload. Sau đó debug lỗi sketch dùng Serial Monitor. Sau khi sửa lỗi xong thì update OTA lại,

## HTTP Server

Lớp''ESPhttpUpdate'' có thể dùng để kiểm tra các update và download một file dạng nhị phân từ HTTP web server. Ta có thể download các update từ mọi địa chỉ IP hoặc tên miền trên Internet.

### Yêu cầu

* web server

### Arduino code

### Simple updater

Simple updater sẽ download file mỗi khi hàm được gọi.

ESPhttpUpdate.update("192.168.0.2", 80, "/arduino.bin");

### Advanced updater

Ta cũng có thể gọi hàm update bằng một đoạn script trên server. Nếu ta có đối số của version, nó sẽ được gởi lên server. Phía server sẽ dùng nó để kiểm tra xem đã đủ điều kiện thực hiện update được chưa.

Script trên server có các kết quả trả về như:

* (Mã) 200: Yêu cầu gởi hình ảnh firmware.
* (Mã) 304: Thông báo là không có bản update mới.

t\_httpUpdate\_return ret **=** ESPhttpUpdate**.**update("192.168.0.2", 80, "/esp/update/arduino.php", "optional current version string here");

switch(ret) {

case HTTP\_UPDATE\_FAILED:

Serial**.**println("[update] Update failed.");

**break**;

case HTTP\_UPDATE\_NO\_UPDATES:

Serial**.**println("[update] Update no Update.");

**break**;

case HTTP\_UPDATE\_OK:

Serial**.**println("[update] Update ok."); **//** may **not** called we reboot the ESP

**break**;

}

### Server request handling

### Simple updater

Với simple updater, server chỉ cần file nhị phân để update.

### Advanced updater

Với advanced update thì script phải chạy trên server, ví dụ như một PHP script. Mỗi lần yêu cầu update, ESP sẽ gởi thông tin lên qua giao thức HTTP bằng các HTTP header tới server.

Ví dụ về header data:

[HTTP\_USER\_AGENT] **=>** ESP8266**-**http**-**Update

[HTTP\_X\_ESP8266\_STA\_MAC] **=>** 18:FE:AA:AA:AA:AA

[HTTP\_X\_ESP8266\_AP\_MAC] **=>** 1A:FE:AA:AA:AA:AA

[HTTP\_X\_ESP8266\_FREE\_SPACE] **=>** 671744

[HTTP\_X\_ESP8266\_SKETCH\_SIZE] **=>** 373940

[HTTP\_X\_ESP8266\_SKETCH\_MD5] **=>** a56f8ef78a0bebd812f62067daf1408a

[HTTP\_X\_ESP8266\_CHIP\_SIZE] **=>** 4194304

[HTTP\_X\_ESP8266\_SDK\_VERSION] **=>** 1.3**.**0

[HTTP\_X\_ESP8266\_VERSION] **=>** DOOR**-**7**-**g14f53a19

Với các thông tin cung cấp thì script có thể kiểm tra nếu có thể update được. Ta cũng có thể làm tương tự với địa chỉ MAC khi yêu cầu update:

Script ví dụ (Với địa chỉ MAC):

<?PHP

header('Content-type: text/plain; charset=utf8', true);

function check\_header($name, $value = false) {

if(!isset($\_SERVER[$name])) {

return false;

}

if($value && $\_SERVER[$name] != $value) {

return false;

}

return true;

}

function sendFile($path) {

header($\_SERVER["SERVER\_PROTOCOL"].' 200 OK', true, 200);

header('Content-Type: application/octet-stream', true);

header('Content-Disposition: attachment; filename='.basename($path));

header('Content-Length: '.filesize($path), true);

header('x-MD5: '.md5\_file($path), true);

readfile($path);

}

if(!check\_header('HTTP\_USER\_AGENT', 'ESP8266-http-Update')) {

header($\_SERVER["SERVER\_PROTOCOL"].' 403 Forbidden', true, 403);

echo "only for ESP8266 updater!\n";

exit();

}

if(

!check\_header('HTTP\_X\_ESP8266\_STA\_MAC') ||

!check\_header('HTTP\_X\_ESP8266\_AP\_MAC') ||

!check\_header('HTTP\_X\_ESP8266\_FREE\_SPACE') ||

!check\_header('HTTP\_X\_ESP8266\_SKETCH\_SIZE') ||

!check\_header('HTTP\_X\_ESP8266\_SKETCH\_MD5') ||

!check\_header('HTTP\_X\_ESP8266\_CHIP\_SIZE') ||

!check\_header('HTTP\_X\_ESP8266\_SDK\_VERSION')

) {

header($\_SERVER["SERVER\_PROTOCOL"].' 403 Forbidden', true, 403);

echo "only for ESP8266 updater! (header)\n";

exit();

}

$db = array(

"18:FE:AA:AA:AA:AA" => "DOOR-7-g14f53a19",

"18:FE:AA:AA:AA:BB" => "TEMP-1.0.0"

);

if(!isset($db[$\_SERVER['HTTP\_X\_ESP8266\_STA\_MAC']])) {

header($\_SERVER["SERVER\_PROTOCOL"].' 500 ESP MAC not configured for updates', true, 500);

}

$localBinary = "./bin/".$db[$\_SERVER['HTTP\_X\_ESP8266\_STA\_MAC']].".bin";

// Check if version has been set and does not match, if not, check if

// MD5 hash between local binary and ESP8266 binary do not match if not.

// then no update has been found.

if((!check\_header('HTTP\_X\_ESP8266\_SDK\_VERSION') && $db[$\_SERVER['HTTP\_X\_ESP8266\_STA\_MAC']] != $\_SERVER['HTTP\_X\_ESP8266\_VERSION'])

|| $\_SERVER["HTTP\_X\_ESP8266\_SKETCH\_MD5"] != md5\_file($localBinary)) {

sendFile($localBinary);

} else {

header($\_SERVER["SERVER\_PROTOCOL"].' 304 Not Modified', true, 304);

}

header($\_SERVER["SERVER\_PROTOCOL"].' 500 no version for ESP MAC', true, 500);

### Stream Interface

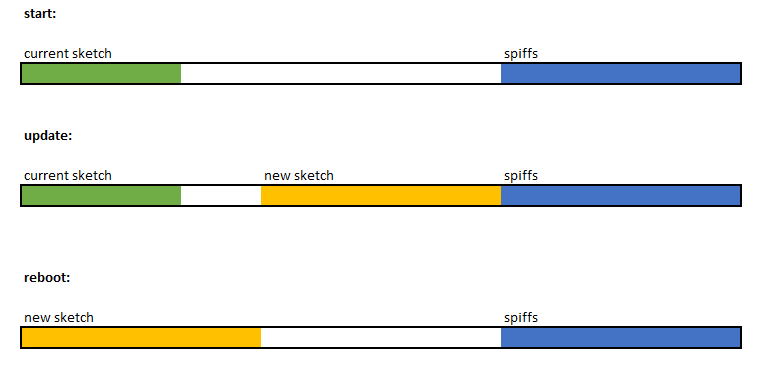
The Stream Interface dựa trên các mode update khác đã trình bày ở trên như OTA, HTTP server/client.

### Updater class

Updater nằm trong Core và dùng để xử lí việc viết firmware lên flash, kiểm tra tính thống nhất của nó và thông báo bootloader để load firmware trên boot kế tiếp.

### Update process - memory view

* Sketch mới sẽ được chứa trong dung lượng trống gĩưa sketch cũ và spiff will be stored in the space between the old sketch and the spiff.
* Trong lần reboot tiếp theo thì “eboot” bootloader kiểm tra các câu lệnh.
* Sketch mới sẽ được copy.
* Sketch mới khởi động.



**Lưu ý:**

Tốt nhất là bạn nên hiểu các phần trên một cách cặn kẽ, thực hành viết và sử dụng các món trên, không nên đọc tài liệu kiểu học thuộc rồi trình bày, như thế sẽ dính sạn ngay.

Trước / Sau khi phỏng vấn hoàn tất, các bạn nên hỏi về xíu về thông tin dự án mà đơn vị đang ứng tuyển, như thế sẽ mở ra câu chuyện thoải mái hơn để 2 bên trao đổi, nhất là tại thời điểm bạn cảm thấy cuộc phỏng vấn hơi áp lực, bạn nên làm ngụm nước rồi đặt câu hỏi để hỏi ngược lại người phỏng vấn.