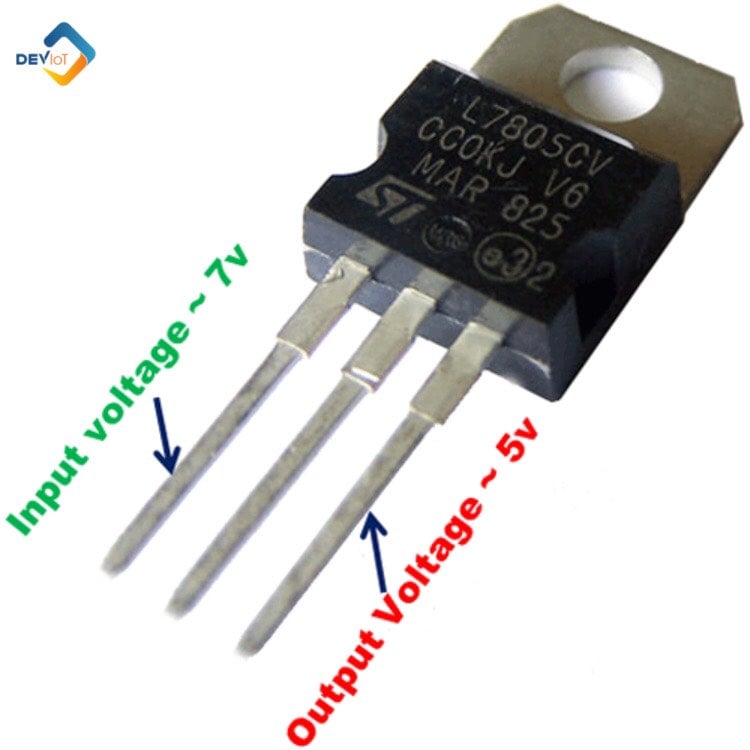
**6 thông số quan trọng khi chọn IC nguồn tuyến tính (LDO)**

IC nguồn là một thiết bị đơn giản và tiết kiệm chi phí có thể thay đổi điện áp đầu vào thành một mức khác ở đầu ra và có thể duy trì điện áp đầu ra không đổi ngay cả trong các điều kiện tải khác nhau. Hầu hết tất cả các thiết bị điện tử từ bộ sạc điện thoại di động đến máy điều hòa không khí cho đến thiết bị cơ điện phức tạp đều sử dụng IC nguồn để cung cấp các điện áp DC khác nhau cho các thành phần khác nhau trong thiết bị.

IC nguồn là một thiết bị đơn giản và tiết kiệm chi phí có thể thay đổi điện áp đầu vào thành một mức khác ở đầu ra và có thể duy trì điện áp đầu ra không đổi ngay cả trong các điều kiện tải khác nhau. Hầu hết tất cả các thiết bị điện tử từ bộ sạc điện thoại di động đến máy điều hòa không khí cho đến thiết bị cơ điện phức tạp đều sử dụng IC nguồn để cung cấp các điện áp DC khác nhau cho các thành phần khác nhau trong thiết bị.



Vì vậy, trong bài viết này, chúng ta sẽ thảo luận về một số thông số quan trọng cần lưu ý trong khi lựa chọn IC nguồn cho dự án của bạn.

***Điện áp đầu vào (Input Voltage) và điện áp đầu ra (Output Voltage)***

Bước đầu tiên để chọn IC nguồn là biết điện áp đầu vào và điện áp đầu ra mà bạn sẽ làm việc. IC nguồn tuyến tính (LDO) cần điện áp đầu vào cao hơn điện áp đầu ra. Nếu điện áp đầu vào nhỏ hơn điện áp đầu ra mong muốn sẽ dẫn đến tình trạng không đủ điện áp khiến bộ điều chỉnh không cung câp đúng điện áp đầu ra.

Ví dụ: nếu bạn đang sử dụng IC nguồn và muốn điện áp đầu ra 5V và điện áp rơi 2V, thì điện áp đầu vào ít nhất phải bằng 7V. Điện áp đầu vào dưới 7V sẽ dẫn đến điện áp đầu ra không được kiểm soát.

***Điện áp rơi (Drop out Voltage)***

Điện áp rơi là hiệu giữa điện áp đầu vào và đầu ra của IC. Ví dụ: Điện áp đầu vào cho LM7805 là 7V, và điện áp đầu ra là 5V, do đó, nó có điện áp rơi 2V. Nếu điện áp đầu vào thấp hơn điện áp đầu ra (5V) + điện áp rơi (2V) => điện áp đầu ra không được kiểm soát có thể làm hỏng thiết bị của bạn. Vì vậy, trước khi chọn một bộ điều chỉnh điện áp, hãy kiểm tra điện áp rơi. Bộ điều chỉnh tuyến tính có thể cực kỳ hiệu quả khi chúng được vận hành với điện áp đầu vào rất thấp. Vì vậy, nếu bạn đang sử dụng pin làm nguồn điện thì bạn có thể sử dụng IC nguồn tuyến tính LDO để có hiệu quả tốt hơn.

***Sự thất thoát điện năng (Power Dissipation)***

IC nguồn tuyến tính (LDO) tiêu hao nhiều điện năng hơn IC nguồn xung (switching voltage). Sự tiêu hao quá mức có thể gây hao pin, quá nhiệt hoặc làm hỏng sản phẩm. Vì vậy, nếu bạn đang sử dụng IC nguồn tuyến tính, trước tiên hãy tính toán công suất tiêu tán. Đối với IC nguồn tuyến tính, công suất tiêu tán có thể được tính bằng:

Power = (Input Voltage – Output Voltage) x Current

Bạn có thể sử dụng IC nguồn xung để tránh vấn đề tiêu tán điện.

***Hiệu suất (Efficiency)***

Hiệu suất là tỷ số giữa công suất đầu ra và công suất đầu vào tỷ lệ với tỷ số giữa điện áp đầu ra và điện áp đầu vào. Vì vậy, hiệu suất của IC nguồn bị giới hạn trực tiếp bởi điện áp rơi và dòng tĩnh vì điện áp rơi càng cao thì hiệu suất càng thấp. Để đạt hiệu suất cao hơn, phải giảm thiểu điện áp rơi và dòng điện tĩnh, và sự chênh lệch điện áp giữa đầu vào và đầu ra phải được giảm thiểu.

***Độ chính xác điện áp (Voltage Accuracy)***

Độ chính xác tổng thể của bộ điều chỉnh điện áp phụ thuộc vào điều chỉnh dòng, điều chỉnh tải, độ lệch điện áp tham chiếu, độ lệch sai số điện áp bộ khuếch đại và hệ số nhiệt. IC nguồn tuyến tính điển hình thường có thông số kỹ thuật về điện áp đầu ra đảm bảo đầu ra được điều chỉnh sẽ nằm trong khoảng 5%. Vì vậy, nếu bạn đang sử dụng bộ điều chỉnh điện áp để cấp nguồn cho các IC số, thì dung sai 5% không phải là mối quan tâm lớn.

***Load Regulation***

Điều chỉnh tải được định nghĩa là khả năng của mạch để duy trì điện áp đầu ra xác định trong các điều kiện tải khác nhau. Quy định tải trọng được thể hiện như sau:

Load Regulation = ∆Vout/ ∆Iout

**Mạch phản chiếu dòng điện**

Author: Tuấn Anh

**I. Định nghĩa**

Mạch phản chiếu dòng điện hay Current mirror circuit là một kỹ thuật phổ biến rộng rãi cho các thiết kế nguyên khối IC. Trong kỹ thuật này, mạch được thiết kế theo cách mà nó sao chép dòng điện qua một thiết bị hoạt động sang một thiết bị hoạt động khác với tính năng điều khiển dòng điện. Trong trường hợp này, dòng điện chạy qua một thiết bị có thể được sao chép vào thiết bị khác nhưng ở dạng nghịch lưu. Nếu dòng điện của thiết bị đầu tiên bị thay đổi, đầu ra dòng điện được nhân đôi của thiết bị kia cũng sẽ thay đổi. Vì vậy, bằng cách kiểm soát dòng điện trong một thiết bị, dòng điện trong một thiết bị khác cũng có thể được kiểm soát. Do đó, mạch phản chiếu dòng điện thường được gọi là nguồn dòng điện được điều khiển dòng hoặc CCCS.

**II. Đặc tính và sự phụ thuộc của mạch phản chiếu dòng điện**

Một mạch phản chiếu dòng điện có thể được đặc trưng bằng 3 thông số kỹ thuật sau:

**1. Tỷ lệ chuyển đổi dòng điện.**

Mạch phản chiếu sao chép dòng điện đầu vào của một thiết bị đang hoạt động sang đầu ra của thiết bị hoạt động khác. Mạch phản chiếu dòng điện lý tưởng là mạch khuếch đại đảo dòng điện lý tưởng có thể đảo chiều dòng điện. Do đó, đối với một bộ khuếch đại dòng điện lý tưởng, tỷ lệ truyền dòng điện là một thông số quan trọng.

**2. Điện trở đầu ra AC**

Điện trở có mối quan hệ điện áp-dòng điện theo định luật ohms. Do đó, điện trở đầu ra AC đóng một vai trò quan trọng trong sự ổn định của dòng điện đầu ra đối với sự thay đổi điện áp.

**3. Giảm điện áp**

Một mạch phản chiếu làm việc thích hợp có điện áp thấp trên đầu ra. Phạm vi điện áp trong đó Mạch phản chiếu dòng điện có thể hoạt động được gọi là phạm vi tuân thủ và điện áp được hỗ trợ tối thiểu đến tối đa trong phạm vi tuân thủ này được gọi là điện áp tuân thủ. Cần có điện áp tối thiểu để giữ cho transistor mở, do đó điện áp tối thiểu phụ thuộc vào thông số kỹ thuật của transistor.

**III. Hạn chế trong mạch phản chiếu dòng điện thực tế**

Mạch lý tưởng và mạch thực, hai cái này hoàn toàn khác nhau. Trong thực tế, không có gì gọi là hoàn hảo hay lý tưởng. Tuy nhiên, trước khi hiểu những hạn chế của mạch phản chiếu dòng điện đối với các ứng dụng thực tế, ta cần hiểu điện áp và nguồn dòng điện, cả lý thuyết và thực tế.

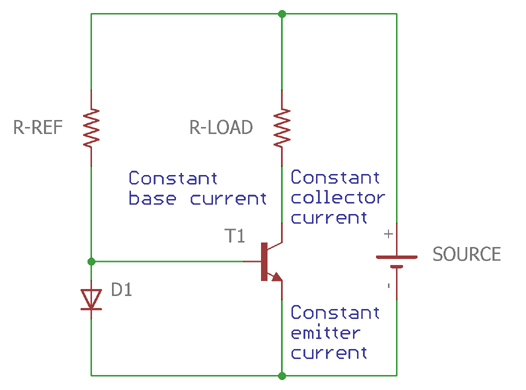
* Nguồn điện áp là thiết bị có khả năng cung cấp điện áp cố định và ổn định cho tải. Theo thuật ngữ lý tưởng, nguồn điện áp sẽ cung cấp một điện áp cố định liên tục mà không bị phụ thuộc vào dòng tải. Do đó, chúng ta có thể kết nối bất kỳ điện trở tải nào qua nguồn điện áp lý tưởng và luôn có được điện áp ổn định và cố định. Nhưng trong thực tế, các nguồn điện áp như pin, nguồn điện, v.v. không thể cung cấp dòng điện vô hạn hoặc vô hạn cho tải.
* Giống như nguồn điện áp lý tưởng, không phụ thuộc vào điện áp đầu cuối, nguồn dòng có thể cung cấp hoặc chấp nhận dòng điện. Nhưng trong thế giới thực, điện áp cũng ảnh hưởng đến quá trình phân phối dòng điện không đổi.

Trong trường hợp mạch phản chiếu dòng điện, nguồn điện áp và dòng điện là lý tưởng. Nhưng trong một ví dụ thực tế, chúng có tiếng ồn, khả năng chịu đựng, gợn sóng do đó điện áp đầu ra khác nhau. Tất cả điều này ảnh hưởng đến đầu ra nhân bản dòng điện.

Không chỉ điều này, mà về mặt lý thuyết trong các mạch phản chiếu dòng lý tưởng, trở kháng AC được chấp nhận là vô hạn, nhưng điều này không đúng trong trường hợp thực tế. Mạch phản chiếu dòng điện trong thực tế có trở kháng hữu hạn ảnh hưởng đến quá trình phân phối dòng điện. Ngoài ra, việc thực hiện mạch tạo ra điện dung ký sinh dẫn đến giới hạn tần số.

**IV. Mạch phản chiếu dòng điện sử dụng BJT**

Các transistor lưỡng cực được sử dụng rộng rãi để phản chiếu dòng điện. Mẹo đầu tiên để sử dụng transistor lưỡng cực làm mạch phản chiếu dòng điện là tạo điện áp hàm mũ thành bộ chuyển đổi dòng điện bằng transistor. Điều này được thực hiện bằng cách cung cấp một điện áp qua đường tiếp giáp base-emitter của BJT và dòng collector được lấy làm đầu ra. Trong cấu hình bộ chuyển đổi điện áp thành dòng điện này, phản hồi âm đơn giản qua transistor sẽ chuyển đổi các đặc tính của bộ chuyển đổi điện áp thành dòng điện thành một biến đổi điện áp thành dòng logarit ngược lại. Nói chung, phản hồi âm được thực hiện bằng cách nối base và collector của transistor.



Trước khi hiểu cách thức hoạt động của mạch, điều cần thiết là phải hiểu các đặc tính hoạt động của transistor. Trong hoạt động ở chế độ tích cực, Tỉ số giữa dòng collector và dòng base được gọi là β. Tỉ số giữa dòng emitter và dòng collector được gọi là ɑ. Mối quan hệ giữa hai điều này có thể được mô tả bằng cách sử dụng công thức:

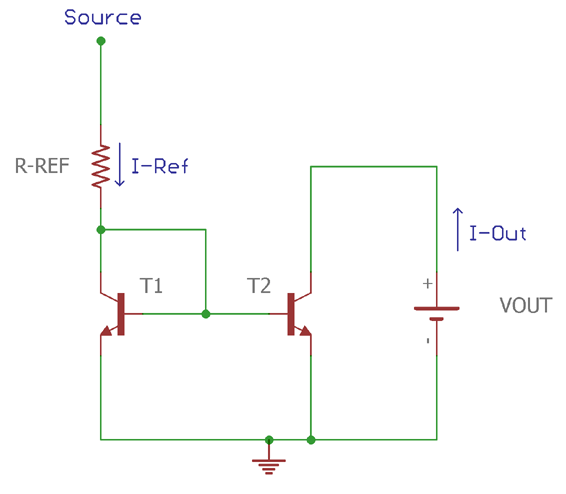
 ɑ = ​​β / (β + 1)

Do đó, điện áp cực base không đổi cung cấp dòng điện phát không đổi. Dòng điện phát không đổi này có thể được nhân với tỷ lệ ɑ không đổi. Hơn nữa cung cấp một dòng điện thu không đổi.

Trong hình trước, một diode phân cực thuận được sử dụng song song với điểm nối cực phát-gốc để cung cấp điện áp không đổi cho transistor. Điện áp trên bộ phát gốc là không đổi tùy thuộc vào dòng điện chạy qua diode. Tuy nhiên, dòng diode có thể được điều khiển bằng điện trở phân cực. Nếu dòng điện qua diode giảm bằng cách tăng giá trị của điện trở phân cực, điện áp giảm trên diode cũng sẽ giảm. Do hiệu ứng của giảm điện áp điểm nối base-emitter, dòng điện phát cũng sẽ giảm theo tỷ lệ tương tự. Cần nhớ một điều rằng ɑ và β của transistor là không đổi.

Bằng cách thay đổi dòng diode, dòng điện phát ra của transistor có thể được kiểm soát. Và đồng nghĩa với việc dòng điện collector của transistor cũng có thể thay đổi theo tỷ lệ tương tự. Theo quy tắc này, dòng phát của transistor có thể được đo vào collector của transistor. Do đó, điện trở phân cực có thể điều khiển dòng collector của transistor.

Trong mạch dưới đây, hai transistor được hiển thị được sử dụng để tạo ra mạch phản chiếu dòng điện. Transistor T1 và T2 cần phải giống nhau. Ngoài ra, hai transistor nên được đặt gần nhau để truyền nhiệt bằng nhau.



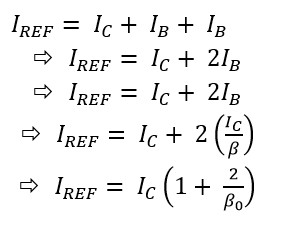
T1 và T2 được mắc song song với nhau. Do đó, hai transistor có cùng dòng điện. Vì vậy, cách tốt nhất để xác định dòng điện đầu ra là cộng dòng điện nút, nơi I REF đang chạy.

Theo định luật Kirchhoff, dòng điện tại collector T1 là :

Iref = Ic + IB1 + IB2

Do đó, khi cả hai transistor làm việc với sai lệch base-collector bằng không, dòng điện base bằng nhau

Dòng điện base của T1 (IB1) = Dòng điện base của T2 (IB2) = Tổng dòng điện base của nút (IB)



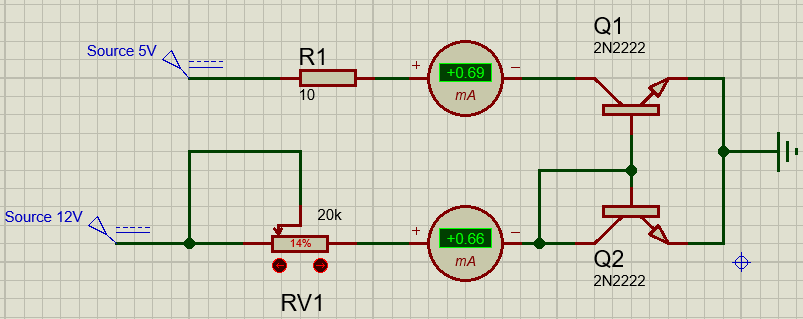
Điện trở đầu ra hữu hạn của transistor đầu ra có thể được tính bằng công thức dưới đây:

R = VA + VCE / IC

Các điện áp phù hợp, nơi V DG = 0 và hành vi nhân bản hiện nay vẫn hoạt động trong các điện áp đầu ra thấp nhất, có thể được tính toán như thế này:

VCV = VT\*ln ((IC / IS) +1)) Trong đó VT là điện áp nhiệt và IS là cường độ dòng điện.

Ví dụ về sao chép (gần đúng) dòng từ nhánh có điện trở lớn sang nhánh có điện trở nhỏ:



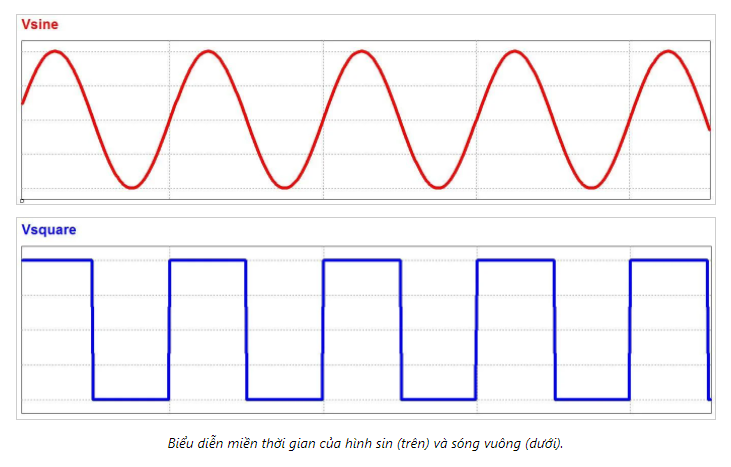
Có rất nhiều ứng dụng của mạch phản chiếu dòng điện trong lĩnh vực sản xuất mạch tích hợp. Nguồn dòng tiêu chuẩn được tạo ra bằng cách sử dụng mạch phản chiếu dòng điện. Bằng cách sử dụng kỹ thuật này, nhiều điểm tham chiếu có thể được tạo ra từ một nguồn duy nhất. Do đó, việc thay đổi một điểm tham chiếu cũng làm thay đổi nguồn dòng qua các phần khác nhau của mạch.

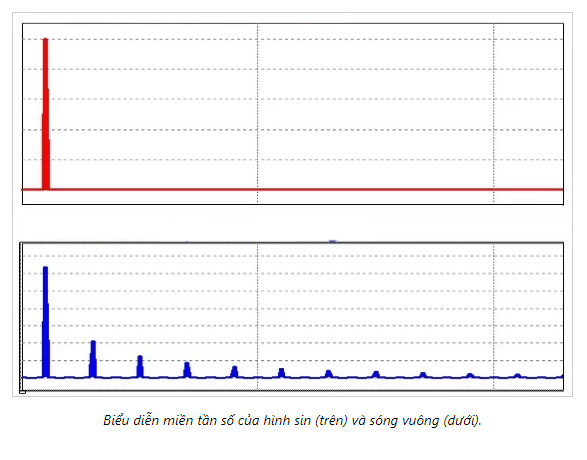
**Bộ lọc thông thấp là gì?**

Kiến thức cơ bản về bộ lọc RC

### **Miền thời gian và miền tần số**

Khi bạn nhìn vào một tín hiệu điện trên máy hiện sóng, bạn sẽ thấy một đường biểu diễn sự thay đổi của điện áp theo thời gian. Tại bất kỳ thời điểm cụ thể nào, tín hiệu chỉ có một giá trị điện áp. Những gì bạn thấy trên máy hiện sóng là **biểu diễn miền thời gian** của tín hiệu. Tuy nhiên nó có 1 hạn chế đó là không biểu diễn nội dung tần số của tín hiệu.





### **Bộ lọc là gì?**

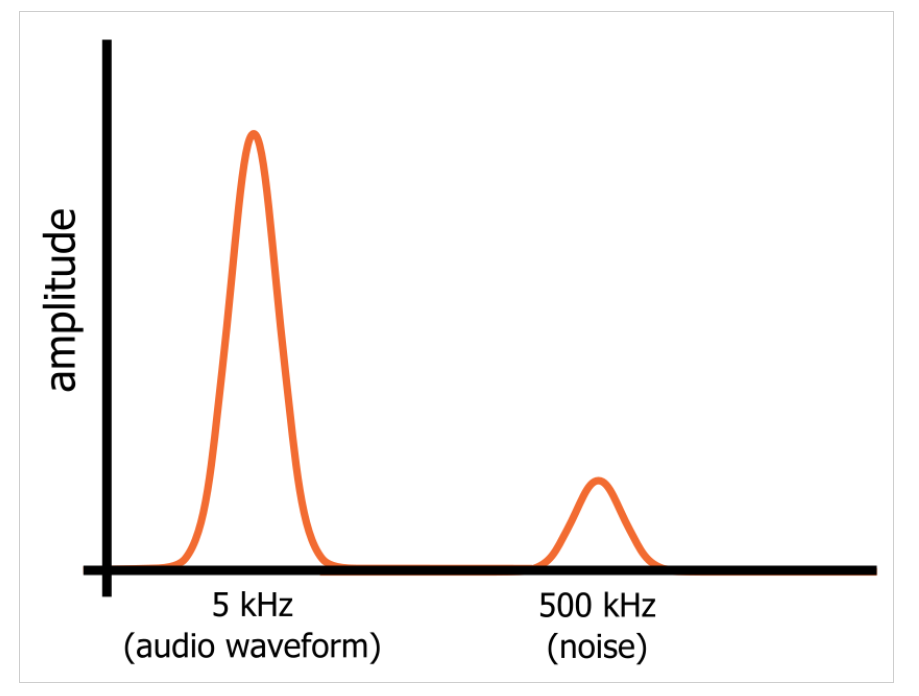
Bộ lọc là một mạch loại bỏ hoặc "lọc ra" một dải tần số được chỉ định. Nói cách khác, nó phân tách phổ của tín hiệu thành các thành phần tần số sẽ được **đi qua** và các thành phần tần số sẽ **bị chặn.**

Nếu bạn không có nhiều kinh nghiệm về phân tích miền tần số, bạn vẫn có thể không chắc chắn về những thành phần tần số này là gì và chúng cùng tồn tại như thế nào trong một tín hiệu không thể có nhiều giá trị điện áp cùng một lúc. Hãy xem một ví dụ ngắn gọn sẽ giúp làm rõ khái niệm này.

Hãy tưởng tượng rằng chúng ta có một tín hiệu âm thanh bao gồm một sóng sin 5 kHz hoàn hảo. Chúng ta biết sóng hình sin trông như thế nào trong miền thời gian, và trong miền tần số, chúng ta sẽ không thấy gì ngoài tần số "tăng đột biến" ở 5 kHz. Bây giờ, giả sử rằng chúng ta kích hoạt một bộ dao động 500 kHz đưa nhiễu tần số cao vào tín hiệu âm thanh.

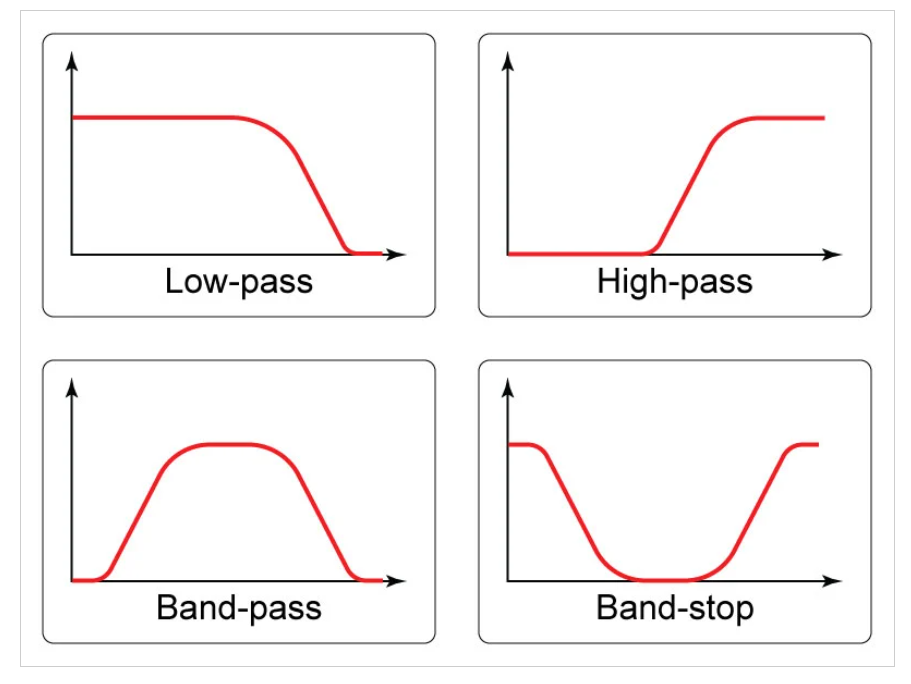
Tín hiệu như được thấy trên máy hiện sóng sẽ vẫn chỉ là một chuỗi điện áp, với một giá trị trên mỗi thời điểm, nhưng tín hiệu sẽ trông khác vì các biến thể miền thời gian của nó bây giờ phải phản ánh cả sóng hình sin 5 kHz và tần số cao, dao động tiếng ồn.

Tuy nhiên, trong miền tần số, sóng sin và nhiễu là các thành phần tần số riêng biệt có mặt đồng thời trong một tín hiệu này. Sóng hình sin và tiếng ồn chiếm các phần khác nhau của biểu diễn miền tần số của tín hiệu (như được hiển thị trong biểu đồ bên dưới) và điều này có nghĩa là chúng ta có thể lọc nhiễu bằng cách hướng tín hiệu qua một mạch chuyển tần số thấp và chặn tần số cao.

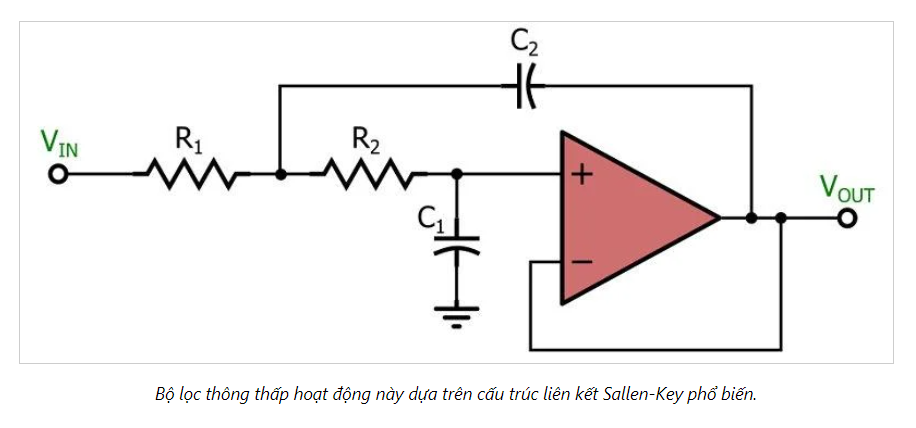


### **Các loại bộ lọc**

Bộ lọc có thể được xếp vào các danh mục rộng tương ứng với các đặc điểm chung của **đáp ứng tần số** của bộ lọc . Nếu một bộ lọc vượt qua các tần số thấp và chặn các tần số cao, nó được gọi là bộ lọc thông thấp. Nếu nó chặn tần số thấp và vượt qua tần số cao, nó là bộ lọc thông cao. Ngoài ra còn có các bộ lọc thông dải, chỉ vượt qua một dải tần số tương đối hẹp và các bộ lọc dừng dải, chỉ chặn một dải tần số tương đối hẹp.

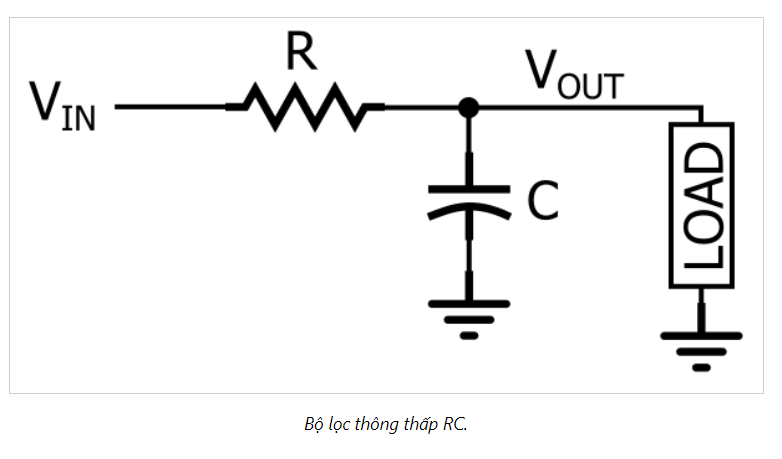


Bộ lọc cũng có thể được phân loại theo các loại thành phần được sử dụng để thực hiện mạch. Bộ lọc thụ động sử dụng điện trở, tụ điện và cuộn cảm. Các thành phần này không có khả năng cung cấp khả năng khuếch đại, và do đó một bộ lọc thụ động chỉ có thể duy trì hoặc giảm biên độ của tín hiệu đầu vào. Mặt khác, bộ lọc tích cực có thể vừa lọc tín hiệu vừa áp dụng độ lợi, bởi vì nó bao gồm một thành phần tích cực như bóng bán dẫn hoặc bộ khuếch đại hoạt động.



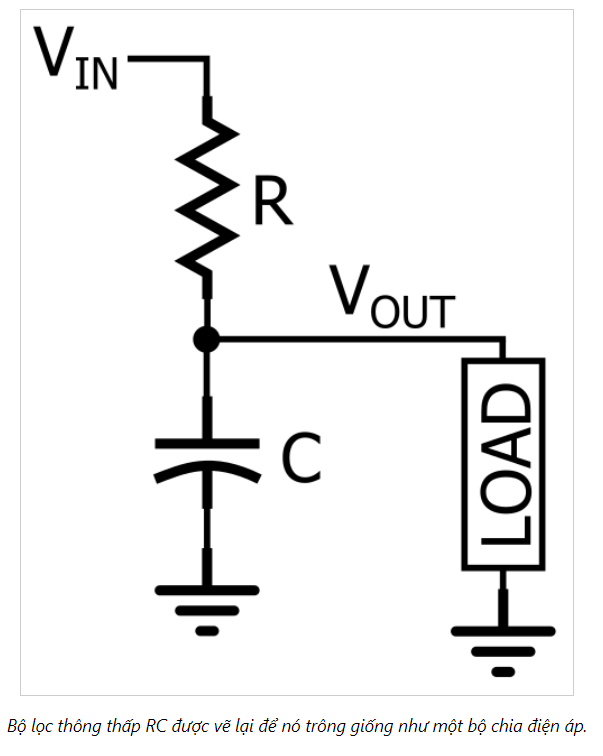
### **Bộ lọc thông thấp RC**

Để tạo bộ lọc thông thấp thụ động, chúng ta cần kết hợp phần tử điện trở với phần tử phản kháng. Nói cách khác, chúng ta cần một mạch điện bao gồm một điện trở và một tụ điện hoặc một cuộn cảm. Về lý thuyết, cấu trúc liên kết thông thấp điện trở-cuộn cảm (RL) tương đương, về khả năng lọc, với cấu trúc liên kết thông thấp điện trở-tụ điện (RC). Tuy nhiên, trong thực tế, phiên bản tụ điện trở phổ biến hơn nhiều, và do đó, phần còn lại của bài viết này sẽ tập trung vào bộ lọc thông thấp RC.



Như bạn có thể thấy trong sơ đồ, đáp ứng thông thấp RC được tạo ra bằng cách đặt một điện trở mắc nối tiếp với đường dẫn tín hiệu và một tụ điện song song với tải. Trong sơ đồ, tải là một thành phần đơn lẻ, nhưng trong một mạch thực, nó có thể phức tạp hơn nhiều, chẳng hạn như bộ chuyển đổi tương tự-kỹ thuật số, bộ khuếch đại hoặc giai đoạn đầu vào của máy hiện sóng mà bạn đang sử dụng để đo phản hồi của bộ lọc.

Chúng ta có thể phân tích trực quan hoạt động lọc của cấu trúc liên kết thông thấp RC nếu chúng ta nhận ra rằng điện trở và tụ điện tạo thành một bộ phân áp phụ thuộc vào tần số.



Khi tần số (f) của tín hiệu đầu vào thấp, trở kháng của tụ điện cao so với trở kháng của điện trở.

Do đó, hầu hết điện áp đầu vào bị rơi trên tụ điện C (và trên tải, song song với tụ điện). Khi tần số đầu vào cao, trở kháng của tụ điện sẽ thấp so với trở kháng của điện trở R, có nghĩa là điện áp rơi trên điện trở nhiều hơn và ít được chuyển đến tải hơn. Do đó, tần số thấp được thông qua và tần số cao bị chặn lại.

Giải thích định tính về chức năng thông thấp RC này là một bước quan trọng đầu tiên, nhưng nó không hữu ích lắm khi chúng ta thực sự cần thiết kế một mạch, bởi vì các thuật ngữ “tần số cao” và “tần số thấp” là vô cùng mơ hồ. Các kỹ sư cần tạo ra các mạch vượt qua và chặn các tần số cụ thể. Ví dụ, trong hệ thống âm thanh được mô tả ở trên, chúng tôi muốn bảo toàn tín hiệu 5 kHz và triệt tiêu tín hiệu 500 kHz. Điều này có nghĩa là chúng ta cần một bộ lọc chuyển từ chuyển sang chặn ở đâu đó trong khoảng từ 5 kHz đến 500 kHz.

**Giao thức Modbus trong Công nghiệp**

Giới thiệu về Modbus

Khái niệm:

Modbus là 1 giao thức truyền thông kỹ thuật số lâu đời và được sử dụng phổ biến bậc nhất trong công nghiệp dùng để trao đổi thông tin data dữ liệu giữa các thiết bị điều khiển và giám sát công nghiệp.

Hoạt động theo mô hình Master/Slave. Một Master với tối thiểu một Slave. Master sẽ hỏi tất cả Slave và nếu Slave nào trùng địa chỉ được hỏi thì sẽ phản hồi lại về cho Master.

Thiết bị Master thường là một PLC, PC…

Thiết bị Slave là bất kỳ thiết bị ngoại vi nào (bộ chuyển I/O, Valve, Driver hoặc thiết bị đo lường khác…) xử lý thông tin và gửi Output của nó đến thiết bị Master bằng Modbus.

Modbus chỉ là 1 chuẩn bản tin do đó nó cần hoạt động trên 1 đường truyền vật lý có thể là UART, RS232, RS422, RS485 hay cổng Ethernet…

Tùy thuộc vào đường truyền vật lý sẽ quyết định xem mô hình Master làm việc với bao nhiêu Slave. Với UART hay RS232 sẽ là mô hình 1 Master 1 Slave. Còn đối với RS422 hay RS485 sẽ hỗ trợ được nhiều Slave hơn.

Chúng ta có các chuẩn Modbus được sử dụng rộng rãi nhất là

-       Modbus ASCII

-       Modbus RTU

-       Modbus TCP

Mô hình 1 (Modbus RTU)

Mô hình 2 (Modbus ASCII)

Mô hình 3 (Modbus TCP)

Tuy nhiên sử dụng phổ biến trong nghành nhúng thì sẽ là ModbusRTU.

Dưới đây chính là khung mô tả 1 bản tin ModbusRTU. Các tin nhắn sẽ được phân cách nhau bằng một khoảng thời gian tối thiểu là 3.5 ký tự (khoảng thời gian để truyền 3.5 ký tự ~ 28bit).



Các trường trong bản tin:

**Trường Address của Slave:** có giá trị từ 1 đến 247 và có độ lớn là 1 byte (8 bit).

**Trường Function:**

[01 READ COIL STATUS](https://www.modbustools.com/modbus.html#Function01)

Đọc trạng thái ON/OFF của n bit Coil trên Slave (chúng ta có thể hiểu đơn giản 1 bit Coil này tương ứng như 1 Relay). Bit Coil này thì vừa có thể đọc vừa có thể ghi được.

[02 READ INPUT STATUS](https://www.modbustools.com/modbus.html#function02)

Đọc trạng thái ON/OFF của n bit Input trên Slave. Khác với Coil thì bit Input này chỉ có thể đọc được chứ không ghi được.

[03 READ HOLDING REGISTERS](https://www.modbustools.com/modbus.html#function03)

Đọc giá trị n thanh ghi Holding trên Slave. Mỗi thanh ghi gồm 16 bits để chứa dữ liệu. (thường là nơi lưu dữ liệu cảm biến). Có thể vừa đọc vừa ghi dữ liệu vào thanh ghi Holding này.

[04 READ INPUT REGISTERS](https://www.modbustools.com/modbus.html#function04)

Đọc giá trị thanh ghi Input trên Slave. Mỗi thanh ghi này gồm 16 bit chứa dữ liệu. Khác với thanh ghi Holding thanh ghi này chỉ đọc được chứ không ghi dữ liệu vào được.

[05 WRITE SINGLE COIL](https://www.modbustools.com/modbus.html#function05)

Ghi trạng thái ON/OFF vào 1 bit Coil trên Slave (hình dung giống việc ta ghi để điều khiển trạng thái 1 Relay vậy).

[06 WRITE SINGLE REGISTER](https://www.modbustools.com/modbus.html#function06)

Ghi giá trị vào trong 1 thanh ghi Holding của Slave.

[15 WRITE MULTIPLE COILS](https://www.modbustools.com/modbus.html#function15)

Ghi trạng thái ON/OFF vào n bit Coil trên Slave (hình dung giống việc ta ghi để điều khiển trạng thái n Relay vậy).

[16 WRITE MULTIPLE REGISTERS](https://www.modbustools.com/modbus.html#function16)

Ghi giá trị vào trong n thanh ghi Holding của Slave.

**Trường data** sẽ chứa các thông tin như là địa chỉ thanh ghi bắt đầu, số lượng thanh ghi muốn thao tác…

**Trường CRC** kiểm tra lỗi truyền nhận của bản tin đã được cấp sẵn API để sử dụng.

Địa chỉ các thanh ghi của Slave cũng đã được quy định chung theo tiêu chuẩn của Modbus. Tham khảo hình dưới:

Các bạn cũng có thể tham khảo chi tiết hơn nội dung tại link: <https://www.modbustools.com/modbus.html>

Chúng ta cũng có các phần mềm mô phỏng để test code Modbus với các thiết bị nhúng có thể kể tới như Modbus Poll hoặc EasyModbus…

**Điện trở kéo lên và điện trở kéo xuống (PULL UP & PULL DOWN)**

Điện trở kéo lên và kéo xuống là gì và tại sao chúng ta cần chúng? Bài viết này sẽ giải thích tất cả vướng mắc của các bạn.

Vị trí điện trở kéo lên và kéo xuống

Điện trở kéo lên và kéo xuống là gì và tại sao chúng ta cần chúng?

Nếu chúng ta sử dụng mạch điện tử số, các chân của IC số là 0 (mức logic thấp ) hoặc 1 (mức logic cao). Trong một số trường hợp, chúng ta cần thay đổi trạng thái từ 0 sang 1 hoặc từ 1 sang 0.

Bây giờ, nếu chúng ta thay thế mức cao và mức thấp bằng giá trị điện áp thực thì mức cao là mức HIGH (giả sử là 5V) và mức thấp là mức LOW (0V).

Một điện trở kéo lên được sử dụng để đặt trạng thái mặc định của chân IC là ở mức logic cao (5V) và một điện trở kéo xuống làm hoàn toàn ngược lại, nó làm cho trạng thái mặc định của chân IC là ở mức logic thấp (0V).

Nhưng tại sao chúng ta cần những điện trở đó thay vì chúng ta có thể kết nối các chân logic IC số trực tiếp với điện áp mức cao hoặc với đất như hình ảnh dưới đây?

Vì mạch kỹ thuật số hoạt động ở dòng điện thấp, kết nối các chân logic trực tiếp với điện áp nguồn hoặc đất không phải là một lựa chọn tốt. Vì kết nối trực tiếp cuối cùng làm tăng dòng điện giống như ngắn mạch và có thể làm hỏng mạch điện tử . Để kiểm soát dòng điện, chúng ta cần những điện trở kéo xuống hoặc kéo lên. Một điện trở kéo lên cho phép dòng điện được kiểm soát từ nguồn điện áp cung cấp đến các chân đầu vào IC số, trong đó điện trở kéo xuống có thể kiểm soát hiệu quả dòng điện từ các chân IC số xuống GND.

Đồng thời cả hai điện trở, điện trở kéo xuống và kéo lên đều giữ trạng thái chân IC số ở mức thấp hoặc mức cao.

Tính toán các giá trị thực tế cho điện trở kéo lên và kéo xuống

Bây giờ, khi chúng ta đã biết cách sử dụng điện trở kéo lên và kéo xuống, câu hỏi đặt ra là giá trị của những điện trở đó sẽ là bao nhiêu? Mặc dù, trong nhiều mạch mức logic kỹ thuật số, chúng ta có thể thấy các điện trở kéo lên hoặc kéo xuống nằm trong khoảng từ 2k đến 4,7k. Nhưng giá trị thực sẽ là bao nhiêu?

Để hiểu được điều này, chúng ta cần biết điện áp logic là gì? Điện áp bao nhiêu được gọi là mức logic thấp và Bao nhiêu được gọi là mức logic cao? Đối với các mức logic khác nhau, các bộ vi điều khiển khác nhau sử dụng một phạm vi khác nhau cho mức logic cao và logic thấp. Nếu chúng ta xem xét một đầu vào Mức logic Transistor-Transistor (TTL), biểu đồ dưới đây sẽ hiển thị điện áp logic tối thiểu để xác định mức Logic cao và điện áp logic tối đa để phát hiện mức logic là 0 hoặc Thấp.

Như chúng ta có thể thấy, đối với logic TTL, điện áp tối đa cho logic thấp là 0,8V. Vì vậy, nếu chúng ta cung cấp ít hơn 0,8V thì mức logic sẽ được chấp nhận là mức thấp. Mặt khác, nếu chúng ta cung cấp nhiều hơn 2V đến mức tối đa 5,25V thì mức logic sẽ được chấp nhận là mức cao. Nhưng ở mức 0,8V đến 2V, nó là vùng trống, ở điện áp đó không thể đảm bảo rằng logic sẽ được chấp nhận là cao hay thấp. Vì vậy, để đảm bảo an toàn, Trong kiến trúc TTL, chúng ta chấp nhận 0V đến 0,8V là Thấp và 2V đến 5V là Cao, đảm bảo rằng Thấp và Cao sẽ được các chip logic ở điện áp biên đó nhận dạng.

Để xác định giá trị, công thức là định luật Ohms đơn giản. Theo định luật ohms, công thức là:

***V = I x R***

***R = V/I***

Trong trường hợp điện trở kéo lên, V sẽ là điện áp nguồn - điện áp tối thiểu được chấp nhận là Cao.

Và dòng điện sẽ là dòng điện tối đa bị chìm bởi các chân logic. Vì thế:

***Rpull-up = (Vsupply – VH(min)) / Isink***

Trong đó Vsupply là điện áp cung cấp, VH(min) là điện áp tối thiểu được chấp nhận là mức cao và Isink là dòng điện tối đa vào chân IC số.

Ví dụ:

Giả sử chúng ta có một mạch logic trong đó Vsupply là 3,3V và VH(min) là 3V, và chúng ta có thể Isink tối đa là 30uA, sau đó chúng ta có thể chọn điện trở kéo lên bằng công thức như ảnh. Điều tương tự cũng áp dụng cho điện trở kéo xuống.

***Rpull-up = (VL(max) – 0) / Isource***

Trong đó VL(max) điện áp tối đa được chấp nhận là mức logic thấp và Isource là dòng điện tối đa được lấy bởi chân IC số.

Bây giờ, Nếu chúng ta xem xét cùng một ví dụ đã nêu ở trên, trong đó VL(max) là 1V và có thể tạo ra dòng điện lên đến 200uA thì điện trở kéo xuống sẽ là:

*Sau bài viết này hi vọng các bạn có thể áp dụng vào các kiến thức thực tế. Chúc các bạn thành công.*

**Giao tiếp I2C**

Giao tiếp I2C trong lập trình nhúng

I. Khái niệm

* I2C ( Inter – Integrated Circuit) là 1 giao thức giao tiếp nối tiếp đồng bộ được phát triển bởi Philips Semiconductors, sử dụng để truyền nhận dữ liệu giữa các IC với nhau chỉ sử dụng hai đường truyền tín hiệu.
* Các bit dữ liệu sẽ được truyền từng bit một theo các khoảng thời gian đều đặn được thiết lập bởi 1 tín hiệu đồng hồ.
* Bus I2C thường được sử dụng để giao tiếp ngoại vi cho rất nhiều loại IC khác nhau như các loại vi điều khiển, cảm biến, EEPROM, … .



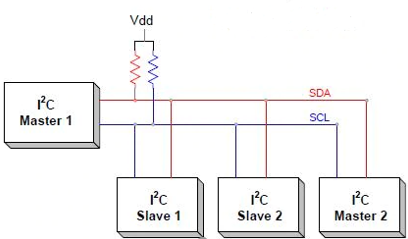
II. Giao tiếp I2C hoạt động như thế nào ??

***Cấu tạo :***

* I2C sử dụng 2 đường truyền tín hiệu:

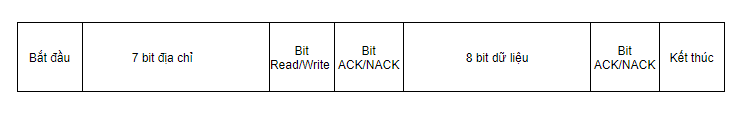
SCL - Serial Clock Line : Tạo xung nhịp đồng hồ do Master phát đi

SDA - Serial Data Line : Đường truyền nhận dữ liệu.



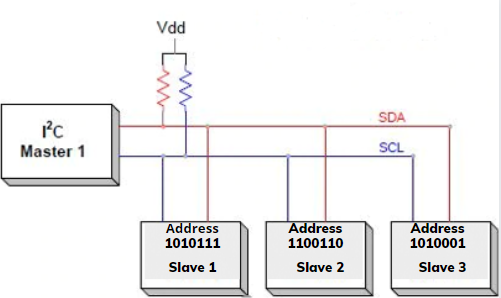
* Giao tiếp I2C bao gồm quá trình truyền nhận dữ liệu giữa các thiết bị chủ tớ, hay Master - Slave.
* Thiết bị Master là 1 vi điều khiển, nó có nhiệm vụ điều khiển đường tín hiệu SCL và gửi nhận dữ liệu hay lệnh thông qua đường SDA đến các thiết bị khác.
* Các thiết bị nhận các dữ liệu lệnh và tín hiệu từ thiết bị Master được gọi là các thiết bị Slave. Các thiết bị Slave thường là các IC, hoặc thậm chí là vi điều khiển.
* Master và Slave được kết nối với nhau như hình trên. Hai đường bus SCL và SDA đều hoạt động ở chế độ Open Drain, nghĩa là bất cứ thiết bị nào kết nối với mạng I2C này cũng chỉ có thể kéo 2 đường bus này xuống mức thấp (LOW), nhưng lại không thể kéo được lên mức cao. Vì để tránh trường hợp bus vừa bị 1 thiết bị kéo lên mức cao vừa bị 1 thiết bị khác kéo xuống mức thấp gây hiện tượng ngắn mạch. Do đó cần có 1 điện trờ ( từ 1 – 4,7 kΩ) để giữ mặc định ở mức cao.

***Khung truyền I2C :***



o  *Khối bit địa chỉ :*

Thông thường quá trình truyền nhận sẽ diễn ra với rất nhiều thiết bị, IC với nhau. Do đó để phân biệt các thiết bị này, chúng sẽ được gắn 1 địa chỉ vật lý 7 bit cố định.



o  *Bit Read/Write:*

Bit này dùng để xác định quá trình là truyền hay nhận dữ liệu từ thiết bị Master. Nếu Master gửi dữ liệu đi thì ứng với bit này bằng ‘0’, và ngược lại, nhận dữ liệu khi bit này bằng ‘1’.

o  *Bit ACK/NACK:*

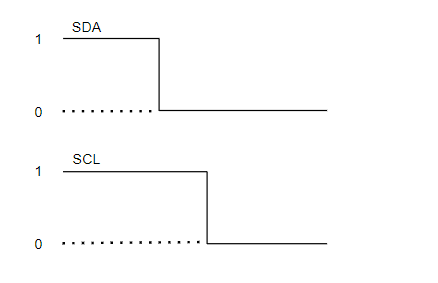
Viết tắt của Acknowledged / Not Acknowledged. Dùng để so sánh bit địa chỉ vật lý của thiết bị so với địa chỉ được gửi tới. Nếu trùng thì Slave sẽ được đặt bằng ‘0’ và ngược lại, nếu không thì mặc định bằng ‘1’.

o  *Khối bit dữ liệu:*

Gồm 8 bit và được thiết lập bởi thiết bị gửi truyền đến thiết bị nhân. Sau khi các bit này được gửi đi, lập tức 1 bit ACK/NACK được gửi ngay theo sau để xác nhận rằng thiết bị nhận đã nhận được dữ liệu thành công hay chưa. Nếu nhận thành công thì bit ACK/NACK được set bằng ‘0’ và ngược lại.

***Quá trình truyền nhận dữ liệu:***

* Bắt đầu: Thiết bị Master sẽ gửi đi 1 xung Start bằng cách kéo lần lượt các đường SDA, SCL từ mức 1 xuống 0.



* Tiếp theo đó, Master gửi đi 7 bit địa chỉ tới Slave muốn giao tiếp cùng với bit Read/Write.
* Slave sẽ so sánh địa chỉ vật lý với địa chỉ vừa được gửi tới. Nếu trùng khớp, Slave sẽ xác nhận bằng cách kéo đường SDA xuống 0 và set bit ACK/NACK bằng ‘0’. Nếu không trùng khớp thì SDA và bit ACK/NACK đều mặc định bằng ‘1’.
* Thiết bị Master sẽ gửi hoặc nhận khung bit dữ liệu. Nếu Master gửi đến Slave thì bit Read/Write ở mức 0. Ngược lại nếu nhận thì bit này ở mức 1.
* Nếu như khung dữ liệu đã được truyền đi thành công, bit ACK/NACK được set thành mức 0 để báo hiệu cho Master tiếp tục.
* Sau khi tất cả dữ liệu đã được gửi đến Slave thành công, Master sẽ phát 1 tín hiệu Stop để báo cho các Slave biết quá trình truyền đã kết thúc bằng các chuyển lần lượt SCL, SDA từ mức 0 lên mức 1.

﻿

***Các chế độ hoạt động của I2C:***

* Chế độ chuẩn (standard mode) với tốc độ 100 kBit/s.
* Chế độ tốc độ thấp (low speed mode) với tốc độ 10 kBit/s.

Ngoài ra, khác với giao tiếp SPI chỉ có thể có 1 Master, giao tiếp I2C cho phép chế độ truyền nhận dữ liệu giữa nhiều thiết bị Master khác nhau với thiết bị Slave. Tuy nhiên quá trình này có hơi phức tạp vì thiết bị Slave có thể nhận 1 lúc nhiều khung dữ liệu từ các thiết bị Master khác nhau, điều đó đôi khi dẫn đến xung đột hoặc sai sót dữ liệu nhận được.

Để tránh điều đó, khi làm việc ở chế độ này, mỗi thiết bị Master cần phát hiện xem đường SDA đang ở trạng thái nào.

Nếu SDA ở mức 0, nghĩa là đang có 1 thiết bị Master khác đang có quyền điều khiển và phải chờ đến khi truyền xong.

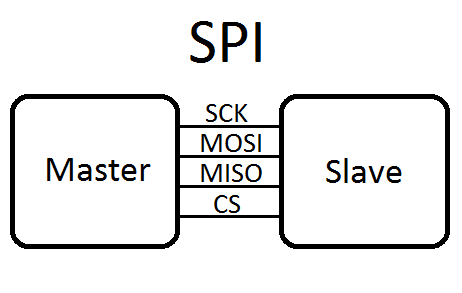
Ngược lại nếu SDA ở mức 1, nghĩa là đường truyền SDA đã an toàn và có sử dụng

**Giao tiếp SPI**

Giao tiếp SPI trong lập trình nhúng

I. Lý thuyết

* SPI – Serial Peripheral Interface – hay còn gọi là giao diện ngoại vi nối tiếp, được phát triển bởi hãng Motorola.
* Chuẩn đồng bộ nối truyền dữ liệu ở chế độ full - duplex (hay gọi là "song công toàn phần". Nghĩa là tại 1 thời điểm có thể xảy ra đồng thời quá trình truyền và nhận.
* Là giao tiếp đồng bộ, bất cứ quá trình nào cũng đều được đồng bộ với xung clock sinh ra bởi thiết bị Master  ► Không cần phải lo lắng về tốc độ truyền dữ liệu.
* SPI thường được sử dụng giao tiếp với bộ nhớ EEPROM, RTC (Đồng hồ thời gian thực), IC âm thanh, các loại cảm biến như nhiệt độ và áp suất, thẻ nhớ như MMC hoặc thẻ SD hoặc thậm chí các bộ vi điều khiển khác.

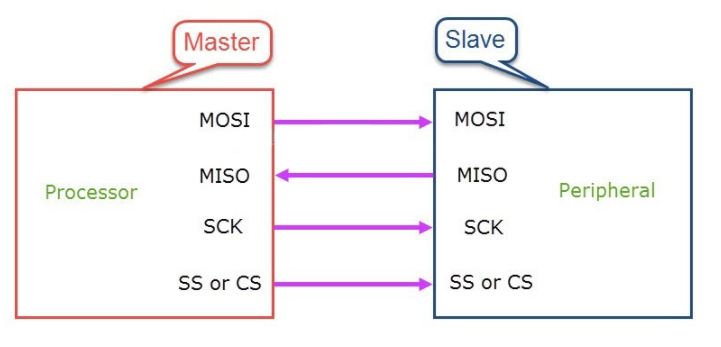


II. Giao tiếp I2C hoạt động như thế nào ?

***Cấu tạo:***

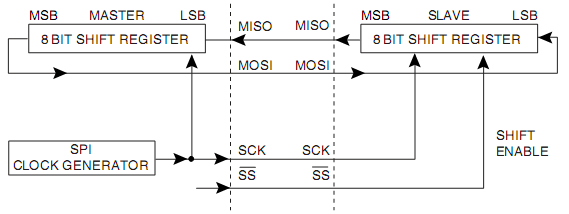
Sử dụng 4 đường giao tiếp nên đôi khi được gọi là chuẩn truyền thông “ 4 dây”. 4 đường đó là :

* *SCK* (Serial Clock): Thiết bị Master tạo xung tín hiệu SCK và cung cấp cho Slave. Xung này có chức năng giữ nhịp cho giao tiếp SPI. Mỗi nhịp trên chân SCK báo 1 bit dữ liệu đến hoặc đi → Quá trình ít bị lỗi và tốc độ truyền cao.
* *MISO* (Master Input Slave Output): Tín hiệu tạo bởi thiết bị Slave và nhận bởi thiết bị Master. Đường MISO phải được kết nối giữa thiết bị Master và Slave.
* *MOSI* (Master Output Slave Input): Tín hiệu tạo bởi thiết bị Master và nhận bởi thiết bị Slave. Đường MOSI phải được kết nối giữa thiết bị Master và Slave.
* *SS* (Slave Select): Chọn thiết bị Slave cụ thể để giao tiếp. Để chọn Slave giao tiếp thiết bị Master chủ động kéo đường SS tương ứng xuống mức 0 (Low). Chân này đôi khi còn được gọi là CS (Chip Select). Chân SS của vi điều khiển (Master) có thể được người dùng tạo bằng cách cấu hình 1 chân GPIO bất kỳ chế độ Output.

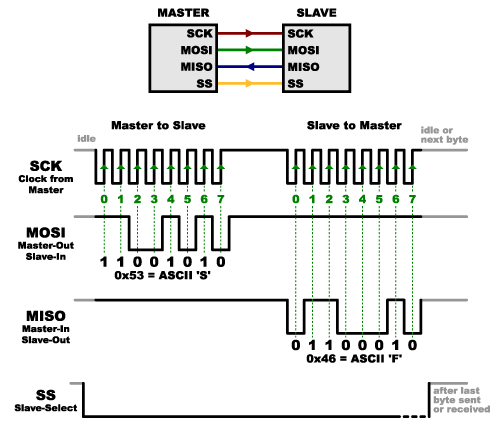


***Khung truyền SPI:***

* Mỗi chip Master hay Slave đều có một thanh ghi dữ liệu 8 bits.
* Quá trình truyền nhận giữa Master và Slave xảy ra đồng thời sau 8 chu kỳ đồng hồ, một byte dữ liệu được truyền theo cả 2 hướng
* Quá trình trao đổi dữ liệu bắt đầu khi Master tạo 1 xung clock từ bộ tạo xung nhịp (Clock Generator) và kéo đường SS của Slave mà nó truyền dữ liệu xuống mức Low.
* Cứ 1 xung clock, Master sẽ gửi đi 1 bit từ thanh ghi dịch (Shift Register) của nó đến thanh ghi dịch của Slave thông qua đường MOSI. Đồng thời Slave cũng gửi lại 1 bit đến cho Master qua đường MISO.Như vậy sau 8 chu kỳ clock thì hoàn tất việc truyền và nhận 1 byte dữ liệu.
* Dữ liệu của 2 thanh ghi được trao đổi với nhau nên tốc độ trao đổi diễn ra nhanh và hiệu quả.
* Lưu ý: Trong giao tiếp SPI, chỉ có thể có 1 Master nhưng có thể 1 hoặc nhiều Slave cùng lúc. Ở trạng thái nghỉ, chân SS của các Slave ở mức 1, muốn giao tiếp với Slave nào thì ta chỉ việc kéo chân SS của Slave đó xuống mức 0.

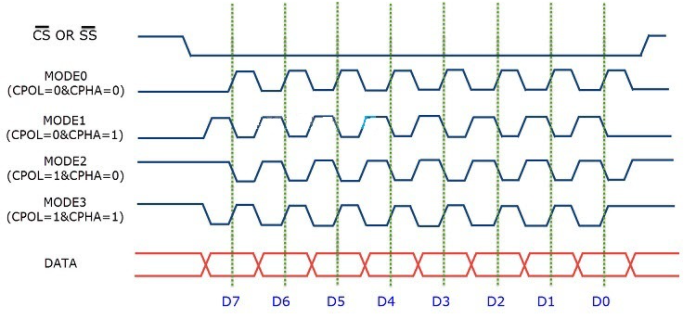


Ví dụ : Dạng tín hiệu xung các chân khi truyền byte đầu tiên 0x53 từ master sang slave và byte thứ 2 là sẽ truyền sang 0xFF để nhận về byte 0x46 từ slave.

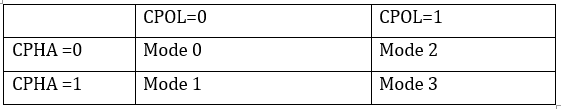


***Chế độ hoạt động:***

* SPI có 4 chế độ hoạt động phụ thuộc vào cực của xung giữ (Clock Polarity – CPOL) và pha (Phase - CPHA).
* CPOL dùng để chỉ trạng thái của chân SCK ở trạng thái nghỉ. Chân SCK giữ ở mức cao khi CPOL=1 hoặc mức thấp khi CPOL=0.
* CPHA dùng để chỉ các mà dữ liệu được lấy mẫu theo xung. Dữ liệu sẽ được lấy ở cạnh lên của SCK khi CPHA=0 hoặc cạnh xuống khi CPHA=1.



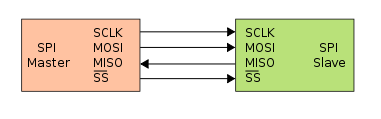
* Mode 0 (mặc định) – xung nhịp của đồng hồ ở mức thấp (CPOL = 0) và dữ liệu được lấy mẫu khi chuyển từ thấp sang cao (cạnh lên) (CPHA = 0).
* Mode 1 - xung nhịp của đồng hồ ở mức thấp (CPOL = 0) và dữ liệu được lấy mẫu khi chuyển từ cao sang thấp (cạnh xuống) (CPHA = 1).
* Mode 2 - xung nhịp của đồng hồ ở mức cao (CPOL = 1) và dữ liệu được lấy mẫu khi chuyển từ cao sang thấp (cạnh lên) (CPHA = 0).
* Mode 3 - xung nhịp của đồng hồ ở mức cao (CPOL = 1) và dữ liệu được lấy mẫu khi chuyển từ thấp sang cao (cạnh xuông) (CPHA = 1).



Lưu ý: Khi giao tiếp SPI giữa vi điều khiển và các thiết bị ngoại vi khác như IC, cảm biến thì 2 bên bắt buộc hoạt động cùng Mode, nếu không dữ liệu truyền nhận có thể bị đọc sai.

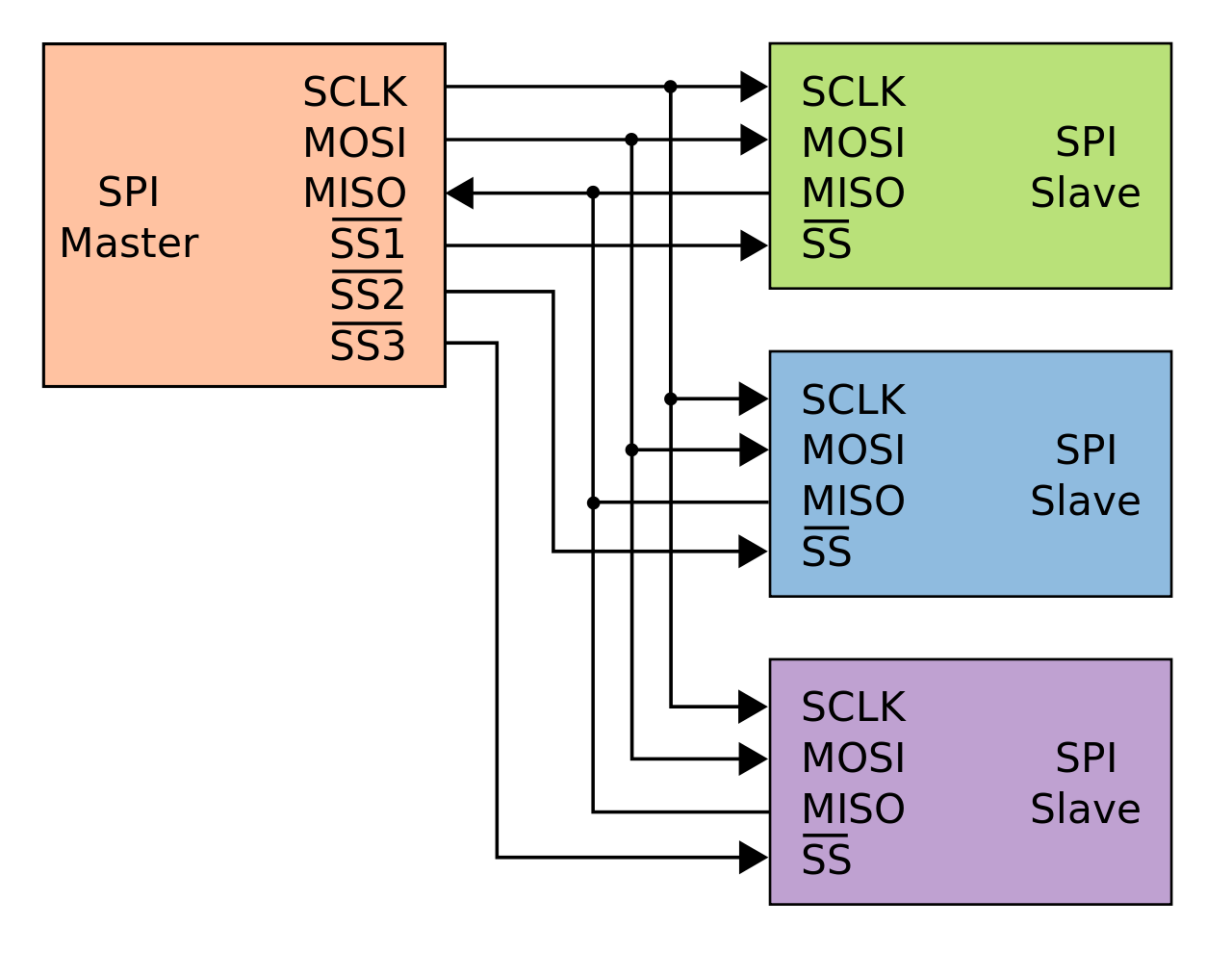
***Các sơ đồ kết nối giao tiếp SPI:***

* 1 thiết bị Master và 1 thiết bị Slave



* 1 thiết bị Master và nhiều thiết bị Slave ( chế độ độc lập - Independent):

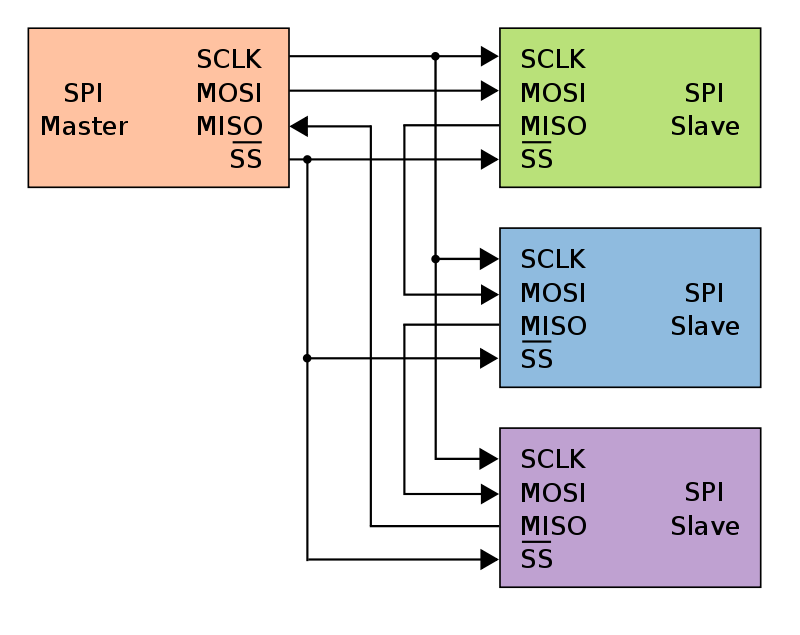
Ở chế độ này, mỗi thiết bị Slave kết nối với Master được quy định riêng bởi những chân SS khác nhau. Khi thiết bị Master muốn giao tiếp với Slave nào thì kéo chân SS tương ứng xuống mức 0, những chân SS còn lại giữ ở mức 1.



* 1 thiết bị Master và nhiều thiết bị Slave ( chế độ dây chuyền - Daisy):

Thử tưởng tượng có bạn cần kết nối SPI với nhiều thiết bị, nếu dùng cách trên thì sẽ tốn nhiều chân SS của vi điều khiển. Thay vào đó chúng ta có thể kết nối các thiết bị Slave theo kiểu dây chuyển như bên dưới mà chỉ cần 1 chân SS từ vi điều khiển. Chân MOSI của Slave này sẽ nối với MISO của Slave tiếp theo.

Dữ liệu gửi từ vi điều khiển ( hay thiết bị Master), đi vào Slave 1 bằng đường MOSI. Sau đó lại đi ra từ chân MISO của Slave 1, gửi tới chân MOSI của Slave 2, và cứ tiếp tục như vậy,.... Các bạn có thể thấy cách hoạt động này khác giống với các IC dịch.



**TÌM HIỂU MODULE NRF24L01 - GIAO THỨC Enhanced ShockBurst**

NRF24L01 - GIAO THỨC Enhanced ShockBurs

Chào các bạn, dòng chip nRF24L01 hẳn không xa lạ với mọi người. Tuy vậy chúng ta vẫn chưa sử dụng hết khả năng của nó. Ở bài post trước mình đã viết về cách biến module RF này thành 1 moudle Bluetooth rồi. Ở bài này chúng ta sẽ tìm hiểu về giao thức mà module RF này vẫn hay được sử dụng nhé.

Advanced ShockBurst (ESB) là 1 giao thức đơn giản hỗ trợ truyền nhận dữ liệu 2 chiều đã bao gồm việc đóng gói, nhận gói và tự động truyền lại gói khi xảy ra lỗi.

ESB protocol được nhúng sẵn trong phần cứng của nRF24Lxx từ hãng Nordic. Thư viện của hãng cho phép 2 series nổi bật nRF5x và nRF24Lxx có thể giao tiếp với nhau qua giao thức này.

**ĐẶC TÍNH NỔI BẬT**

1. Hỗ trợ kiểu mạng Star với 1 thiết bị nhận đồng thời 8 thiết bị phát.
2. Kích thước Payload của gói tin lên tới 32 bytes dữ liệu.
3. Truyền nhận 2 chiều.
4. Tự động kiểm tra và truyền lại gói tin bị lỗi.
5. Bộ đêm cho TX và RX độc lập cho mỗi đường truyền.

**QUÁ TRÌNH TRUYỀN**

1. Quá trình truyền bắt đầu từ thiết bị PTX và kết thúc khi nó nhận được gói tin ACK từ thiết bị PRX.
2. Nếu PTX không nhận được gói tin ACK trong lần truyền đầu tiên, nó sẽ gửi lại gói tin đến khi nhận được ACK. Số lần gửi lại tối đa có thể được cấu hình bằng Software.
3. Người dùng có thể cấu hình cho PTX gửi các bản tin không yêu cầu ACK, PRX khi nhận được gói tin này sẽ không gửi lại bản tin ACK phản hồi cho PTX.

**ỨNG DỤNG**

1. Tạo ra 1 hệ thống truyền nhận không dây giữa nhiều thiết bị theo mô hình mạng Star.
2. Kết hợp giữa 2 series nRF5x và nRF24L01 để chạy multi protocol Bluetooth và RF 2.4Ghz.

**TÀI LIỆU**

[https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF24-series](https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF24-series?fbclid=IwAR1zr5UK6SO8ZPxtGEEbte0I3J1u3hRew8mPzMv-NHnX9svVvbdpdAoRenk)

**Tại sao dùng delay trong ngắt lại bị treo?**

Author: Đặng Lợi

*Ví dụ với STM32:*

Hàm HAL\_Delay(): đây là hàm để delay theo đơn vị ms. Trong hàm này, VĐK sẽ gọi hàm GetTick để lấy giá trị đếm ban đầu và lưu vào biến tickstart, thời gian cần delay sẽ được lưu vào biến wait. Ở vòng lặp while, VĐK tiếp tục gọi hàm GetTick để lấy giá trị đếm hiện tại, nếu giá trị đếm hiện tại trừ giá trị đếm ban đầu không nhỏ hơn giá trị cần delay (ms) thì điều kiện vòng lặp while không còn thỏa mãn, vi điều khiển đã thực hiện xong hàm delay (ms).

\_\_weak void HAL\_Delay(uint32\_t Delay)

{

uint32\_t tickstart = HAL\_GetTick();

uint32\_t wait = Delay;

/\* Add a freq to guarantee minimum wait \*/

**if** (wait < HAL\_MAX\_DELAY)

{

wait += (uint32\_t)(uwTickFreq);

}

**while** ((HAL\_GetTick() - tickstart) < wait);

}

Hàm HAL\_GetTick(): hàm này thực hiện trả về giá trị đếm hiện tại của System Tick Timer.

\_\_weak **uint32\_t** HAL\_GetTick**(void)**

{

**return** uwTick;

}

Hàm HAL\_IncTick(): Trong hàm này biến uwTick sẽ được mặc định tăng lên một đơn bị khi hàm được gọi. (uwTickFreq =1)

\_\_weak **void** HAL\_IncTick**(void)**

{

uwTick += uwTickFreq;

}

=> Hiểu đơn giản mỗi 1ms VĐK sẽ nhảy vào hàm ngắt Systick (SysTick\_Handler()) và thực hiện tăng biến uwTick lên 1 đơn vị (HAL\_IncTick()).

Giả sử ta có một ngắt A có độ ưu tiên ngang bằng hoặc hơn so với ngắt Systick, khi ngắt A được gọi và có hàm HAL\_Delay(), sau khi biến tickstart lấy được giá trị trả về của hàm GetTick, vòng while để delay sẽ xảy ra, tuy nhiên ngắt Systick lúc này đang ở chế độ chờ (do ngắt A chưa chạy xong) như vậy, hàm GetTick không thể tăng được giá trị trả về, dẫn tới treo while.

﻿

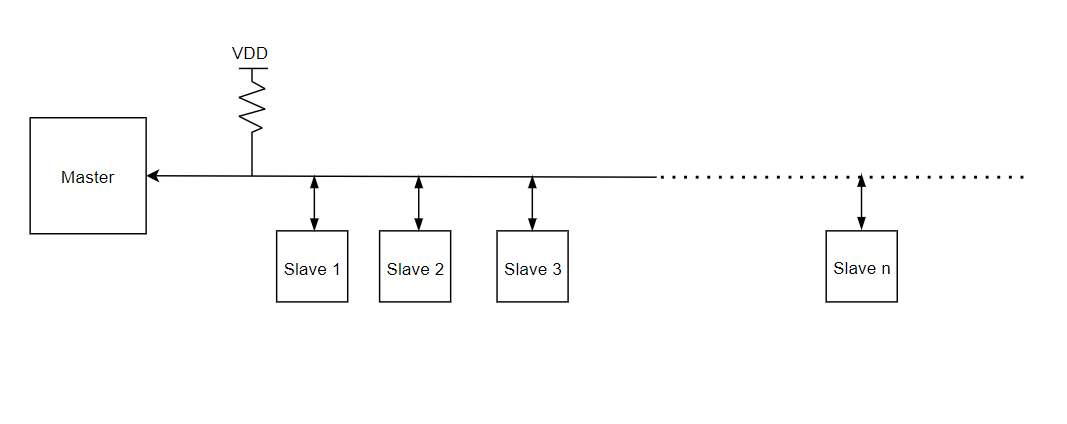
Nên xem xét sử dụng hàm delay trong ngắt khi thực sự cần thiết vì các chương trình phục vụ ngắt nên được xử lý tức thời và càng ngắn gọn càng tốt, tránh ảnh hưởng đến các ngắt sau, không đáp ứng được realtime của hệ thống dẫn đến bỏ lỡ sự kiện hoặc mất dữ liệu.

**Giao tiếp OneWire**

Giao tiếp OneWire trong lập trình nhúng

I. Lý thuyết

* OneWire là hệ thống bus giao tiếp được thiết kế bởi [Dallas Semiconductor Corp](https://en.wikipedia.org/wiki/Dallas_Semiconductor). Giống như tên gọi, hệ thống bus này chỉ sử dụng 1 dây để truyền nhận dữ liệu.
* Chính vì chỉ sử dụng 1 dây nên giao tiếp này có tốc độ truyền thấp nhưng dữ liệu lại truyền được khoảng cách xa hơn.
* OneWire chủ yếu sử dụng để giao tiếp với các thiết bị nhỏ, thu thập và truyền nhận dữ liệu thời tiết, nhiệt độ,… các công việc không yêu cầu tốc độ cao.
* Giống như các chuẩn giao tiếp khác, 1-Wire cho phép truyền nhận dữ liệu với nhiều Slave trên đường truyền. Tuy nhiên chỉ có thể có 1 Master ( điểm này giống với SPI).



II. Khung truyền của giao tiếp OneWire

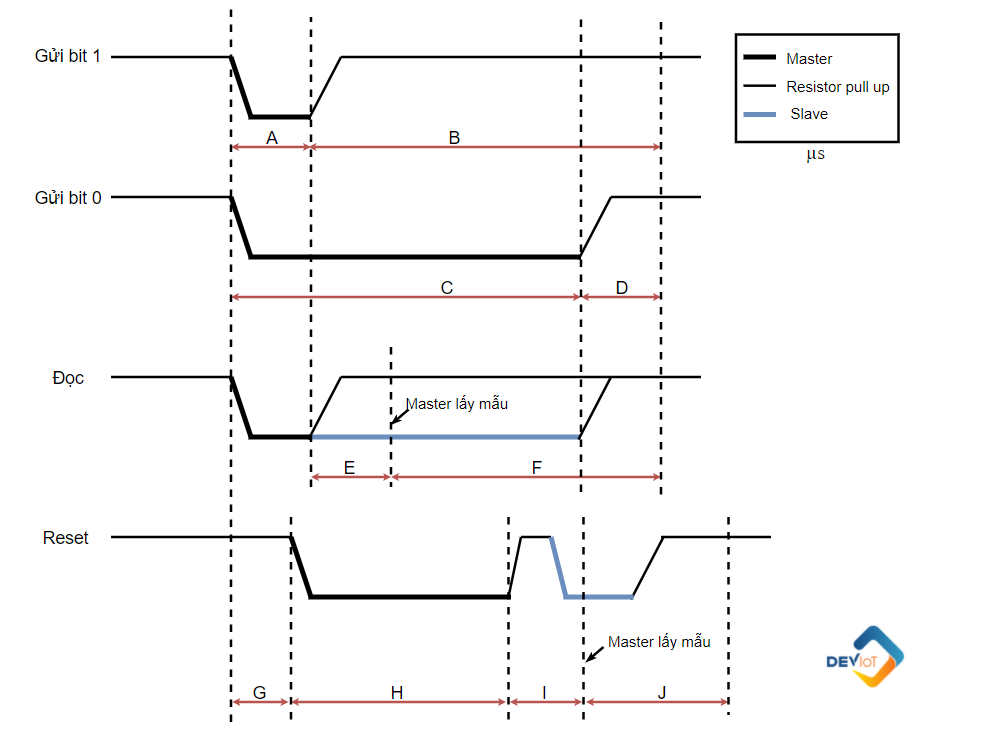
**1-Wire hoạt động như thế nào ?**

So với các chuẩn giao tiếp cơ bản như UART, SPI, I2C mà chúng ta đã biết, cách thức hoạt động của OneWire có hơi “ lạ “ 1 chút.

Như chúng ta thấy ở hình trên, đường dây luôn được giữ ở mức cao (High).

Các thao tác hoạt động cơ bản của bus sẽ được quy định bởi thời gian kéo đường truyền xuống mức thấp (Low) như hình vẽ dưới.

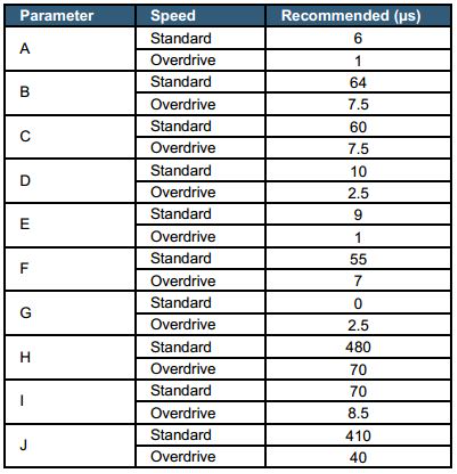
Có 4 thao tác cơ bản như sau:



 Giải thích ý nghĩa :

* Gửi bit 1: Khi muốn gửi đi bit 1, thiết bị Master sẽ kéo bus xuống mức 0 trong một khoảng thời gian A (µs) và trở về mức 1 trong khoảng B (µs).
* Gửi bit 0: Thiết bị Master kéo bus xuống mức 0 trong một khoảng thời gian C (µs) và trở về mức 1 trong khoảng D (µs).
* Đọc bit: Thiết bị Master kéo bus xuống 1 khoảng A (µs). Trong khoảng thời gian E (µs) tiếp theo, thiết bị master sẽ tiến hành lấy mẫu. Có nghĩa trong E (µs) này, nếu bus ở mức 1, thiết bị master sẽ đọc bit 1. Ngược lại, nếu bus ở mức 0 thì master sẽ đọc được bit 0.
* Reset:  Thiết bị Master kéo bus xuống 1 khoảng thời gian H (µs) và sau đó về mức 1. Khoảng thời gian này gọi là tín hiệu reset. Trong khoảng thời gian I (µs) tiếp theo, thiết bị master tiến hành lấy mẫu. Nếu thiết bị slave gắn với bus gửi về tín hiệu 0, (tức bus ở mức 0), master sẽ hiểu rằng slave vẫn có mặt và quá trình trao đổi dữ liệu lại tiếp tục. Ngược lại nếu slave gửi về tin hiệu 1 ( bus ở mức 1) thì master hiểu rằng không có thiết bị slave nào tồn tại và dừng quá trình.

Bảng giá trị thời gian



***Chế độ hoạt động:***

* Chế độ Standard - Chế độ tiêu chuẩn:

o  15.4 Kb/s

o  65 µs bit

* Chế độ Overdrive – Chế độ tốc độ nhanh:

o  125 Kb/s

o  8 µs bit

**ĐÓNG GÓI DỮ LIỆU TRONG TRUYỀN NHẬN UART.**

Author: Sơn Tùng

Chắc anh em làm nhúng không xa lạ gì với Uart rồi. Giao thức khá đơn giản, truyền nhận theo byte.

Nhưng trong thực tế nếu truyền nhận 1 lượng dữ liệu lớn thì thường gặp vấn đề như sau.

 Giả sử vi điều khiển A truyền và vi điều khiển B nhận dữ liệu liên tục :

0x10 0x20 0x30 ... 0x90

 Ta lập trình cho nhận 10 byte 1 lần, nhưng vì 1 sự cố nào đó, dây lỏng, mất nguồn,..., khiến cho quá trình bị gián đoạn, lúc sau thì dữ liệu truyền nhận trở thành:

0x30 0x40 ... 0x90 0x10 0x20 bị sai thứ tự bắt đầu.

 Để giải quyết vấn đề trên thì có 1 cách đơn giản nhất là đóng dữ liệu lại thành dạng "gói", bằng cách tự thêm 2 byte bắt đầu (START BYTE) và kết thúc(STOP BYTE) vào gói dữ liệu.( gọi là flag byte - các cờ kiểm tra)

 Ví dụ với gói dữ liệu trên, mình chọn thêm START BYTE là 0x55 và STOP BYTE 0xAA, như vậy dữ liệu vdk A truyền đi sẽ là:

0x55 0x10 0x20 ... 0x90 0xAA.

 Ở vđk B, kiểm tra xem byte nhận được có phải START BYTE: 0x55 không, nếu đúng thì mới bắt đầu nhận những byte sau, và nếu nhận được STOP BYTE 0xAA thì dừng quá trình nhận. Như vậy cho dù có bị gián đoạn như trên thì vdk B vẫn nhận đúng gói dữ liệu gửi sang.

 **Vấn đề 2**: Thế giả sử trong gói dữ liệu gửi đi, có 1 byte trùng với START BYTE hoặc STOP BYTE mình đã chọn thì sao ??

**Ví dụ:**

0x55 0x10 0x20 ... 0x55 ... 0xAA ... 0x80 0x90 0xAA

Anh em sẽ giải quyết như thế nào tiếp ??

**Watchdog Timer**

Author: Tuấn Anh

Ai đã từng làm thực tế với với vi điều khiển đều đã gặp tình trạng treo chip, nguyên nhân thì rất nhiều ví dụ như: nhiễu điện từ trường (EMC), nhiễu phóng tĩnh điện (ESC), nhiễu sụt áp nguồn, do lập trình lỗi, có 1 số vòng lặp không thoát ra được,... Để khắc phúc tình trạng treo chip, các nhà sản xuất thường cung cấp tính năng watchdog timer cho chip.

Watchdog timer là 1 bộ đếm có khả năng hoạt động độc lập với CPU, có thể là đến tăng hoặc giảm tùy dòng chip. Khi mà nó đếm đến một giá trị nào đó (do người dùng đặt) thì nó sẽ kích hoạt reset chip. Đúng như tên gọi mang ý nghĩa là cún canh giữ phải không nào 

#### **ỨNG DỤNG**

* Giả sử ta dùng 1 watchdog timer đếm tăng và giả sử rằng khi nó đếm đến giá trị 69 thì sẽ kích hoạt reset chip. Ta sẽ sử dụng 1 hàm nạp lại giá trị 0 cho bộ đếm của watchdog trong vòng lặp while(1) để watchdog đếm lại từ 0 đảm bảo là watchdog không đếm đến giá trị 69 (tức là chip sẽ không bị reset).
* Nếu chương trình của chúng ta bị treo thì bộ đếm của watchdog sẽ không được nạp lại về 0 và sẽ tăng lên giá trị 69, kết quả là chip bị reset.

Ví dụ về 1 cách dùng:

void main()

{

watchdog\_init(); // Khởi tạo watchdog timer

while(1)

{

//do something here :))

watchdog\_reload(); // nạp lại giá trị cho watchdog

}

}

Nhưng mà phòng vẫn tốt hơn chữa, tốt nhất hãy dùng phần cứng thật chuẩn (chống nhiễu tốt) và nên chú ý đến những vòng lặp trong chương trình.

Đa phần các dòng chip bán tại Việt Nam đều hỗ trợ watchdog timer nên mình khuyến khích các bạn hãy học cách sử dụng nó nhé 

**Chia nhỏ khối dữ liệu lớn với UNION + STRUCT và ứng dụng trong lập trình nhúng.**

Trong lập trình nhúng việc xử lý chuỗi và mảng byte là rất quan trọng. Vì thế nên các thủ thuật xử lý chuỗi là rất hữu ích. Bài ngày hôm nay sẽ giới thiệu một thủ thuật nhỏ trong đó.

**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;

}

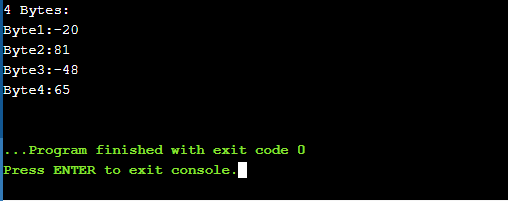
Như vậy ta đã có thể tách khối dữ liệu 4 bytes riêng từng byte 1 để xử lý. Đồng thời ta cũng có thể từ các dữ liệu nhỏ gộp thành một khối dữ liệu lớn.

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 · 7964 views

**Con trỏ và tham chiếu (Pointers vs references)**

Tuấn Anh

Cả con trỏ và tham chiếu đều cho phép bạn truy cập đến các đối tượng gián tiếp khác. Như vậy làm thế nào để phân biệt và biết khi nào thì nên dùng con trỏ và khi nào thì dùng tham chiếu? Ta sẽ cùng đi tìm hiểu trong bài viết này.

## ****KHỞI TẠO****

Ví dụ ta sẽ khởi tạo tham chiếu và con trỏ đến 1 biến int a = 5:

Khai báo biến con trỏ: <kiểu dữ liệu> \*<tên biến>;

int \*p = &a;

hoặc

int \*p;

p = &a;

Khai báo biến tham chiếu: <kiểu dữ liệu> &<tên biến> = <tên đối tượng>;

int &p = a;

## ****TÍNH CHẤT****

* Con trỏ có thể được chỉ định để truy cập đến các đối tượng khác nhau bằng cách thay đổi giá trị địa chỉ mà con trỏ lưu giữ. Tuy nhiên, tham chiếu chỉ truy cập được duy nhất một đối tượng đã được khởi tạo cho nó. Vì vậy con trỏ linh hoạt hơn tham chiếu.
* Kích thước một biến con trỏ là cố định. Kích thước biến tham chiếu phụ thuộc kiểu dữ liệu của đối tượng.
* Tham chiếu thường được dùng làm các tham số truyền vào hàm khi muốn giá trị của tham số thay đổi sau khi ra khỏi hàm. Khi thay đổi giá trị bởi con trỏ thì ở bên ngoài hàm cũng sẽ thay dổi ("pass by value" và "pass by reference")
* Con trỏ có thể được gán NULL trực tiếp, còn tham chiếu thì không thể, một tham chiếu phải luôn chỉ đến một đối tượng nào đó.
* Các phép toán số học khác nhau có thể được thực hiện trên con trỏ trong khi tham chiếu thì không.

## ﻿****CÁCH SỬ DỤNG****

Sử dụng tham chiếu:

* Trong các tham số của hàm và kiểu trả về.

Sử dụng con trỏ:

* Nếu có liên quan đến các phép toán số học hoặc cần sử dụng NULL-pointer. Ví dụ truy cập mảng được thực hiện bằng cách sử dụng phép toán ("++","--",...)cho con trỏ.
* Để triển khai các cấu trúc dữ liệu như danh sách liên kết, cây, ... và các thuật toán của chúng vì để trỏ vào các ô khác nhau, chúng ta phải sử dụng khái niệm con trỏ.

# 8.0 Con trỏ

#### Chào các bạn học viên đang theo dõi khóa học lập trình trực tuyến ngôn ngữ C++.

Trong chương này, chúng ta sẽ cùng nhau tìm hiểu về khái niệm con trỏ (**Pointer**) - một đặc trưng của ngôn ngữ lập trình C/C++.

Trước khi vào bài học này, chúng ta cùng nhau xem lại một số khái niệm liên quan đến vùng nhớ, địa chỉ của biến, tham chiếu...

##### Variable

Variable (hay còn gọi là biến) là một ô nhớ đơn lẻ hoặc một vùng nhớ được hệ điều hành cấp phát cho chương trình C++ nhằm để lưu trữ giá trị vào bên trong vùng nhớ đó. Để truy xuất đến giá trị mà biến đang nắm giữ, chương trình cần tìm đến vùng nhớ (địa chỉ) của biến để đọc giá trị bên trong vùng nhớ đó, cũng như bạn muốn lấy món đồ bên trong cái hộp, bạn cần biết cái hộp được đặt ở đâu.

Khi thao tác với các biến thông thường, chúng ta không cần quan tâm đến địa chỉ vùng nhớ của biến. Khi cần truy xuất giá trị của biến, chúng ta chỉ cần gọi định danh (hay thường gọi là tên biến).

Ví dụ:

int money*;*

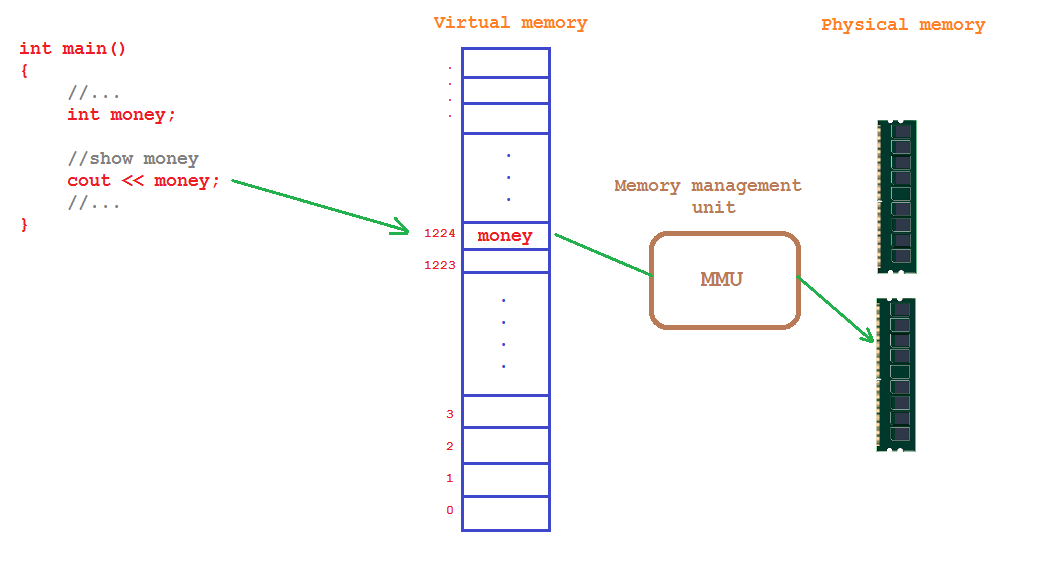
Khi dòng lệnh này được CPU thực thi, một vùng nhớ có kích thước 4 bytes sẽ được cấp phát. Lấy ví dụ biến money này được đặt tại ô nhớ 1224 (trong địa chỉ ảo của máy tính).



Bất cứ khi nào chương trình thấy các bạn sử dụng biến money trong câu lệnh, chương trình hiểu rằng cần tìm đến ô nhớ 1224 để lấy giá trị đó ra.

##### Virtual memory & Physical memory

Việc truy xuất dữ liệu trên bộ nhớ máy tính cần phải thông qua một số bước trung gian, người dùng không thể trực tiếp truy xuất vào các ô nhớ trên các thiết bị lưu trữ. Chúng ta chỉ có thể trỏ đến vùng nhớ ảo (virtual memory) trên máy tính, còn việc truy xuất đến bộ nhớ vật lý (physical memory) từ bộ nhớ ảo phải được thực hiện bởi thiết bị phần cứng có tên là **Memory management unit (MMU)** và một chương trình định vị địa chỉ bộ nhớ gọi là **Virtual address space**.

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-0-con-tro/1.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-0-con-tro/1.png?raw=true" \o "1.png?raw=true)**

**[1.png?raw=true1045x566](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-0-con-tro/1.png?raw=true" \o "1.png?raw=true)**

Virtual memory làm che giấu sự phân mảnh của bộ nhớ vật lý, khiến chúng ta có cảm giác đang thao tác với các vùng nhớ liên tục. Trong hình trên, từ phía Virtual memory cho đến Physical memory thuộc về phần quản lý của hệ điều hành, lập trình viên và người dùng chúng ta không thể can thiệp trực tiếp đến trong quá trình máy tính đang hoạt động.

##### Variable address & address-of operator

Địa chỉ của biến mà chúng ta nhìn thấy thật ra chỉ là những giá trị đã được đánh số thứ tự đặt trên Virtual memory. Để lấy được địa chỉ ảo của biến trong chương trình, chúng ta sử dụng toán tử **'&'** đặt trước tên biến.

**int** x = 5;

std::cout << x << '\n'; *// print the value of variable x*

std::cout << &x << '\n'; *// print the memory address of variable x*

Trên máy tính của mình, kết quả của đoạn chương trình trên được in ra như sau:

5

0027FEA0

Dòng đầu tiên là kết quả của việc truy xuất giá trị của biến thông qua định danh (tên biến). Dòng thứ hai là kết quả của việc truy xuất đến địa chỉ ảo của biến.

##### Tham chiếu (Reference)

Mục đích của tham chiếu trong C++ là tạo ra một biến khác có cùng kiểu dữ liệu nhưng sử dụng chung vùng nhớ với biến được tham chiếu đến.

**int** i1 = 10;

**int** &i\_ref = i1; *//reference to i1, not means address of i1*

cout << &i1 << endl; *//get address of i1*

cout << &i\_ref << endl; *//get address of i\_ref*

Kết quả chúng ta được:

0xBFEB475C

0xBFEB475C

Như vậy, mọi hành vi thay đổi giá trị của i\_ref đều tác động trực tiếp đến i1.

**Lưu ý: Biến tham chiếu sẽ có địa chỉ cố định sau khi khởi tạo. Chúng ta không thể tham chiếu lại lần nữa.**

##### Dereference operator

Toán tử trỏ đến (**dereference operator**) hay còn gọi là **indirection operator** (toán tử điều hành gián tiếp) được kí hiệu bằng dấu sao **" \* "** cho phép chúng ta lấy ra giá trị của vùng nhớ có địa chỉ cụ thể.

Ví dụ:

**int** n = 5;

cout << n << endl; *//print the value of variable n*

cout << &n << endl; *//print the virtual memory address of variable n*

cout << \*(&n) << endl; *//print the value at the virtual memory address of variable n*

* Dòng lệnh cout đầu tiên khá dễ hiểu, nó thực hiện in ra giá trị của biến n bằng cách gọi định danh n, còn lại phần truy xuất đến địa chỉ ảo của biến n sẽ do chương trình đảm nhiệm.
* Dòng lệnh cout thứ hai không dùng để lấy ra giá trị bên trong vùng nhớ mà biến n đang nắm giữ, mà nó lấy ra địa chỉ ảo của biến n.
* Dòng lệnh cout thứ ba chúng ta sử dụng toán tử trỏ đến **" \* "** đặt trước toán tử **address-of**. Khi đó, **(&n)** sẽ lấy ra địa chỉ ảo của biến n, và toán tử \* sẽ truy xuất giá trị bên trong địa chỉ đó.

Kết quả của đoạn chương trình trên là:

5

0xBFD181AC

5

Ngoài việc truy xuất giá trị trong vùng nhớ của một địa chỉ cụ thể, toán tử trỏ đến (**dereference operator**) còn có thể dùng để thay đổi giá trị bên trong vùng nhớ đó.

**int** n = 5;

cout << n << endl;

\*(&n) = 10;

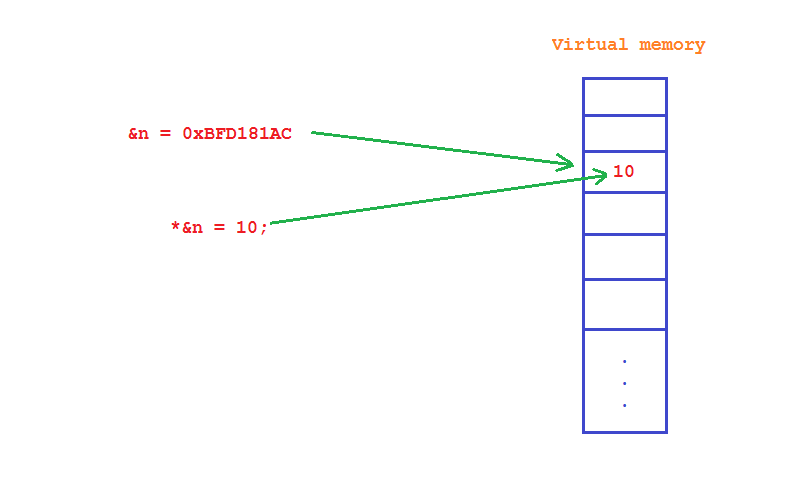
cout << n << endl;

Kết quả đoạn chương trình này là:

5

10

Như vậy, **dereference operator** cho phép chúng ta thao tác trực tiếp trên **Virtual memory** mà không cần thông qua định danh (tên biến).

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-0-con-tro/2.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-0-con-tro/2.png?raw=true" \o "2.png?raw=true)**

**[2.png?raw=true786x490](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-0-con-tro/2.png?raw=true" \o "2.png?raw=true)**

Mặc dù **dereference operator** có kí hiệu giống **multiplication operator**, nhưng các bạn có thể phân biệt được vì **dereference operator** là toán tử một ngôi, trong khi đó, **multiplication operator** là toán tử hai ngôi.

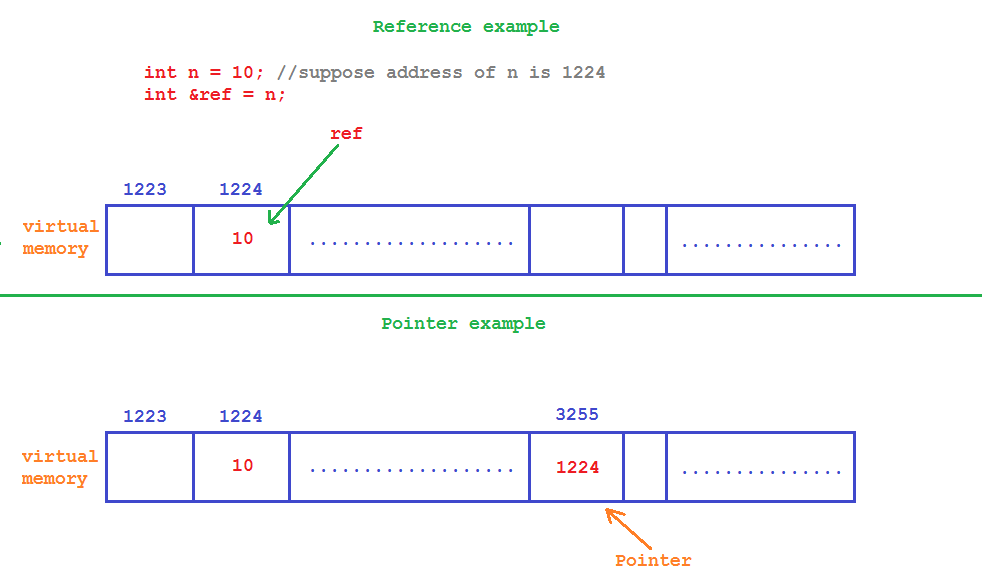
Khác với tham chiếu (**reference**), toán tử trỏ đến (**dereference operator**) không tạo ra một tên biến khác, mà nó truy xuất trực tiếp đến vùng nhớ có địa chỉ cụ thể trên **Virtual memory**.

### Con trỏ (Pointer)

Với những khái niệm mình trình bày ở trên (một số khái niệm các bạn đã được học), bây giờ chúng ta có thể nói đến con trỏ (**pointer**).

Một con trỏ (a **pointer**) là một biến được dùng để lưu trữ địa chỉ của biến khác.

Khác với tham chiếu, con trỏ là một biến có địa chỉ độc lập so với vùng nhớ mà nó trỏ đến, nhưng giá trị bên trong vùng nhớ của con trỏ chính là địa chỉ của biến (hoặc địa chỉ ảo) mà nó trỏ tới.

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-0-con-tro/3.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-0-con-tro/3.png?raw=true" \o "3.png?raw=true)**

**[3.png?raw=true983x586](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-0-con-tro/3.png?raw=true" \o "3.png?raw=true)**

Trong ví dụ trên, một con trỏ sau khi khai báo đã được cấp phát vùng nhớ tại địa chỉ 3255, và nó trỏ đến địa chỉ 1224, do đó, giá trị bên trong vùng nhớ của con trỏ là 1224.

##### Khai báo con trỏ

Cũng giống như biến thông thường, biến con trỏ cần được khai báo trước khi sử dụng. Con trỏ yêu cầu cú pháp khai báo mới hơn một chút so với biến thông thường.

<data\_type> \*<name\_of\_pointer>;

Khác với biến thông thường, chúng ta cần đặt thêm dấu sao giữa kiểu dữ liệu và tên biến của con trỏ.

Ví dụ:

**int** \*iPtr;

**float** \*fPtr;

**double** \*dPtr;

**int** \*iPtr1, \*iPtr2;

**Lưu ý: Dấu sao trong khai báo con trỏ không phải là toán tử trỏ đến (dereference operator), nó chỉ là cú pháp được ngôn ngữ C/C++ quy định.**

##### Cách khai báo dễ gây nhầm lẫn

Ngôn ngữ C/C++ yêu cầu đặt dấu sao giữa kiểu dữ liệu và tên con trỏ nhưng không bắt buộc phải đặt nó gần với kiểu dữ liệu hay gần với tên con trỏ. Do đó, những cách khai báo dưới đây đều được cho phép:

**int** \*iPtr1; *//We recommended you use this way to declare pointers*

**int**\* iPtr2;

Nhưng mình khuyến nghị các bạn sử dụng cách khai báo đặt dấu sao ngay trước tên con trỏ vì cách thứ hai có thể gây nhầm lẫn.

**int**\* iPtr1, iPtr2;

Với cách khai báo này, **iPtr1** là một con trỏ kiểu **int**, trong khi đó, **iPtr2** là một biến kiểu **int**. Để có được hai con trỏ, chúng ta cần khai báo như sau:

**int** \*iPtr1, \*iPtr2;

##### Kích thước của con trỏ trong bộ nhớ

Các bạn cùng chạy thử đoạn chương trình dưới đây:

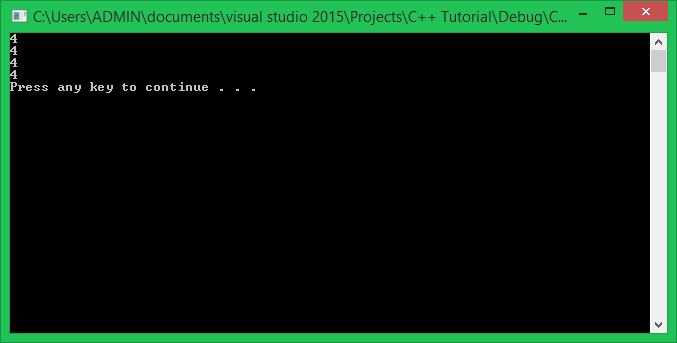
cout << **sizeof**(**char**\*) << endl;

cout << **sizeof**(**int**\*) << endl;

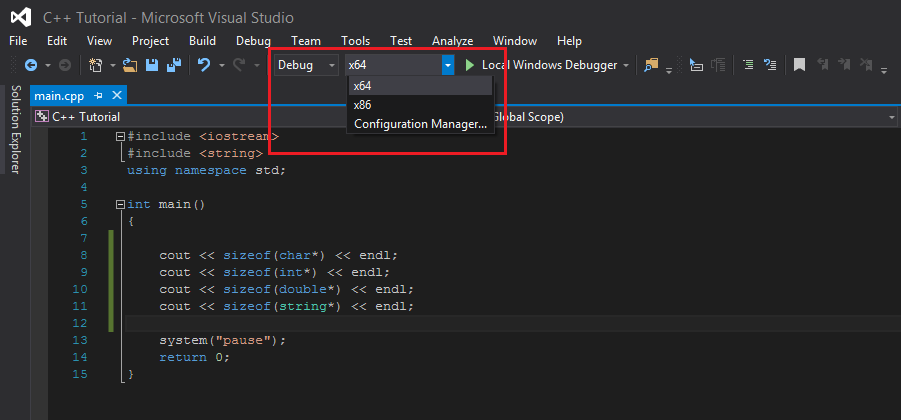
cout << **sizeof**(**double**\*) << endl;

cout << **sizeof**(string\*) << endl;

Đoạn chương trình trên cho ra kết quả như sau:

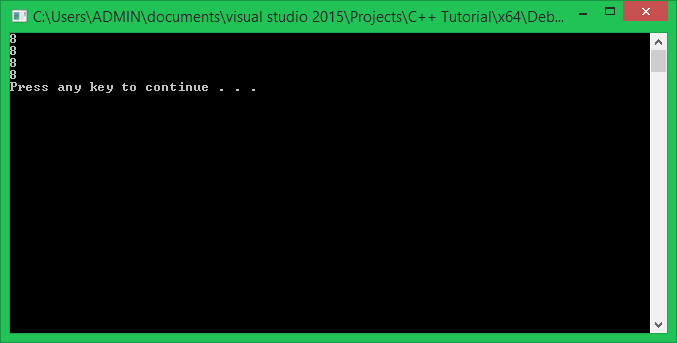


Tại cửa sổ giao diện của Visual studio 2015, chúng ta chuyển sang Debug trên nền tảng 64 bits.

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-0-con-tro/5.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-0-con-tro/5.png?raw=true" \o "5.png?raw=true)**

**[5.png?raw=true901x420](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-0-con-tro/5.png?raw=true" \o "5.png?raw=true)**

Nhấn F5 lại lần nữa và xem lại kết quả:



**Như vậy, chúng ta thấy rằng khi chạy trên nền tảng hệ điều hành 32 bits con trỏ sẽ có kích thước 4 bytes, khi chạy trên nền tảng hệ điều hành 64 bits con trỏ sẽ có kích thước 8 bytes.**

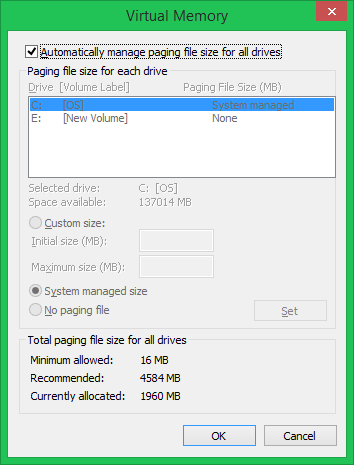
Kiểu dữ liệu của con trỏ thay đổi không hề tác động đến kích thước bộ nhớ của con trỏ. Bởi vì giá trị thực sự của con trỏ là kiểu số nguyên không dấu (**unsigned int**), trong nền tảng hệ điều hành **32 bits**, giá trị mà con trỏ lưu trữ sẽ là unsigned \_\_int32, và trong nền tảng hệ điều hành **64 bits**, giá trị của con trỏ lưu trữ có kiểu unsigned \_\_int64.

**Kiểu dữ liệu của con trỏ không mô tả giá trị địa chỉ được lưu trữ bên trong con trỏ, mà kiểu dữ liệu của con trỏ dùng để xác định kiểu dữ liệu của biến mà nó trỏ đến trên bộ nhớ ảo.**

Vậy tại sao lại cần 4 bytes cho một con trỏ trong hệ điều hành 32 bits, và cần 8 bytes cho một con trỏ trong hệ điều hành 64 bits?

Phạm vi giá trị của unsigned \_\_int32 là từ 0 đến 4294967295, phạm vi giá trị này đủ để đánh dấu địa chỉ tất cả ô nhớ trong **Virtual memory** của nền tảng hệ điều hành **32 bits**. Tương tự, phạm vi giá trị của unsigned \_\_int64 là tử 0 đến 18446744073709551615, đủ để đánh dấu địa chỉ của tất cả ô nhớ trong **Virtual memory** của nền tảng hệ điều hành **64 bits**.

Dưới đây là thông tin Virtual memory trên máy tính của mình:



Dung lượng bộ nhớ ảo hiện tại của máy mình là **1960MB**, tương đương với **2055208960 bytes**. Trong khi đó, con trỏ trong nền tảng hệ điều hành **32 bits** có kích thước **4 bytes**, giá trị địa chỉ lớn nhất mà con trỏ **4 bytes** có thể lưu trữ được là **4294967295**, nên nó đủ để lưu trữ bất kì địa chỉ của biến nào được cấp phát trên bộ nhớ ảo.

##### Gán giá trị cho con trỏ

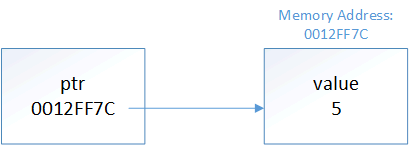
Giá trị mà biến con trỏ lưu trữ là địa chỉ của biến khác có cùng kiểu dữ liệu với biến con trỏ.

int \*ptr*;*

int value = 5*;*

ptr = &value*;*

Do đó, chúng ta cần sử dụng **address-of operator** để lấy ra địa chỉ ảo của biến rồi mới gán cho con trỏ được. Lúc này, biến ptr sẽ lưu trữ địa chỉ ảo của biến value.



(Nguồn: [**www.learncpp.com**](http://www.learncpp.com/))

Chúng ta có thể nói rằng con trỏ **ptr** đang nắm giữ địa chỉ của biến **value**, cũng có thể nói con trỏ **ptr** trỏ đến biến **value**.

Đoạn chương trình sau sẽ in ra địa chỉ của biến value và giá trị được lưu bởi con trỏ ptr sau khi trỏ đến biến value:

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

{

**int** value = 5;

**int** \*ptr = &value;

cout << &value << endl;

cout << ptr << endl;

system("pause");

**return** 0;

}

Kết quả thu được trên màn hình console:

0012FF7C

0012FF7C

Lý do mà chúng ta gán được địa chỉ của biến **value** cho con trỏ kiểu **int** (int \*) là vì **address-of operator** của một biến kiểu **int** trả về giá trị kiểu con trỏ kiểu **int** (int \*).

Thử xem xét đoạn chương trình sau:

**#include <iostream>**

**using** **namespace** std;

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

{

**int** value = 5;

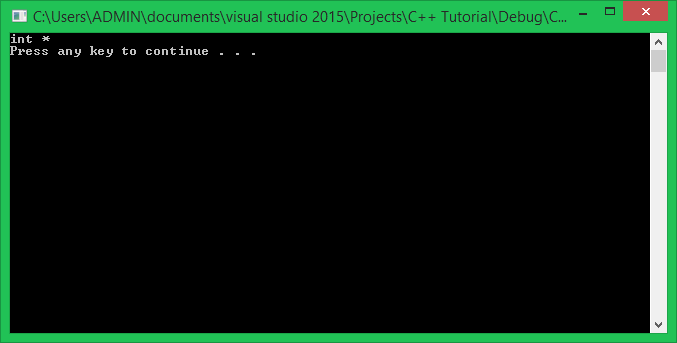
cout << **typeid**(&value).name() << endl;

system("pause");

**return** 0;

}

Kết quả in ra màn hình của toán tử lấy địa chỉ ở trên là:



Do đó, chúng ta có thể gán &value cho con trỏ kiểu **int** (int \*).

Bên cạnh đó, khi có hai con trỏ cùng kiểu thì chúng ta có thể gán trực tiếp mà không cần sử dụng **address-of operator**.

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

{

**int** value = 5;

**int** \*ptr1, \*ptr2;

ptr1 = &value; *//ptr1 point to value*

ptr2 = ptr1; *//assign value of ptr1 to ptr2*

cout << ptr1 << endl;

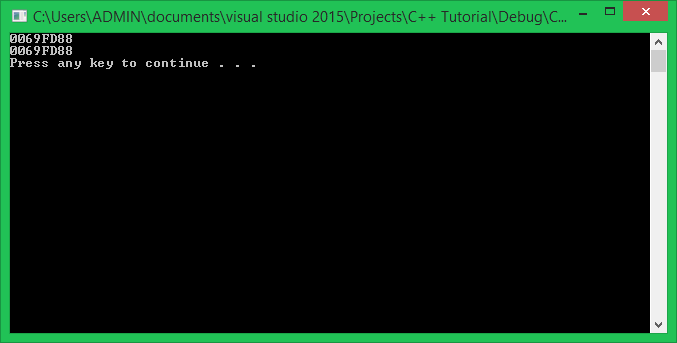
cout << ptr2 << endl;

system("pause");

**return** 0;

}

Lúc này, **ptr1** và **ptr2** cùng giữ địa chỉ của biến **value**.



**Khác với tham chiếu (reference), một con trỏ có thể trỏ đến địa chỉ khác trong bộ nhớ ảo sau khi đã được gán giá trị. Tham chiếu (reference) không thể thay đổi địa chỉ sau lần tham chiếu đầu tiên.**

Ví dụ:

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

{

**int** \*ptr;

**int** arr[5] = { 1, 2, 3, 4, 5 };

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

{

ptr = &arr[i];

cout << ptr << endl;

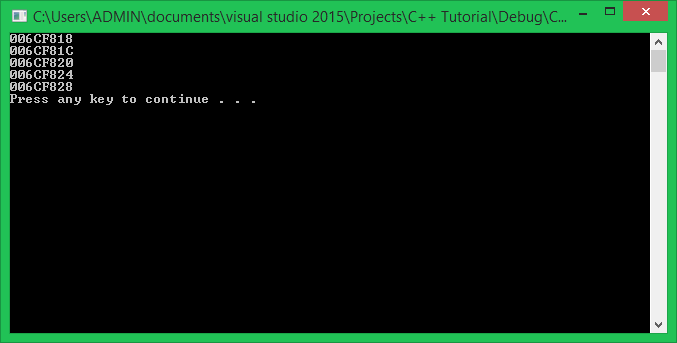
}

system("pause");

**return** 0;

}

Kết quả của đoạn chương trình này là:



Con trỏ **ptr** đã trỏ đến lần lượt 5 phần tử của mảng **arr**. Nếu các bạn để ý sẽ thấy 5 địa chỉ này liên tiếp nhau trên bộ nhớ ảo. Mình sẽ trình bày vấn đề này trong các bài học sau.

##### Các phép gán không hợp lệ khi sử dụng con trỏ

Phép gán của con trỏ chỉ thực hiện được khi kiểu dữ liệu của con trỏ phù hợp kiểu dữ liệu của biến mà nó sẽ trỏ tới. Do đó, các phép gán dưới đây là không hợp lệ:

int iValue = 0;

float fValue = 0.0;

int \*i\_ptr = fValue; //wrong! int pointer cannot point to the address **of** a double variable

float \*f\_ptr = iValue; //wrong! float pointer cannot point to the address **of** an int variable

Mặc dù giá trị mà con trỏ lưu trữ có kiểu **unsigned int**, nhưng chúng ta không thể gán trực tiếp một giá trị địa chỉ cho con trỏ được.

**int** \*ptr = 1245052; *//wrong!*

Giá trị 1245052 không có địa chỉ cụ thể, trong khi đó, con trỏ chỉ nhận giá trị là địa chỉ nên phép gán trên là sai. Mặc dù giá trị được chuyển về dạng cơ số thập lục phân để tương xứng với định dạng giá trị mà con trỏ in ra, điều này cũng không được cho phép.

**int** \*ptr = 0012FF7C; *//wrong!*

**Chỉ có giá trị kiểu con trỏ (có được nhờ toán tử address-of, hoặc từ một biến con trỏ cùng kiểu khác) mới có thể gán được cho biến con trỏ.**

##### Truy xuất giá trị bên trong vùng nhớ mà con trỏ trỏ đến

Khi chúng ta có một con trỏ đã được trỏ đến địa chỉ nào đó trong bộ nhớ ảo, chúng ta có thể truy xuất giá trị tại địa chỉ đó bằng **dereference operator**. **Dereference operator** sẽ đánh giá nội dung địa chỉ được trỏ đến.

**int** \*ptr; *//declare an int pointer*

**int** value = 5;

ptr = &value; *//ptr point to value*

cout << &value << endl; *//print the address of value*

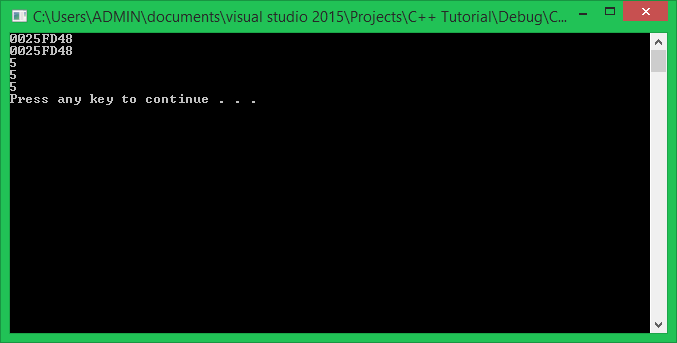
cout << ptr << endl; *//print the address of value which is held in ptr*

cout << value << endl; *//print the content of value*

cout << \*(&value) << endl; *//print the content of value*

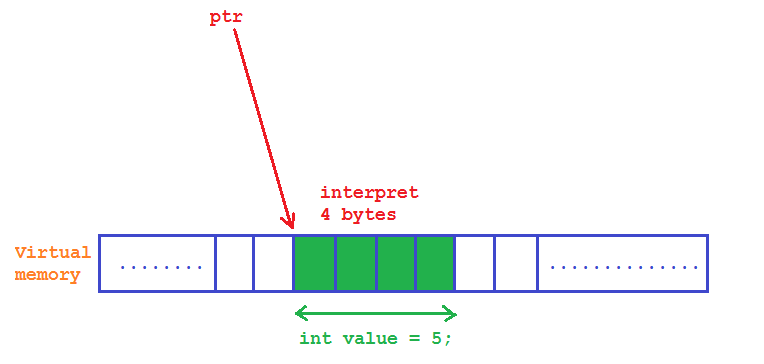
cout << \*ptr << endl; *//print the content of value*

Kết quả của đoạn chương trình trên như sau:



Toán tử trỏ đến (**dereference operator**) được dùng để truy cập trực tiếp vào vùng nhớ có địa chỉ cụ thể trên bộ nhớ ảo (**virtual memory**), vì biến con trỏ **ptr** đang giữ địa chỉ của biến **value** nên khi đặt toán tử trỏ đến (**dereference operator**) trước con trỏ **ptr**, nó sẽ truy xuất giá trị tại địa chỉ mà con trỏ **ptr** đang giữ.

Vì **ptr** có kiểu dữ liệu **con trỏ int** (int \*), **ptr** chỉ có thể trỏ đến biến kiểu **int**. Lúc này, **compiler** hiểu rằng cần phân tích **4 bytes** (đúng bằng kích thước kiểu **int**) trên bộ nhớ ảo tại địa chỉ mà **ptr** đang lưu trữ.

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-0-con-tro/12.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-0-con-tro/12.png?raw=true" \o "12.png?raw=true)**

**[12.png?raw=true764x364](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-0-con-tro/12.png?raw=true" \o "12.png?raw=true)**

Đây là lý do tại sao chúng ta cần khai báo kiểu dữ liệu của con trỏ. Nếu không khai báo kiểu dữ liệu cho con trỏ, toán tử trỏ đến (**dereference operator**) sẽ không biết phải phân tích bao nhiêu bytes tại địa chỉ con trỏ trỏ đến để tính toán được giá trị của vùng nhớ đó. Không những thế, đây còn là lý do kiểu dữ liệu của biến phải tương xứng với kiểu dữ liệu được khai báo cho con trỏ.

Vì chúng ta có thể gán lại địa chỉ mới cho một con trỏ, nên chúng ta có thể truy xuất được giá trị của nhiều vùng nhớ khác nhau chỉ với một con trỏ:

**int** value1 = 1;

**int** value2 = 2;

**int** \*ptr = &value1;

cout << \*ptr << endl;

ptr = &value2;

cout << \*ptr << endl;

Với khả năng truy cập đến vùng nhớ có địa chỉ cụ thể và thay đổi giá trị bên trong vùng nhớ của toán tử trỏ đến (**dereference operator**), chúng ta có thể sử dụng như sau:

int value = 5*;*

int \*ptr = &value*;*

\*ptr = 10*;*

cout << \*ptr << endl*;*

Đoạn chương trình này sẽ in ra giá trị **10**.

Có thể giải thích dòng lệnh \*ptr = 10; như sau:

Biến con trỏ **ptr** sau khi khai báo đã được khởi tạo bằng cách gán địa chỉ của biến **value**. Sử dụng **dereference operator** cho con trỏ **ptr** để truy cập đến địa chỉ ảo mà **ptr** đang nắm giữ, gán giá trị 10 vào vùng nhớ tại vị trí đó.

##### Con trỏ chưa được gán địa chỉ

Con trỏ trong ngôn ngữ C/C++ vốn không an toàn. Nếu sử dụng con trỏ không hợp lý có thể gây crash chương trình.

Khác với tham chiếu (**reference**), biến con trỏ có thể không cần khởi tạo giá trị ngay khi khai báo. Nhưng thực hiện truy xuất giá trị của con trỏ bằng dereference operator khi chưa gán địa chỉ cụ thể cho con trỏ, chương trình có thể bị đóng bởi hệ điều hành. Nguyên nhân là do con trỏ đang nắm giữ một giá trị rác, giá trị rác đó có thể là địa chỉ thuộc một vùng nhớ đang được ứng dụng khác sử dụng, hoặc giá trị vượt quá giới hạn của bộ nhớ ảo.

Trong chế độ **Debug** của Visual studio 2015, trường hợp này sẽ bị cảnh báo và ngăn chặn chương trình thực thi.

Ví dụ:

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

{

**int** \*ptr; *//declare an int pointer*

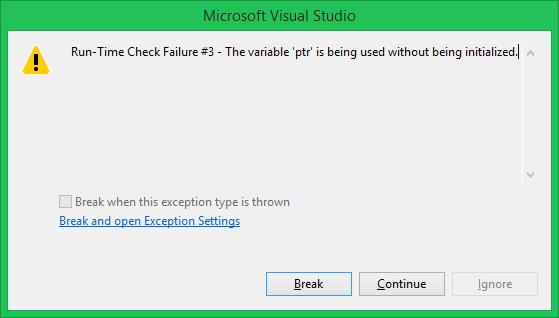
cout << \*ptr << endl;

system("pause");

**return** 0;

}

Nhấn F5 để Debug chương trình sẽ nhận được thông báo:



Do đó, khi khai báo con trỏ nhưng chưa có địa chỉ khởi tạo cụ thể, chúng ta nên gán cho con trỏ giá trị **NULL**.

### NULL

**NULL** là một macro đã được định nghĩa sẵn trong ngôn ngữ C/C++.

**#define NULL 0**

Đối với con trỏ, **NULL** là một giá trị đặc biệt, khi gán **NULL** cho con trỏ, điều đó có nghĩa là con trỏ đó chưa trỏ đến địa chỉ nào cả. Con trỏ đang giữ giá trị **NULL** được gọi là con trỏ **NULL** (**NULL pointer**).

int \***ptr** = **NULL**; //**ptr** **is** now a **NULL** pointer

Lúc này, chúng ta có thể kiểm tra xem con trỏ đã được gán địa chỉ cụ thể hay chưa:

**int** \*ptr = NULL;

**if**(ptr == NULL)

{

cout << "Do nothing" << endl;

}

**else**

{

cout << \*ptr << endl;

}

Đoạn chương trình này sẽ giúp chương trình đảm bảo rằng con trỏ **ptr** sẽ không được sử dụng khi chưa được gán địa chỉ cụ thể.

Chuẩn **C++11** cung cấp cho chúng ta từ khóa **nullptr** tương tự như macro **NULL**. Chúng ta có thể sử dụng như sau:

**int** \*ptr = nullptr;

Bên cạnh đó, **C++11** còn định nghĩa cho chúng ta kiểu dữ liệu std::nullptr\_t, nullptr\_t chỉ có thể lưu trữ giá trị **nullptr**. Nhưng chúng ta ít khi sử dụng kiểu dữ liệu này nên các bạn cũng không cần quan tâm lắm.

### Tổng kết

Trong bài học này, các bạn đã được tìm hiểu khái niệm con trỏ và một số khái niệm có liên quan. Việc sử dụng con trỏ thường có một số hoạt động chủ yếu: **(a) khai báo một con trỏ, (b) gán địa chỉ cho con trỏ, (c) truy cập đến địa chỉ mà con trỏ đang nắm giữ bằng dereference operator**.

Thử liên hệ một chút với cuộc sống thực tế, tưởng tượng rằng con đường nhà bạn (street) là bộ nhớ ảo, trên con đường đó có rất nhiều ngôi nhà (house), mỗi ngôi nhà đều được đánh số thứ tự gọi là địa chỉ nhà (house's address). Chúng ta tạm hình dung số người ở trong mỗi ngôi nhà (content) tương đương với nội dung của mỗi ô trên bộ nhớ ảo. Như vậy, address-of operator (&house) sẽ trả về địa chỉ của ngôi nhà, dereference operator (\*&house) sẽ lấy ra số lượng người bên trong ngôi nhà có địa chỉ được xác định. Để sử dụng con trỏ trỏ đến mỗi ngôi nhà, chúng ta phải sử dụng một con trỏ kiểu **House** (giống với kiểu của từng ngôi nhà), giả sử con trỏ kiểu **House** được khai báo là House \*h\_ptr; thì con trỏ h\_ptr có thể trỏ đến bất kì ngôi nhà nào trên con đường, và nó còn có thể thay đổi nội dung bên trong từng ngôi nhà mà nó trỏ đến.

Con trỏ (**Pointer**) là một công cụ mạnh mẽ đặc trưng của ngôn ngữ C/C++. Con trỏ cho phép chúng ta trực tiếp quản lý dung lượng của chương trình trên bộ nhớ ảo. Nhưng bên cạnh đó, việc sử dụng con trỏ không hợp lý có thể gây lãng phí tài nguyên của hệ thống máy tính. Chúng ta sẽ cùng tìm hiểu các kĩ thuật quản lý bộ nhớ ảo của chương trình trong các bài học tiếp theo.

# 8.1 Toán tử tăng, giảm dùng cho con trỏ

#### Chào các bạn học viên đang theo dõi khóa học lập trình trực tuyến ngôn ngữ C++.

Trong bài học trước, chúng ta tạm dừng sau khi tìm hiểu những khái niệm cơ bản nhất khi sử dụng con trỏ trong C/C++, vẫn còn rất nhiều thứ cần phải nói khi nhắc đến con trỏ.

**Một câu hỏi đặt ra là các phép toán khi sử dụng cho con trỏ có gì khác so với sử dụng các phép toán với các biến thông thường hay không?**

Về mặt bản chất, giá trị lưu trữ bên trong vùng nhớ của con trỏ là địa chỉ, địa chỉ của một biến (hoặc vùng nhớ) có kiểu unsigned int (số nguyên không dấu), do đó, chúng ta có thể thực hiện các phép toán trên con trỏ. Nhưng kết quả của các phép toán thực hiện trên con trỏ sẽ khác các phép toán số học thông thường về giá trị và cả ý nghĩa.

Ngôn ngữ C/C++ định nghĩa cho chúng ta 4 toán tử toán học có thể sử dụng cho con trỏ: **++, --, +, và -**.

Trước khi tìm hiểu về các toán tử toán học dùng cho con trỏ, chúng ta khai báo trước một biến thông thường và một biến con trỏ (có kiểu dữ liệu phù hợp để trỏ tới biến thông thường vừa được khai báo):

**int** **value** = 0;

**int** \*ptr = &**value**;

##### Increment operator (++)

Như các bạn đã được học, **increment operator (++)** được dùng để tăng giá trị bên trong vùng nhớ của biến lên 1 đơn vị. **Increment operator (++)** là toán tử một ngôi, có thể đặt trước tên biến, hoặc đặt sau tên biến.

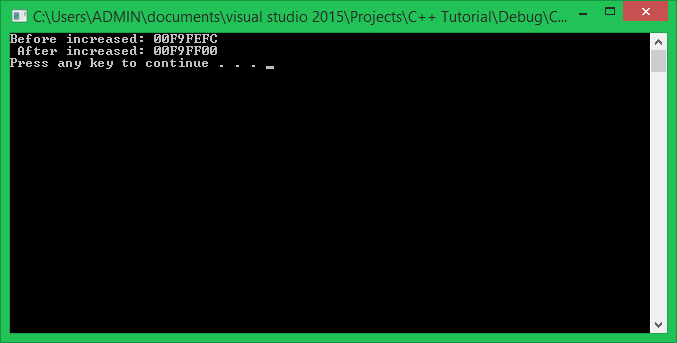
Bây giờ, chúng ta sử dụng toán tử **(++)** cho con trỏ ptr để xem kết quả:

cout << "Before increased: " << ptr << endl;

ptr++;

cout << " After increased: " << ptr << endl;

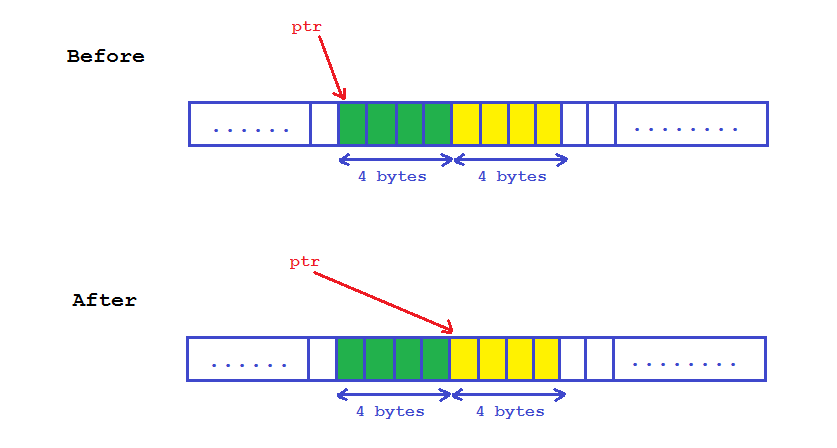
Kết quả:



* Before increased: **0x00F9FEFC** (heximal) tương đương **16383740** (decimal)
* After increased: **0x00F9FF00** (heximal) tương đương **16383744** (decimal)

Địa chỉ mới của ptr lúc này là **16383744**, giá trị này lớn hơn giá trị cũ **4** đơn vị. Đúng bằng kích thước của kiểu dữ liệu **int** mà mình dùng để khai báo cho biến **value**.

Như vậy, **increment operator (++)** sẽ làm con trỏ trỏ đến địa chỉ tiếp theo trên bộ nhớ ảo. Khoảng cách của 2 địa chỉ này đúng bằng kích thước của kiểu dữ liệu được khai báo cho con trỏ.

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-1-cac-toan-tu-su-dung-cho-con-tro/1.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-1-cac-toan-tu-su-dung-cho-con-tro/1.png?raw=true" \o "1.png?raw=true)**

**[1.png?raw=true837x448](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-1-cac-toan-tu-su-dung-cho-con-tro/1.png?raw=true" \o "1.png?raw=true)**

Giả sử cũng với địa chỉ ban đầu là **16383740**, nếu con trỏ được khai báo là char \*ptr; thì khi sử dụng toán tử (++), địa chỉ mới của con trỏ lúc này sẽ là **16383741**.

##### Decrement operator (--)

Ngược lại so với **increment operator (++), decrement operator (--)** sẽ giảm giá trị bên trong vùng nhớ của biến thông thường đi 1 đơn vị. Đối với biến con trỏ, khi sử dụng **decrement operator (--)**, nó sẽ làm thay đổi địa chỉ của con trỏ đang trỏ đến, giá trị địa chỉ mới sẽ bằng giá trị địa chỉ cũ trừ đi kích thước của kiểu dữ liệu mà con trỏ đang trỏ đến.

Để dễ hình dung, mình lấy lại ví dụ trên:

**int** value = 0;

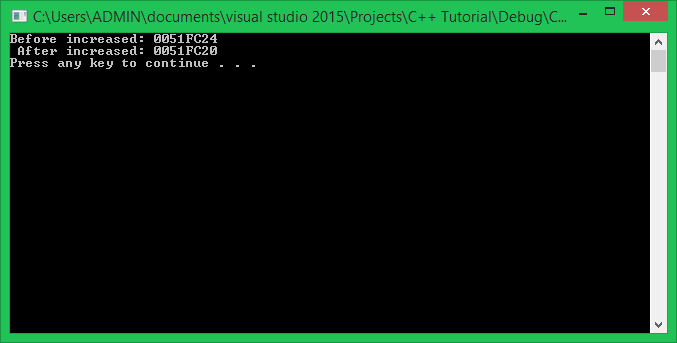
**int** \*ptr = &value;

cout << "Before decreased: " << ptr << endl;

ptr--;

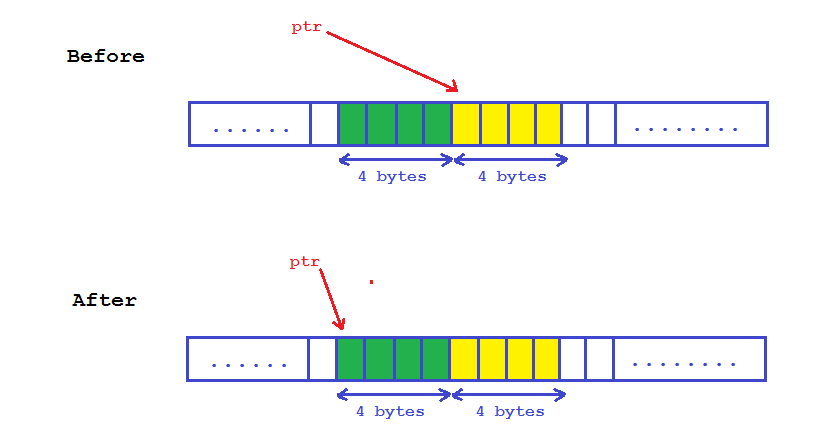
cout << " After decreased: " << ptr << endl;

Kết quả:



* Before increased: **0x0051FC24** (heximal) tương đương **5372964** (decimal)
* After increased: **0x0051FC20** (heximal) tương đương **5372960** (decimal)

Như chúng ta thấy, địa chỉ mới nhỏ hơn 4 (bytes) so với địa chỉ ban đầu, 4 bytes này chính là kích thước kiểu dữ liệu **int** mà con trỏ được khai báo.

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-1-cac-toan-tu-su-dung-cho-con-tro/3.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-1-cac-toan-tu-su-dung-cho-con-tro/3.png?raw=true" \o "3.png?raw=true)**

**[3.png?raw=true837x448](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-1-cac-toan-tu-su-dung-cho-con-tro/3.png?raw=true" \o "3.png?raw=true)**

Giả sử cũng với địa chỉ ban đầu là **5372964**, nếu con trỏ được khai báo double \*ptr; thì sau khi sử dụng toán tử **(--)**, địa chỉ mới của con trỏ sẽ là **5372956**.

##### Addition operator (+)

Sử dụng **increment operator (++)** cho con trỏ chỉ có thể làm con trỏ trỏ đến địa chỉ tiếp theo trên bộ nhớ ảo bắt đầu từ địa chỉ ban đầu mà con trỏ đang nắm giữ. Trong khi đó, toán tử **addition (+)** cho phép chúng ta trỏ đến vùng nhớ bất kỳ phía sau địa chỉ mà con trỏ đang nắm giữ.

Xét đoạn chương trình sau:

int value = 0*;*

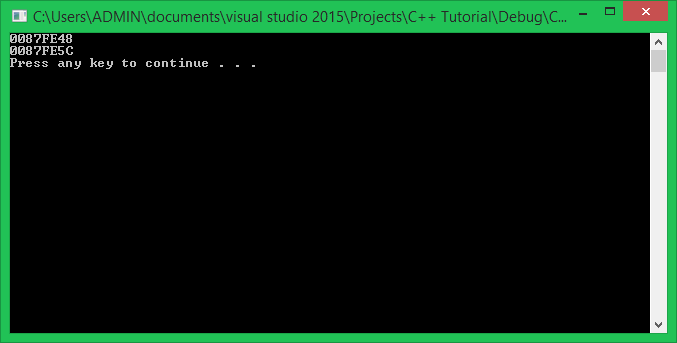
int \*ptr = &value*;*

cout << ptr << endl*;*

ptr = ptr + 5*;*

cout << ptr << endl*;*

Kết quả:



* Before added 5: **0x0087FE48** (heximal) tương đương **8912456**.
* After added 5: **0x0087FE5C** (heximal) tương đương **8912476**.

8912476 - 8912456 = 20 (bytes)

Như vậy, con trỏ **ptr** đã trỏ đến địa chỉ mới đứng sau địa chỉ ban đầu **20 bytes** (tương đương với **5 lần** kích thước kiểu **int**)

Chúng ta có thể sử dụng **dereference operator** để truy xuất trực tiếp giá trị bên trong các vùng nhớ ảo bất kỳ khi sử dụng toán tử **(+)**.

int value = 0*;*

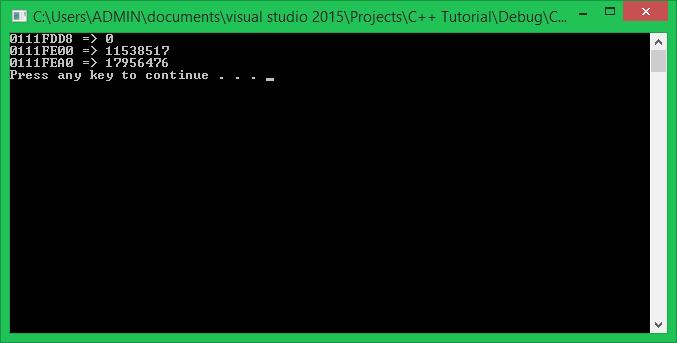
int \*ptr = &value*;*

cout << ptr << " => " << \*ptr << endl*;*

cout << ptr + 10 << " => " << \*(ptr + 10) << endl*;*

cout << ptr + 50 << " => " << \*(ptr + 50) << endl*;*

Kết quả của đoạn chương trình này là:



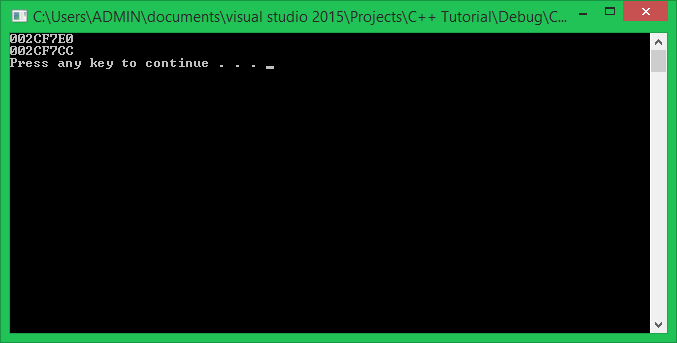
Giá trị 0 ban đầu là của biến value đang nắm giữ, những giá trị rác phía sau là của các vùng nhớ khác nắm giữ, chúng ta không cần thông qua tên biến nhưng vẫn có thể truy xuất giá trị của chúng thông qua **dereference operator**.

Những giá trị này có thể do chương trình khác đang sử dụng, nhưng những vùng nhớ này chưa được truy xuất bởi các chương trình khác hoặc không phải vùng nhớ hệ thống quan trọng, nên chương trình của chúng ta vẫn có thể truy xuất đến giá trị bên trong những địa chỉ này. **Nếu có 2 chương trình cùng truy cập đến một vùng nhớ, hệ thống sẽ xảy ra xung đột.**

**Lưu ý: Toán tử (+) chỉ cho phép thực hiện với số nguyên.**

##### Subtraction operator (-)

Ngược lại so với toán tử **(+)**.



* Before subtracted 5: **0x002CF7E0** (heximal) tương đương **2947040**
* After subtracted 5: **0x002CF7CC** (heximal) tương đương **2947020**

2947040 - 2947020 = 20 (bytes)

Như vậy, con trỏ **ptr** đã trỏ đến địa chỉ mới đứng trước địa chỉ ban đầu **20 bytes** (tương đương với **5 lần** kích thước kiểu **int**).

Chúng ta có thể sử dụng **dereference operator** để truy xuất trực tiếp giá trị bên trong các vùng nhớ ảo bất kỳ khi sử dụng toán tử **(-)**.

int value = 0*;*

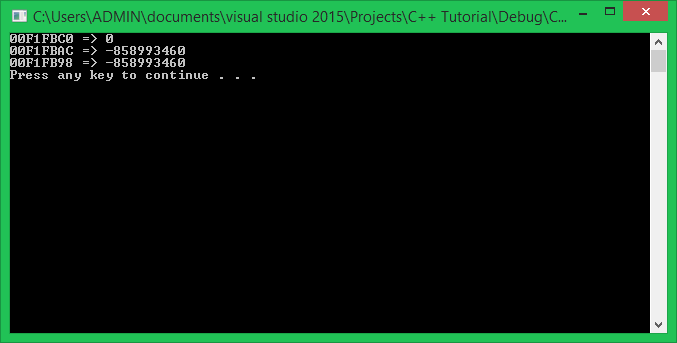
int \*ptr = &value*;*

cout << ptr << " => " << \*ptr << endl*;*

cout << ptr - 5 << " => " << \*(ptr - 5) << endl*;*

cout << ptr - 10 << " => " << \*(ptr - 10) << endl*;*

Kết quả của đoạn chương trình này là:



Giải thích tương tự khi sử dụng toán tử **(+)**.

**Lưu ý: Toán tử (-) chỉ cho phép thực hiện với số nguyên.**

##### So sánh hai con trỏ

Ngoài các toán tử toán học, chúng ta còn có thể áp dụng các toán tử quan hệ khi sử dụng con trỏ. Giả sử chúng ta khai báo 2 con trỏ p1 và p2 như sau:

int value1, value2;

int \*p1;

int \*p2;

p1 = &value1;

p2 = &value2;

Con trỏ **p1** trỏ đến value1 và con trỏ **p2** trỏ đến value2. Chúng ta thực hiện lần lượt 6 phép so sánh:

cout << "Is p1 less than p2? " << (p1 < p2) << endl;

cout << "Is p1 greater than p2? " << (p1 > p2) << endl;

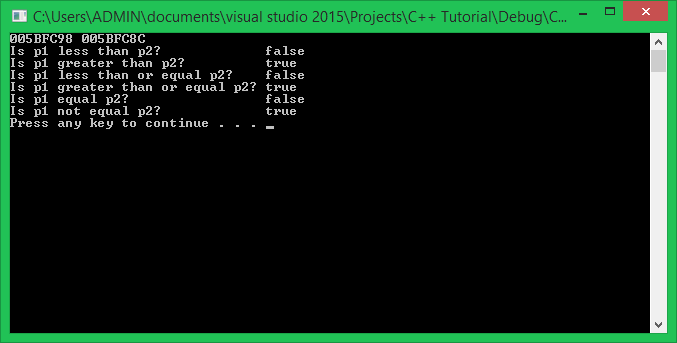
cout << "Is p1 less than or equal p2? " << (p1 <= p2) << endl;

cout << "Is p1 greater than or equal p2? " << (p1 >= p2) << endl;

cout << "Is p1 equal p2? " << (p1 == p2) << endl;

cout << "Is p1 not equal p2? " << (p1 != p2) << endl;

Kết quả chúng ta được như sau:



Trong đó, phép so sánh bằng **(==)** sẽ kiểm tra xem 2 con trỏ này có trỏ đến cùng một địa chỉ hay không.

##### Một số lưu ý khi sử dụng các toán tử dùng cho con trỏ

Vì các toán tử dùng cho con trỏ có ý nghĩa hoàn toàn khác so với việc áp dụng các toán tử lên giá trị hoặc biến thông thường. Chúng ta cần có cách sử dụng hợp lý để tránh gây nhầm lẫn hoặc gây rối mắt.

Lấy đoạn chương trình sau để làm ví dụ:

int n = 5;

int \*p = &n; //p point to n

\*p++;

p++;

int n2 = \*p\*n;

Đây là một số cách sử dụng các toán tử toán học cho con trỏ gây khó hiểu cho người đọc.

* Lệnh \*p++; sẽ thực hiện hai bước, đầu tiên là sử dụng toán tử **dereference** để truy xuất đến vùng nhớ tại địa chỉ mà con trỏ p đang nắm giữ, bước thứ hai là trỏ đến địa chỉ tiếp theo (đứng sau n).
* Sau đó, chúng ta bắt gặp lệnh p++; có nghĩa là cho con trỏ p trỏ đến địa chỉ tiếp theo lớn hơn địa chỉ ban đầu **4 bytes** (kích thước của kiểu **int**).
* Dòng cuối cùng, chúng ta có phép gán giá trị của phép nhân \*p và n cho biến n2.

Để chương trình rõ ràng hơn, chúng ta nên thêm các cặp dấu ngoặc vào chương trình tương tự như thế này:

int n = 5;

int \*p = &n; //p point to n

(\*p)++;

p++;

int n2 = (\*p) \* n;

Những cặp dấu ngoặc sẽ giúp phân biệt lúc nào chúng ta sử dụng giá trị là địa chỉ lưu trong con trỏ, lúc nào chúng ta sử dụng giá trị trong vùng nhớ mà con trỏ đang trỏ đến.

### Tổng kết

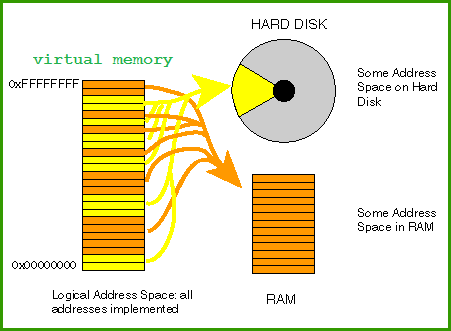
Việc sử dụng toán các toán tử toán học cho biến con trỏ mà không có mục đích rõ ràng có thể gây xung đột vùng nhớ, có thể dẫn đến crash chương trình. Chúng ta thường sử dụng các toán tử toán học khi con trỏ trỏ đến mảng một chiều, vì mảng một chiều lưu trữ trên bộ nhớ ảo là một vùng nhớ mà những phần tử có địa chỉ liên tiếp nhau. Chúng ta sẽ tìm hiểu vấn đề này trong các bài học sau.

# 8.2 Con trỏ và mảng một chiều

#### Chào các bạn đang theo dõi khóa học lập trình trực tuyến ngôn ngữ C++.

Qua một số bài học tìm hiểu về khái niệm và cách sử dụng của con trỏ trong ngôn ngữ C/C++, chúng ta biết rằng chức năng của con trỏ là để lưu trữ một địa chỉ của một vùng nhớ trên bộ nhớ ảo (**virtual memory**), tận dụng sức mạnh của con trỏ chúng ta có thể dùng nó để quản lý vùng nhớ tại địa chỉ mà con trỏ đang giữ, kích thước vùng nhớ đó là bao nhiêu còn tùy thuộc vào kiểu dữ liệu chúng ta khai báo cho con trỏ.

Trước khi vào phần trọng tâm bài học, chúng ta cùng xem lại một chút về khái niệm **virtual memory**. **Virtual memory** là một kĩ thuật quản lý bộ nhớ được thực hiện bởi cả phần cứng lẫn phần mềm trên máy tính chúng ta đang sử dụng. Mục đích của việc sử dụng kỹ thuật này là tổ chức các vùng bộ nhớ có thể sử dụng được trên các thiết bị lưu trữ (RAM, Hard disk drive, ...) thành một dãy địa chỉ ảo liên tiếp nhau **từ 0x00000000 (0) đến 0xFFFFFFFF (4294967295)** (giả sử mình đang xét trên hệ điều hành nền tảng 32 bits).



Khi thao tác với **virtual memory** chúng ta sẽ có cảm giác như đang làm việc với những vùng nhớ có dãy địa chỉ liên tục nhau. Và với con trỏ trong ngôn ngữ C/C++, chúng ta có thể làm việc trực tiếp với các vùng nhớ trên bộ nhớ ảo.

Các bạn có thấy cấu trúc tổ chức lưu trữ của **virtual memory** giống với cấu trúc dữ liệu nào mà chúng ta đã cùng tìm hiểu không? Đó chính là **mảng một chiều**.

Mảng một chiều là tập hợp các phần tử có cùng kiểu dữ liệu được lưu trữ liên tiếp nhau trên bộ nhớ ảo, nếu mảng một chiều có một hoặc nhiều hơn một phần tử, địa chỉ của phần tử đầu tiên cũng chính là địa chỉ của mảng một chiều.

Như mình đã nói ở trên, con trỏ trong ngôn ngữ C/C++ có thể thao tác trực tiếp với bộ nhớ ảo, vậy thì chúng ta cũng có thể sử dụng con trỏ để thao tác trực tiếp với mảng một chiều.

##### Địa chỉ của mảng một chiều và các phần tử trong mảng một chiều

Mình lấy một ví dụ về mảng một chiều được khai báo với 5 phần tử:

int arr[] = { 32, 13, 66, 11, 22 };

Như chúng ta đã biết, địa chỉ của mảng một chiều cũng là địa chỉ của phần tử đầu tiên, vì thế, đoạn chương trình bên dưới sẽ in ra 2 giá trị giống nhau:

//**show** address **of** arr **in** **virtual** **memory**

cout << &arr << endl;

//**show** address **of** the **first** **element** **of** arr

cout << &arr[0] << endl;

Có một điểm đặc biệt của mảng một chiều trong C/C++, đoạn chương trình sau sẽ cho thấy điều đó:

*//show address of arr in virtual memory*

cout << &arr << endl;

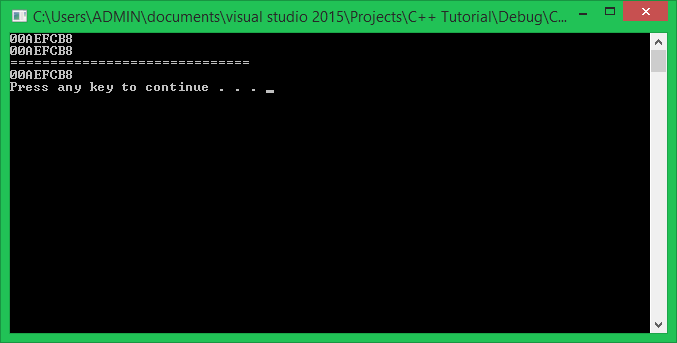
*//show address of the first element of arr*

cout << &arr[0] << endl;

cout << "==============================" << endl;

cout << arr << endl;

Kết quả ghi nhận được trên máy tính của mình:



Điều này chứng tỏ rằng việc sử dụng tên mảng một chiều cũng đồng nghĩa đang sử dụng địa chỉ của mảng một chiều (&arr tương đương với arr). Vì thế, chúng ta có thể in ra địa chỉ của cả 5 phần tử của mảng arr bằng cách sau:

cout << arr << enld;

cout << arr + 1 << endl;

cout << arr + 2 << endl;

cout << arr + 3 << endl;

cout << arr + 4 << endl;

Mảng arr là một tập hợp các phần tử số nguyên được cấp phát địa chỉ liên tiếp nhau trên bộ nhớ ảo. Các bạn cũng đã biết, sử dụng toán tử **address-of** sẽ trả về giá trị kiểu con trỏ, mình sử dụng toán tử **(+)** cho con mảng arr sẽ lấy được địa chỉ của các phần tử đứng sau phần tử đầu tiên của mảng arr.

Với những địa chỉ này, chúng ta cũng có thể sử dụng toán tử **dereference** để truy xuất giá trị của chúng:

cout << \*(arr) << enld*;*

cout << \*(arr + 1) << endl*;*

cout << \*(arr + 2) << endl*;*

cout << \*(arr + 3) << endl*;*

cout << \*(arr + 4) << endl*;*

##### Con trỏ trỏ đến mảng một chiều

Mình lấy lại ví dụ mảng một chiều có 5 phần tử kiểu **int** giống như trên:

int arr[] = { 3, 5, 65, 23, 11 };

Vì mỗi phần tử bên trong mảng đều có kiểu **int**, do đó, chúng ta có thể sử dụng 1 con trỏ có kiểu dữ liệu tương ứng (int \*) để trỏ đến từng phần tử của mảng **arr**.

**int** \*ptr = &arr[2]; *//ptr point to the 3rd element*

Mình cho con trỏ **ptr** trỏ đến phần tử có chỉ số là 2 trong mảng **arr**.

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-2-con-tro-va-mang-mot-chieu/2.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-2-con-tro-va-mang-mot-chieu/2.png?raw=true" \o "2.png?raw=true)**

**[2.png?raw=true938x360](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-2-con-tro-va-mang-mot-chieu/2.png?raw=true" \o "2.png?raw=true)**

Lúc này, chúng ta sử dụng **dereference operator** để truy xuất giá trị của **ptr** sẽ được giá trị **65**.

cout << \*ptr << endl;

Từ địa chỉ của arr[2] mà con trỏ ptr đang nắm giữ, chúng ta cũng có thể sử dụng toán tử **(+)** hoặc **(-)** để truy xuất đến tất cả các phần tử còn lại trong mảng **arr** vì các phần tử của mảng có địa chỉ nối tiếp nhau trên bộ nhớ ảo.

cout << \*(ptr - 1) << endl; *//access the second element of arr*

cout << \*(ptr + 2) << endl; *//access the last element of arr*

Chúng ta cũng có thể sử dụng toán tử **(++)** hoặc **(--)** để cho con trỏ ptr trỏ đến phần tử tiếp theo hoặc phần tử đứng trước đó:

**ptr**++;

cout << \***ptr** << endl; //**ptr** **is** now point to &arr[3]

Như các bạn thấy, chỉ với một con trỏ có kiểu dữ liệu tương ứng với kiểu của mảng một chiều, chúng ta có thể quản lý được toàn bộ phần tử trong mảng:

**for** (**ptr** = &arr[0]; **ptr** <= &arr[4]; **ptr**++)

{

cout << \***ptr** << " ";

}

Vòng lặp **for** ở ví dụ trên ban đầu khởi tạo bằng cách gán địa chỉ phần tử đầu tiên của mảng **arr** cho con trỏ **ptr**, khi nào địa chỉ mà **ptr** nắm giữ vẫn còn nhỏ hơn hoặc bằng địa chỉ của phần tử cuối cùng thì tiếp tục in giá trị mà **ptr** trỏ đến, cuối vòng lặp là cho **ptr** trỏ đến phần tử tiếp theo trong mảng.

Chúng ta có thể thay phép gán ptr = &arr[0]; bằng phép gán ptr = &arr; hoặc ngắn gọn hơn là ptr = arr; vì **&arr[0], &arr hoặc arr** đều cho chúng ta địa chỉ của phần tử đầu tiên trong mảng **arr**.

Vì thế, chúng ta có thể viết lại như sau:

**for** (**ptr** = arr; **ptr** <= arr + 4; **ptr**++)

{

cout << \***ptr** << " ";

}

Cũng là in ra toàn bộ giá trị của các phần tử trong mảng **arr**, nhưng sử dụng con trỏ chúng ta có rất nhiều cách viết khác nhau:

int \***ptr** = arr; //**ptr** point to &arr[0]

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

{

cout << \*(**ptr** + i) << " ";

}

Chúng ta có thể sử dụng **dereference operator** để truy xuất giá trị của từng phần tử thông qua tên của mảng:

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

{

cout << \*(arr + i) << " ";

}

Sau khi cho con trỏ trỏ đến mảng một chiều, chúng ta còn có thể sử dụng toán tử **[]** cho con trỏ để truy xuất đến các phần tử thay vì dùng tên mảng:

int \*ptr = arr*;*

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

{

cout << ptr[i] << " "*;*

}

Giả sử chúng ta có 2 mảng một chiều kiểu **int** có cùng kích thước như sau:

int src[5] = { 3, 1, 5, 7, 4 };

int des[5];

Việc copy dữ liệu từ mảng src sang mảng des có thể thực hiện được bằng 2 con trỏ:

**int** \*p\_src = src;

**int** \*p\_des = des;

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

{

\*(p\_des + i) = \*(p\_src + i);

}

Đối với mảng kí tự (**C-style string**), chúng ta có thể trực tiếp in nội dung của chuỗi kí tự sử dụng đối tượng **cout**. Ví dụ:

char my\_name[50]*;*

cout << "Enter your name: "*;*

gets\_s(my\_name)*;*

cout << "Hello " << my\_name << endl*;*

Như vậy chúng ta chỉ cần cung cấp cho đối tượng **cout** địa chỉ của mảng kí tự my\_name, toàn bộ nội dung của mảng kí tự my\_name sẽ được in ra màn hình. Và nếu chúng ta sử dụng một con trỏ kiểu (char \*) để trỏ đến mảng my\_name, chúng ta có thể dùng tên con trỏ để in mảng đó ra màn hình:

**char** \*p\_name = my\_name;

cout << "Hello " << p\_name << endl;

Bên cạnh đó, chúng ta có thể cho con trỏ kiểu (char \*) trỏ đến một chuỗi kí tự cố định nào đó, và vẫn có thể sử dụng đối tượng cout để in nội dung mà con trỏ đó đang trỏ đến. Ví dụ:

**char** \*p\_str = "This is an example string";

cout << p\_str << endl;

Nhưng vùng nhớ của chuỗi kí tự này được xem là hằng số (const) nên chúng ta chỉ có thể xem nội dung mà p\_str trỏ đến chứ không thể thay đổi kí tự bên trong chuỗi. Chúng ta sẽ tìm hiểu về vấn đề này trong các bài học tiếp theo.

##### Sự khác nhau khi sử dụng mảng một chiều và con trỏ trỏ đến mảng một chiều

Sau khi con trỏ trỏ đến mảng một chiều, chúng ta có thể sử dụng tên con trỏ thay vì sử dụng tên mảng. Tuy vậy, giữa chúng vẫn có một số điểm khác biệt. Dễ nhận thấy nhất là khi sử dụng toán tử sizeof(). Ví dụ:

**int** arr[5];

**int** \*ptr = arr;

cout << "Size of arr: " << **sizeof**(arr) << endl;

cout << "Size of ptr: " << **sizeof**(ptr) << endl;

Debug đoạn chương trình này trên nền tảng 32 bits chúng ta thu được kết quả:

**Size** **of** arr: 20

**Size** **of** **ptr**: 4

Khi sử dụng mảng một chiều, toán tử sizeof trả về kích thước của toàn bộ phần tử bên trong mảng. Trong khi đó, con trỏ sau khi trỏ đến mảng một chiều vẫn có kích thước 4 bytes (trên hệ điều hành 32 bits) như cũ.

Như vậy, sử dụng mảng một chiều chúng ta có thể biết được chính xác số lượng phần tử chúng ta cần quản lý trong khi con trỏ không làm được điều này.

Ngoài ra, mảng một chiều sau khi khai báo có địa chỉ cố định trên bộ nhớ ảo, con trỏ sau khi trỏ đến mảng một chiều vẫn có thể được trỏ đi nơi khác.

# 8.3 Con trỏ và mảng kí tự

#### Rất vui khi nhận được sự quan tâm theo dõi của các bạn trong khóa học lập trình trực tuyến ngôn ngữ C++.

Trong bài học này, chúng ta sẽ cùng nhau tìm hiểu một số điểm cần lưu ý khi sử dụng con trỏ trỏ đến mảng kí tự (**C-style string**).

##### C-style string symbolic constants

C-style string là một trường hợp đặc biệt của mảng một chiều, được ngôn ngữ C++ hổ trợ một số đặc điểm nhằm giúp lập trình viên thao tác với C-style string một cách thuận tiện hơn.

Ngoài cách khởi tạo mảng một chiều thông thường, C-style string còn có thể khởi tạo bằng một hằng chuỗi kí tự như sau:

char my\_name[] = "Le Tran Dat"*;*

Chuỗi kí tự "Le Tran Dat" được xem như là một chuỗi hằng kí tự, nó có địa chỉ cụ thể trên bộ nhớ ảo, nó được lưu trên bộ nhớ ảo, nhưng không có tên biến để truy xuất đến địa chỉ của chuỗi hằng kí tự này. Nhưng sau khi sử dụng chuỗi hằng kí tự "Le Tran Dat" để khởi tạo cho mảng my\_name, mảng my\_name không được khai báo là kiểu chuỗi hằng kí tự (const char []) nên các kí tự trong mảng my\_name hoàn toàn có thể bị thay đổi.

Ví dụ:

char my\_name[] = "Le Tran Dat";

my\_name[1] = 'E'; //=> "LE Tran Dat"

Điều này chứng tỏ mảng my\_name được cấp phát bộ nhớ tại địa chỉ khác chuỗi hằng kí tự "Le Tran Dat", việc khởi tạo mảng kí tự bằng một chuỗi hằng kí tự chỉ đơn giản là copy từng kí tự của chuỗi "Le Tran Dat" và đưa vào mảng.

Do đó, con trỏ kiểu char (char \*) trỏ đến mảng my\_name và trỏ đến vùng nhớ của chuỗi hằng kí tự "Le Tran Dat" là 2 trường hợp khác nhau.

Mình lấy ví dụ một con trỏ kiểu char (char \*) trỏ đến mảng my\_name:

**char** my\_name[] = "Le Tran Dat";

**char** \*p\_name = my\_name;

p\_name[1] = 'E';

cout << my\_name << endl;

Kết quả in ra màn hình là:

LE Tran Dat

Như vậy, con trỏ p\_name sau khi trỏ đến mảng my\_name thì có thể thay đổi giá trị bên trong vùng nhớ mà mảng my\_name đang nắm giữ, vì vùng nhớ này không phải là vùng nhớ hằng.

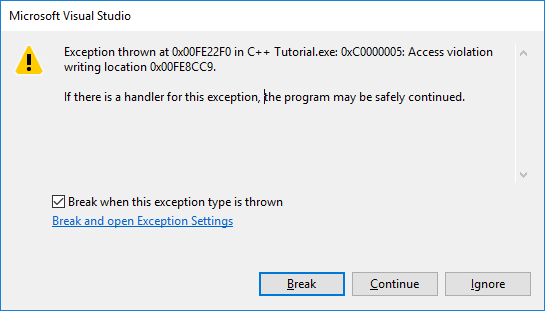
Trường hợp tiếp theo, mình sẽ cho một con trỏ kiểu char (char \*) trỏ trực tiếp đến chuỗi hằng kí tự:

**char** \*p\_name = "Le Tran Dat";

p\_name[1] = 'E';

cout << p\_name << endl;

Khi nhấn F5 để Debug đoạn chương trình này, Visual studio 2015 đưa ra thông báo xảy ra xung đột vùng nhớ.



Nguyên nhân là do vùng nhớ lưu trữ chuỗi kí tự "Le Tran Dat" là vùng nhớ hằng, giá trị bên trong vùng nhớ này không thể thay đổi, trong khi đó lệnh p\_name[1] = 'E'; cố gắng thay đổi giá trị bên trong vùng nhớ hằng.

Đến đây có thể có một số bạn thắc mắc về địa chỉ của chuỗi hằng kí tự "Le Tran Dat" mà mình sử dụng. Mặc dù chuỗi hằng kí tự không được khai báo như một biến thông thường, nhưng nó được tạo ra và có địa chỉ cụ thể trên vùng nhớ ảo. Chúng ta truy xuất địa chỉ của chuỗi hằng kí tự bằng chính nội dung của chuỗi đó:

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

{

cout << &("Le Tran Dat") << endl;

cout << &("LE TRAN DAT") << endl;

system("pause");

**return** 0;

}

Kết quả của đoạn chương trình này trên máy tính của mình là:

00EF8CC8

00EF8B30

Như vậy, mỗi chuỗi hằng kí tự có nội dung khác nhau sẽ có một địa chỉ khác nhau. Chúng ta có thể sử dụng nội dung của chuỗi hằng kí tự này như mảng một chiều, nhưng không thể thay đổi nội dung của nó.

**for** (**int** i = 0; i < strlen("Le Tran Dat"); i++)

{

cout << "Le Tran Dat"[i];

}

cout << endl;

"Le Tran Dat"[1] = 'E'; *//this line will make an error*

##### std::cout and char pointers

Với các mảng một chiều có kiểu dữ liệu khác, để xem được nội dung bên trong mảng, chúng ta cần sử dụng vòng lặp để duyệt từng phần tử bên trong mảng. Ví dụ:

**float** arr[] = { 2.5, 1.6, 0.2, 3.14 };

**int** size = **sizeof**(arr) / **sizeof**(arr[0]);

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

{

cout << arr[i] << " ";

}

Đối với mảng kí tự (**C-style string**) chúng ta có thể in toàn bộ nội dung của mảng bằng cách sử dụng đối tượng **cout** như sau:

**char** str[] = "This is an example string";

cout << str << endl;

Đối với các kiểu dữ liệu không phải kiểu con trỏ char (char \*), đối tượng **cout** chỉ in ra địa chỉ của mảng (vì arr tương đương với &arr), nhưng với kiểu con trỏ char (char \*), đối tượng cout có cách định nghĩa khác.

Thực ra đối tượng **cout** chỉ hổ trợ cho kiểu con trỏ char (char \*), nhưng vì sử dụng tên mảng str tương đương với &str. Như các bạn biết, toán tử **address-of** trả về kiểu con trỏ, nên str truyền vào đối tượng **cout** được xem là con trỏ kiểu char (char \*).

**char** str[] = "Hello!";

**char** \*p\_str = str;

cout << str << endl;

cout << p\_str << endl;

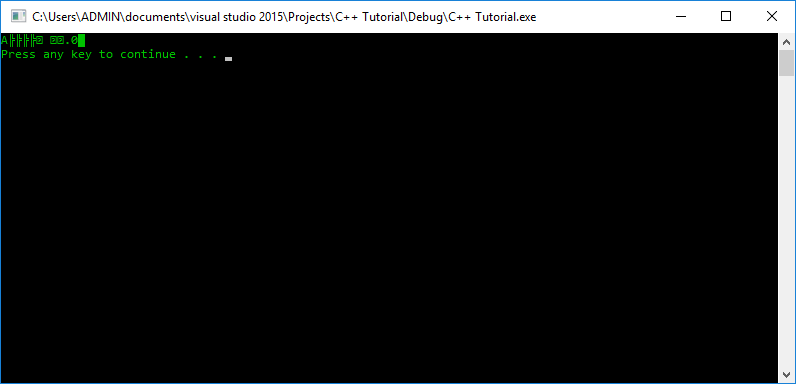
Do đó, đoạn chương trình này in ra 2 dòng có nội dung giống nhau.

Điều này dẫn để một hệ quả, chúng ta không thể in ra địa chỉ của một biến kiểu kí tự (char).

**char** ch = 'A';

cout << &ch << endl;

Trên máy tính của mình, kết quả cho ra màn hình là:

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-3-con-tro-va-mang-ki-tu/1.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-3-con-tro-va-mang-ki-tu/1.png?raw=true" \o "1.png?raw=true)**

**[1.png?raw=true796x384](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-3-con-tro-va-mang-ki-tu/1.png?raw=true" \o "1.png?raw=true)**

Vì &ch trả về dữ liệu kiểu (char \*) nên đối tượng **cout** xem nó như là **C-style string** nên in ra kí tự A và tiếp tục cho đến khi gặp giá trị '\0'.

# 8.4 Cấp phát bộ nhớ động

#### Chào các bạn học viên đang theo dõi khóa học lập trình trực tuyến ngôn ngữ C++.

Trong bài học này, mình sẽ tiếp tục giới thiệu đến các bạn một số vấn đề về con trỏ và sử dụng con trỏ để quản lý bộ nhớ ảo trong ngôn ngữ C++.

Như mình đã đề cập trong bài học [**phạm vi của biến**](https://daynhauhoc.com/t/pham-vi-cua-bien/29939), thời gian tồn tại của biến phụ thuộc vào vị trí bạn khai báo biến.

* Biến toàn cục (**global variable**) được khai báo bên ngoài khối lệnh, có thể được truy xuất tại bất cứ dòng lệnh nào đặt bên dưới biến đó. Biến toàn cục tồn tại đến khi chương trình bị kết thúc.
* Biến cục bộ (**local variable**) được khai báo bên trong khối lệnh, có thể được truy xuất tại bất cứ dòng lệnh nào đặt bên dưới biến đó và trong cùng khối lệnh. Biến cục bộ bị hủy khi chương trình chạy ra ngoài khối lệnh chứa biến đó.

Tương ứng với 2 kiểu khai báo biến này là 2 cách thức cấp phát bộ nhớ cho chương trình trên bộ nhớ ảo:

##### Static memory allocation (cấp phát bộ nhớ tĩnh)

**Static memory allocation** còn được gọi là **Compile-time allocation**, được áp dụng cho biến **static** và biến toàn cục.

* Vùng nhớ của các biến này được cấp phát ngay khi chạy chương trình.
* Kích thước của vùng nhớ được cấp phát phải được cung cấp tại thời điểm biên dịch chương trình.
* Đối với việc khai báo mảng một chiều, đây là lý do tại sao số lượng phần tử là hằng số.

##### Automatic memory allocation (cấp phát bộ nhớ tự động)

**Automatic memory allocation** được sử dụng để cấp phát vùng nhớ cho các biến cục bộ, tham số của hàm.

* Bộ nhớ được cấp phát tại thời điểm chương trình đang chạy, khi chương trình đi vào một khối lệnh.
* Các vùng nhớ được cấp phát sẽ được thu hồi khi chương trình đi ra khỏi một khối lệnh.
* Kích thước vùng cần cấp phát cũng phải được cung cấp rõ ràng.

##### Nhược điểm của các phương thức cấp phát bộ nhớ đã học

**Kích thước vùng nhớ cấp phát phải được cung cấp tại thời điểm biên dịch chương trình**

Lấy ví dụ, chúng ta cần lưu trữ tên của tất cả sinh viên trong một lớp học. Chúng ta sẽ sử dụng một mảng các string để lưu trữ như sau:

string name\_of\_students[50];

Mình hiện tại không biết có bao nhiêu sinh viên trong một lớp học, nên mình chỉ ước tính con số tối đa lượng sinh viên của lớp này là 50 người. Vậy điều gì xảy ra khi lớp học có nhiều hơn 50 sinh viên? Mảng name\_of\_students sẽ không thể lưu hết tên của tất cả sinh viên được. Bên cạnh đó, nếu số lượng sinh viên của lớp học chỉ có 30 người, mảng name\_of\_students sẽ thừa ra 20 phần tử không cần sử dụng đến.

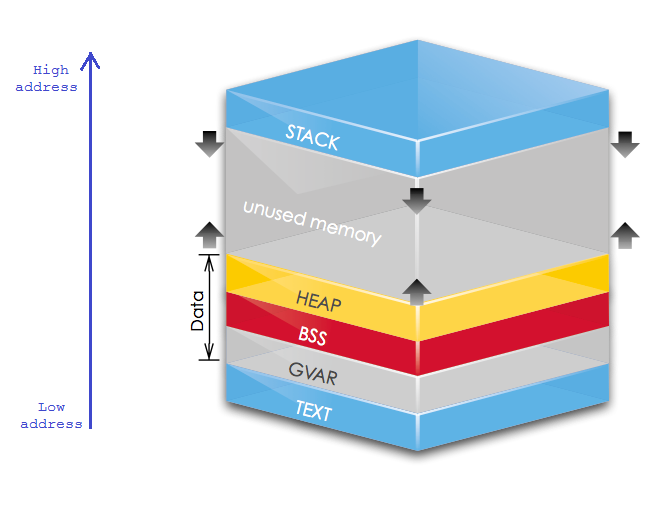
**Cấp phát và thu hồi vùng nhớ do chương trình quyết định**

Trong một số trường hợp, chúng ta cần sử dụng biến toàn cục để có thể truy cập vùng nhớ của biến tại nhiều khối lệnh khác nhau trong chương trình, nhưng thời gian tồn tại của biến toàn cục khá lâu, nên khi sử dụng biến toàn cục sẽ gây ảnh hưởng đáng kể lượng tài nguyên bộ nhớ của máy tính nếu chúng ta cấp phát cho biến toàn cục một vùng nhớ lớn.

Hoặc trong một số trường hợp khác, chúng ta vẫn muốn sử dụng tiếp vùng nhớ cấp phát cho biến bên trong hàm, nhưng biến cục bộ đặt trong khối lệnh (cùng với vùng nhớ nó quản lý) sẽ bị hủy khi hàm kết thúc.

**Kích thước bộ nhớ dùng cho Static memory allocation và Automatic memory allocation bị giới hạn**

Bộ nhớ ảo được chia thành nhiều phân vùng khác nhau sử dụng cho những loại tài nguyên khác nhau. Trong đó, các phương thức cấp phát bộ nhớ **Static memory allocation** hay **Automatic memory allocation** sẽ sử dụng phân vùng **Stack** để lưu trữ. Chúng ta sẽ có một bài học để nói chi tiết về các phân vùng trên bộ nhớ ảo. Bây giờ các bạn tạm thời hình dung bộ nhớ ảo chúng ta sẽ chia thành các phần như sau:

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-4-cap-phat-dong/0.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-4-cap-phat-dong/0.png?raw=true" \o "0.png?raw=true)**

**[0.png?raw=true670x523](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-4-cap-phat-dong/0.png?raw=true" \o "0.png?raw=true)**

Phân vùng **Stack** được đặt tại vùng có địa chỉ cao nhất trong dãy bộ nhớ ảo. Dung lượng của phân vùng này khá hạn chế. Tùy vào mỗi hệ điều hành mà dung lượng bộ nhớ của phân vùng **Stack** khác nhau. Đối với Visual studio 2015 chạy trên hệ điều hành Windows, dung lượng bộ nhớ của phân vùng **Stack** là khoảng 1MB (tương đương khoảng 1024 Kilobytes hay 1024\*1024 bytes).

Với sự hạn chế về dung lượng bộ nhớ của phân vùng **Stack**, chương trình của chúng ta sẽ phát sinh lỗi **stack overflow** nếu các bạn yêu cầu cấp phát vùng nhớ vượt quá dung lượng của **Stack**. Các bạn có thể chạy thử 2 đoạn chương trình sau để kiểm chứng:

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

{

**char** ch\_array[1024 \* 1000];

system("pause");

**return** 0;

}

Trong đoạn chương trình trên, mình khai báo một mảng kí tự có tên ch\_array, như các bạn biết kiểu char có kích thước 1 byte cho mỗi biến đơn (tương ứng với mỗi phần tử trong mảng kí tự), 1024 bytes sẽ tương ứng với 1Kb (Kilobyte). Do ch\_array là biến cục bộ, nó sẽ được cấp phát vùng nhớ trên phân vùng Stack của bộ nhớ ảo. Như vậy, mảng ch\_array sẽ được cấp phát 1000 kilobytes trên phân vùng Stack, nhưng con số này vẫn chưa vượt quá giới hạn 1Mb (1 Megabyte = 1024 Kilobytes) nên chương trình vẫn chạy bình thường. Bây giờ các bạn thử lại với đoạn chương trình sau:

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

{

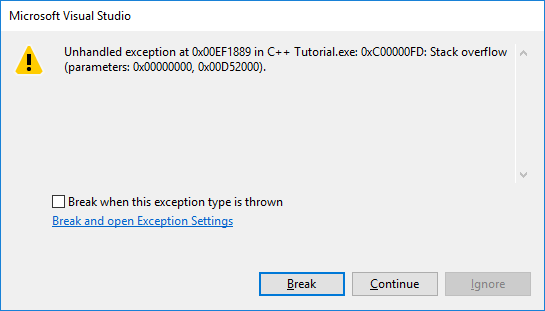
**char** ch\_array[1024 \* 1024];

system("pause");

**return** 0;

}

Kích thước vùng nhớ được yêu cầu cấp phát bây giờ là đúng bằng **1 Mb**. Thử chạy chương trình ở chế độ Debug, Visual Studio 2015 trên máy tính mình đưa ra thông báo:

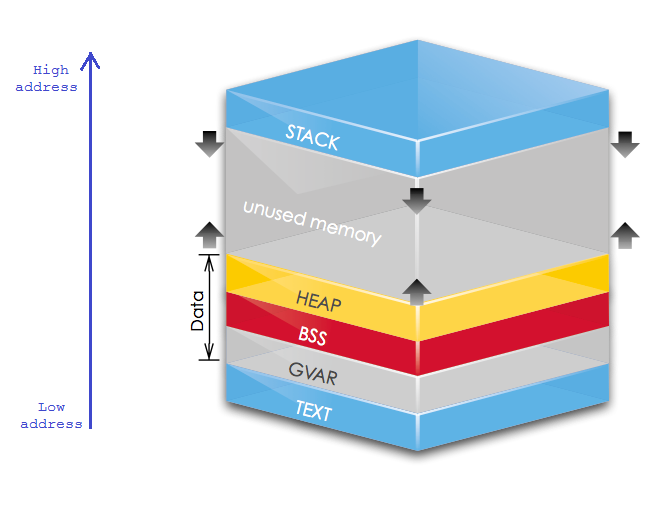


Việc cấp phát vùng nhớ có kích thước 1 Mb đã gây tràn bộ nhớ phân vùng Stack.

Đây là một số hạn chế của các phương thức cấp phát bộ nhớ **Static memory allocation và Automatic memory allocation**. Để khắc phục hạn chế này, mình giới thiệu đến các bạn một phương thức cấp phát bộ nhớ mới được ngôn ngữ C++ hổ trợ.

### Dynamic memory allocation

**Dynamic memory allocation** là một giải pháp cấp phát bộ nhớ cho chương trình tại thời điểm chương trình đang chạy (run-time). **Dynamic memory allocation** sử dụng phân vùng **Heap** trên bộ nhớ ảo để cấp phát cho chương trình.

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-4-cap-phat-dong/0.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-4-cap-phat-dong/0.png?raw=true" \o "0.png?raw=true)**

**[0.png?raw=true670x523](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-4-cap-phat-dong/0.png?raw=true" \o "0.png?raw=true)**

Như các bạn thấy trong hình trên, phân vùng **Heap** của bộ nhớ ảo có dung lượng bộ nhớ lớn nhất. Do đó, bộ nhớ dùng để cấp phát cho chương trình trên phân vùng Heap chỉ bị giới hạn bởi thiết bị phần cứng (ví dụ là RAM) chứ không phụ thuộc vào hệ điều hành. Trong các máy tính hiện đại ngày nay, dung lượng bộ nhớ của phân vùng **Heap** có thể lên đến đơn vị GB (1 Gigabyte = 1024 Megabytes = 1024 \* 1024 Kilobytes).

##### Đọc kỹ hướng dẫn sử dụng trước khi dùng

Kỹ thuật **Dynamic memory allocation** dùng để cấp phát bộ nhớ tại thời điểm run-time. Tại thời điểm này, chúng ta không thể tạo ra tên biến mới, mà chỉ có thể tạo ra vùng nhớ mới. Do đó, cách duy nhất để kiểm soát được những vùng nhớ được cấp phát bằng kỹ thuật **Dynamic memory allocation** là sử dụng con trỏ lưu trữ địa chỉ đầu tiên của vùng nhớ được cấp phát, thông qua con trỏ để quản lý vùng nhớ trên **Heap**.

Vậy, việc thực hiện cấp phát bộ nhớ cần thực hiện qua 2 bước:

* Yêu cầu cấp phát vùng nhớ trên **Heap**.
* Lưu trữ địa chỉ của vùng nhớ vừa được cấp phát bằng con trỏ.

**Để yêu cầu cấp phát bộ nhớ trên Heap, chúng ta sử dụng new operator.**

Vùng nhớ được cấp phát trên Heap sẽ không tự động hủy bởi chương trình khi kết thúc khối lệnh, việc thu hồi vùng nhớ đã cấp phát trên Heap được giao cho lập trình viên tự quản lý. Nếu trong chương trình có yêu cầu cấp phát bộ nhớ trên Heap mà không được thu hồi hợp lý sẽ gây lãng phí tài nguyên hệ thống. Cũng giống như xin nhà nước cấp phát cho một vùng đất để xây dựng nhà máy, đang xây giữa chừng thì bên thầu công trình ăn hết vốn nên dự án xây dựng nhà máy bị hoãn lại, nhưng đất được nhà nước cấp phát không được trả lại cho nhà nước để làm việc khác, thế là lãng phí một vùng đất mà không làm được gì, tài nguyên trên máy tính cũng tương tự như vậy.

**Để thu hồi vùng nhớ đã được cấp phát thông qua toán tử new, chúng ta sử dụng toán tử delete.**

### Dynamically allocate single variables

##### new operator

Toán tử new được dùng để xin cấp phát vùng nhớ trên phân vùng Heap của bộ nhớ ảo.

Toán tử new trong chuẩn C++11 được định nghĩa với **3 prototype** như sau:

**void**\* **operator** **new** (std::**size\_t** size);

**void**\* **operator** **new** (std::**size\_t** size, **const** std::**nothrow\_t**& nothrow\_value) **noexcept**;

**void**\* **operator** **new** (std::**size\_t** size, **void**\* ptr) **noexcept**;

Các bạn chưa cần phải hiểu những tham số khai báo cho toán tử **new**, mà hiện tại chỉ cần chú ý kiểu trả về của nó (void \*). Toán tử new sau khi xin cấp phát vùng nhớ trên **Heap** sẽ trả về một con trỏ chứa địa chỉ của vùng nhớ được cấp phát (nếu cấp phát thành công).

Kiểu trả về của toán tử **new** là con trỏ kiểu **void**, đây là một con trỏ đặc biệt, chúng ta sẽ tìm hiểu nó trong bài học sau. Nhưng dù nó là con trỏ kiểu gì thì mục đích của nó vẫn là chứa địa chỉ, do đó, chúng ta có thể gán giá trị trả về của toán tử **new** cho một con trỏ khác để quản lý vùng nhớ đã được cấp phát.

##### usage of new operator

Cú pháp sử dụng toán tử **new** như sau:

**new** <data\_**type**>;

Ví dụ:

**new** **int**; *//allocate 4 bytes on Heap partition to an int variable*

**new** **double**; *//allocate 8 bytes on Heap partition to a double variable*

Khi chương trình đang chạy, nếu quá trình cấp phát bộ nhớ trên thành công, chúng ta sẽ có địa chỉ của 2 vùng nhớ được trả về. Nhưng như mình đã nói, chúng ta không thể tạo thêm tên biến mới khi chương trình đang chạy, do đó chúng ta cần gán nó cho những con trỏ cùng kiểu để quản lý:

**int** \*p\_int = **new** **int**;

**double** \*p\_double = **new** **double**;

Bây giờ, vùng nhớ được cấp phát sẽ được quản lý bởi 2 con trỏ p\_int và p\_double, 2 vùng nhớ này được hệ điều hành trao quyền sử dụng tạm thời cho chương trình của chúng ta, thông qua con trỏ, chúng ta có thể thay đổi giá trị bên trong vùng nhớ này. Ví dụ:

**int** \*p\_int = **new** **int**;

cout << "Put value into memory area" << endl;

cin >> \*p\_int;

cout << "Value at " << p\_int << " is " << \*p\_int << endl;

Chúng ta còn có thể vừa cấp phát bộ nhớ vừa khởi tạo giá trị tại vùng nhớ đó cho một biến đơn:

**int** \*p1 = **new** **int**(5);

**int** \*p2 = **new** **int** { \*p1 };

##### usage of delete operator

Khi không muốn sử dụng tiếp vùng nhớ đã được cấp phát cho chương trình trên **Heap**, chúng ta nên trả lại vùng nhớ đó cho hệ điều hành. Thật ra khi chương trình kết thúc, tất cả vùng nhớ của chương trình đều bị hệ điều hành thu hồi, nhưng chúng ta nên giải phóng vùng nhớ không cần thiết càng sớm càng tốt.

Để xóa một vùng nhớ, chúng ta cần có một địa chỉ cụ thể, địa chỉ đó được giữ bởi con trỏ sau khi gán địa chỉ cấp phát cho nó:

**int** \*p = **new** **int**;

*//using memory area at p*

*//and then set it free*

**delete** p;

Lúc này, con trỏ p vẫn còn giữ địa chỉ của vùng nhớ đã được cấp phát trên **Heap**. Nếu may mắn, vùng nhớ đó chưa được hệ điều hành cấp phát cho chương trình khác, chúng ta vẫn có thể dùng con trỏ p để thay đổi giá trị bên trong nó.

**int** \*p = **new** **int**

**delete** p;

*//keep using that memory area*

\*p = 10;

cout << p << endl;

Nếu không may mắn, con trỏ p sẽ mang tội danh xâm nhập bất hợp pháp vào vùng nhớ của chương trình khác, và chương trình của chúng ta sẽ bị crash.

##### mean of delete operator

Sử dụng toán tử delete không có nghĩa là **delete** tất cả mọi thứ bên trong vùng nhớ mà con trỏ trỏ đến. Toán tử **new và delete** chỉ mang ý nghĩa về "quyền sử dụng" vùng nhớ. Toàn bộ dãy địa chỉ trên bộ nhớ ảo được quản lý bởi một chương trình mang tên "Hệ điều hành", và hệ điều hành có quyền trao lại quyền sử dụng một vùng nhớ nào đó (trên Stack hoặc trên Heap...) cho những chương trình đáng tin cậy trên máy tính.

Và toán tử **new** dùng để làm hợp đồng sử dụng vùng nhớ trên **Heap**, các bạn lấy vùng nhớ được cấp phát thông qua hợp đồng (**make by new operator**) để chương trình chạy, vậy khi bạn sử dụng toán tử **delete**, đơn giản là bạn chỉ xé bản hợp đồng đó đi (hoặc đưa lại cho hệ điều hành). Lúc này, Giá trị trên vùng nhớ đó **có thể** vẫn còn giữ nguyên do chưa có chương trình nào can thiệp vào.

Toán tử **delete** không tác động gì đến con trỏ.

##### Dangling pointer

"Con trỏ bị treo" thường xảy ra sau khi giải phóng vùng nhớ bằng toán tử **delete**. Sau khi sử dụng toán tử **delete**, vùng nhớ được cấp phát được trả lại cho hệ điều hành quản lý, nhưng con trỏ vẫn còn trỏ vào địa chỉ đó. Sử dụng toán tử **dereference** cho con trỏ tại thời điểm này sẽ gây ra lỗi **undefined behavior**.

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

{

**int** \*ptr = **new** **int**; *// dynamically allocate an integer*

\*ptr = 7; *// put a value in that memory location*

**delete** ptr; *// return the memory to the operating system. ptr is now a dangling pointer.*

std::cout << \*ptr; *// Dereferencing a dangling pointer will cause undefined behavior*

**delete** ptr; *// trying to deallocate the memory again will also lead to undefined behavior.*

**return** 0;

}

Còn nhiều trường hợp khác nhau có thể khiến con trỏ bị treo, mình sẽ dành ra một bài học để nói về cách quản lý vùng nhớ và con trỏ khi sử dụng kỹ thuật **Dynamic memory allocation**.

##### Điều gì xảy ra khi xin cấp phát vùng nhớ trên Heap thất bại?

Quá trình cấp phát vùng nhớ trên **Heap** thất bại có thể do có chương trình nào đó đang sử dụng lượng bộ nhớ quá lớn (ví dụ chương trình tạo máy ảo), và chương trình của bạn yêu cầu cung cấp vùng nhớ có kích thước nên hệ điều hành không thế tìm thấy đoạn vùng nhớ nào đủ cho yêu cầu của chương trình của bạn.

Chúng ta cùng xem lại các protoyte của toán tử new:

**void**\* **operator** **new** (std::**size\_t** size); *// (1)*

**void**\* **operator** **new** (std::**size\_t** size, **const** std::**nothrow\_t**& nothrow\_value) **noexcept**; *// (2)*

**void**\* **operator** **new** (std::**size\_t** size, **void**\* ptr) **noexcept**; *// (3)*

Mặc định, chúng ta sử dụng toán tử new ở cách khai báo (1), trong trường hợp này, nếu cấp phát vùng nhớ thất bại, toán tử new sẽ ném ra ngoại lệ std::bad\_alloc. Nếu ngoại lệ này không được xử lý, chương trình chúng ta sẽ bị kết thúc với lỗi **unhandled exception error**.

Trong một số trường hợp, chúng ta không muốn dính đến ngoại lệ (exception) trong C++, chúng ta nên chọn sử dụng phiên bản toán tử new (2), ví dụ:

**int** \*p = **new** (std::nothrow) **int**;

Sử dụng cách này, nếu quá trình cấp phát thất bại, toán tử **new** sẽ trả về giá trị **NULL**. Lúc này, chúng ta có thể kiểm tra xem chương trình của chúng ta có xin được vùng nhớ hay không:

**if** (p == NULL)

{

cout << "Could not allocate memory on Heap partition" << endl;

**exit**(1);

}

**else**

{

//use that memory area

//and then **delete** it

**delete** p;

}

Sử dụng cách này sẽ giúp chương trình chúng ta sử dụng con trỏ an toàn hơn khi sử dụng kỹ thuật **Dynamic memory allocation**.

### Dynamically allocate arrays

Để xin cấp phát và giải phóng vùng nhớ cho mảng một chiều trên **Heap**, chúng ta cũng sử dụng toán tử **new và delete** để xử lý.

##### Dynamically allocate arrays

Đối với việc yêu cầu cấp phát bộ nhớ cho biến đơn trên **Heap**, chúng ta chỉ cần cung cấp kiểu dữ liệu cho toán tử **new**, hệ điều hành sẽ tự tính được kích thước cần cấp phát (tương tự việc sử dụng toán tử sizeof). Nhưng khi cần cấp phát một dãy vùng nhớ liên tục nhau (mảng một chiều), ngoài kiểu dữ liệu chúng ta cần cung cấp thêm số lượng phần tử.

**new** <data\_type>[num\_**of**\_elements];

Nếu quá trình cấp phát thành công, toán tử **new** sẽ trả về địa chỉ của phần tử đầu tiên của vùng nhớ được cấp phát, và tương tự như cấp phát cho biến đơn, chúng ta cho 1 con trỏ có kiểu dữ liệu phù hợp lưu trữ địa trả về để quản lý vùng nhớ. Ví dụ:

**int** \*p\_arr = **new** **int**[10];

*//using this memory area*

**for** (**int** i = 0; i < 10; i++)

{

*//Set value for each element*

cin >> \*p\_arr[i];

}

Chúng ta có thể khởi tạo cho vùng nhớ đã được cấp phát tương tự như khởi tạo mảng một chiều thông thường. Ví dụ:

int arr[5] = { 1, 2, 3, 4, 5 };

int \*p\_arr = new int[5] { 1, 2, 3, 4, 5 }; *//no operator = between array-size and initializer list*

**Lưu ý cách này chỉ sử dụng được trong chuẩn C++11 trở lên.**

Trường hợp mảng kí tự luôn là trường hợp đặc biệt của mảng một chiều. Chúng ta không thể sử dụng cách khởi tạo này trong chuẩn C++11:

**char** \*c\_str = **new** **char** [100] { "Allocated on Heap partition" };

Nhưng trường hợp này có thể chạy được trên Visual studio 2015 với chuẩn C++14.

Điều khiến cho kỹ thuật **Dynamic memory allocation** khác với **Static memory allocation** là số lượng phần tử có thể được cung cấp trong khi chương trình đang chạy. Ví dụ:

**int** num\_of\_elements;

cout << "Enter number of elements you want to create: ";

cin >> num\_of\_elements;

**int** \*p\_arr = **new** **int**[num\_of\_elements];

Chúng ta sử dụng giá trị của biến num\_of\_elements làm số lượng phần tử cung cấp cho toán tử new, và giá trị này chỉ được xác định sau khi người dùng nhập vào từ bàn phím. Để hạn chế trường hợp người dùng nhập số âm, chúng ta cần kiểm tra trước khi xin cấp phát:

**int** num\_of\_elements;

cout << "Enter number of elements you want to create: ";

cin >> num\_of\_elements;

**if**(num\_of\_elements > 0)

**int** \*p\_arr = **new** **int**[num\_of\_elements];

##### dynamically delete arrays

Đối với dãy vùng nhớ liên tục được cấp phát trên **Heap**, chúng ta cần thêm vào toán tử [ ] để báo với hệ điều hành rằng vùng nhớ đã được cấp phát không dùng cho một biến đơn.

**int** \*p\_arr = **new** **int**[10];

*//...........*

**delete**[] p\_arr;

Sử dụng toán tử **delete** theo cách giải phóng vùng nhớ biến đơn cho dãy vùng nhớ liên tục có thể gây ra nhiều vấn đề khác nhau cho chương trình (**memory leak, data corruption**, ...).

##### resizing dynamic arrays

Trong nhiều trường hợp, chúng ta cần thay đổi kích thước vùng nhớ đã được cấp phát cho phù hợp với yêu cầu của chương trình. Cách duy nhất là:

* Cấp phát lại vùng nhớ mới.
* (Copy dữ liệu từ vùng nhớ cũ sáng vùng nhớ mới nếu cần).
* Giải phóng vùng nhớ cũ.
* Cho con trỏ trỏ đến vùng nhớ mới.

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

{

**int** \*p = **new** **int**[5];

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

{

cin >> \*(p + i);

}

*//re-allocate*

**int** \*p\_temp = p;

p = **new** **int**[10];

*//copy data*

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

{

\*(p + i) = \*(p\_temp + i);

}

*//dealocate old memory area*

**delete**[] p\_temp;

*//keep using data*

*//and then delete it*

**delete**[] p;

system("pause");

**return** 0;

}

Do vùng nhớ mới sẽ có địa chỉ khác với vùng nhớ đã cấp phát ban đầu, mình cần sử dụng con trỏ p\_temp để giữ lại khả năng truy cập đến vùng nhớ ban đầu. Sau khi copy toàn bộ dữ liệu từ vùng nhớ cũ sang vùng nhớ mới, chúng ta nên giải phóng vùng nhớ cũ ngay để khỏi lãng phí tài nguyên hệ thông.

### Tổng kết

Trong bài học này, chúng ta đã tìm hiểu về kỹ thuật **Dynamic memory allocation** trong ngôn ngữ C++. Kỹ thuật này giúp chương trình chúng ta ít bị giới hạn dung lượng bộ nhớ hơn. Nhưng bên cạnh đó, chúng ta cần có kỹ năng về quản lý các vùng nhớ trong chương trình. Sử dụng kỹ thuật **Dynamic memory allocation** không thành thạo là nguyên nhân gây phổ biến gây ra lỗi **memory leak**. Do đó, chúng ta sẽ có một bài học nói về các lỗi thường gặp khi sử dụng **Dynamic memory allocation** và cách kiểm soát các lỗi này.

# 8.5 Con trỏ và hằng

#### Chào các bạn đang theo dõi khóa học lập trình trực tuyến ngôn ngữ C++.

Trong các bài học có liên quan đến con trỏ trước đây, chúng ta đã biết con trỏ cũng là một biến thông thường mà giá trị nó có thể chứa là địa chỉ của vùng nhớ khác. Như vậy, từ khóa const cũng có thể được sử dụng cho con trỏ như các biến có kiểu dữ liệu khác. Tuy nhiên, tùy vào vị trí đặt từ khóa const khi khai báo con trỏ mà nó lại có những ý nghĩa khác nhau.

##### Pointer to const

Thử xem xét ví dụ sau:

**int** **value** = 5;

**int** \*ptr = &**value**;

\*ptr = 10; *//change value to 10*

Với đoạn code này, chương trình của chúng ta hoạt động bình thường. Nó đơn thuần chỉ là dùng một con trỏ có tên ptr trỏ đến địa chỉ của biến value. Bây giờ chúng ta có một chút thay đổi như sau:

**const** **int** **value** = 5;

**int** \*ptr = &**value**; *//compile error*

Trong đoạn code trên, mình đã đặt vùng nhớ tại địa chỉ của biến value là vùng nhớ hằng, điều đó có nghĩa giá trị bên trong vùng nhớ đó không thể bị thay đổi. Mặc dù chúng ta chỉ mới cho con trỏ ptr trỏ đến vùng nhớ hằng đó chứ chưa thực hiện câu lệnh nào liên quan đến việc thay đổi giá trị bên trong vùng nhớ của biến value, nhưng compiler ngăn chặn điều này để đảm bảo an toàn dữ liệu cho vùng nhớ của biến value.

Như vậy, công cụ con trỏ thông thường không được phép sử dụng để trỏ đến vùng nhớ hằng, chúng ta cần sử dụng công cụ khác, có thể gọi là **Pointer to const** (Con trỏ dùng để trỏ đến hằng). Để có một Pointer to const, chúng ta chỉ cần thêm từ khóa const đứng trước kiểu dữ liệu của con trỏ.

**const** **int** **value** = 5;

**const** **int** \*ptr = &**value**; *//it's ok, ptr point to a "const int"*

\*ptr = 10; *//compile error*

Lúc này, con trỏ ptr trở thành Pointer to const nên nó đã có thể trỏ đến vùng nhớ hằng. Tuy nhiên, con trỏ này cũng không thể thay đổi giá trị bên trong vùng nhớ hằng. Do đó, compiler thông báo lỗi "cannot assign to a variable that is const".

Một Pointer to const dùng để trỏ đến một vùng nhớ hằng, nó cũng có thể trỏ đến một vùng nhớ không phải hằng. Ví dụ:

**int** **value** = 5;

**const** **int** \*ptr = &**value**;

\*ptr = 10; *//compile error*

Mặc dù Pointer to const có thể trỏ đến vùng nhớ không phải hằng, nhưng nó lại không thể thay đổi giá trị bên trong vùng nhớ đó. Nếu biên dịch đoạn code trên, compiler sẽ thông báo lỗi "assignment of read-only location '\* ptr'". Điều này có nghĩa Pointer to const là loại con trỏ chỉ có chức năng đọc nội dung của vùng nhớ (bất kể vùng nhớ đó có phải hằng hay không) chứ không có chức năng ghi giá trị vào vùng nhớ.

**Do đó, sử dụng Pointer to const sẽ đảm bảo toàn vẹn dữ liệu cho vùng nhớ mà nó trỏ đến.**

Điểm đáng chú ý ở Pointer to const là một Pointer to const không phải là một biến hằng, nó chỉ là một loại công cụ có chức năng read-only. Do đó, chúng ta vẫn có thể cho Pointer to const trỏ đến vùng nhớ khác sau khi khởi tạo.

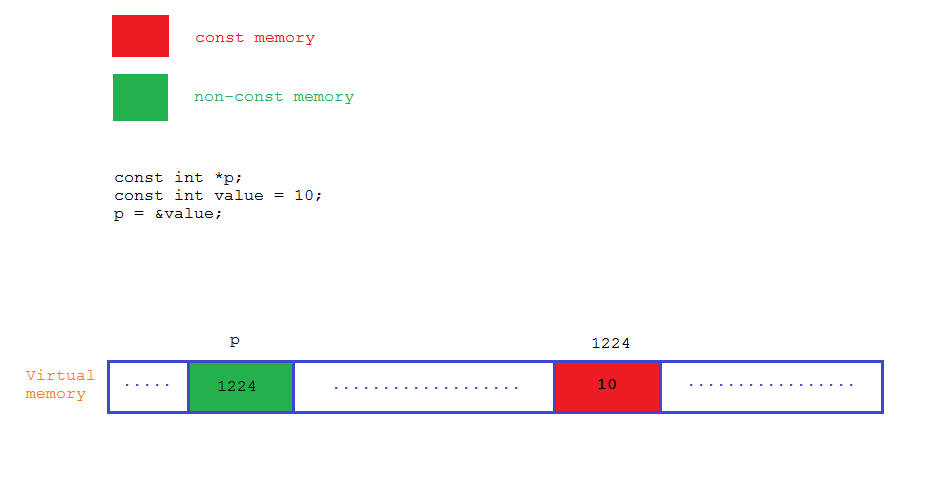
**const** int \*ptr = NULL*;*

int value1 = 5*;*

ptr = &value1*;*

int value2 = 10*;*

ptr = &value2*;*

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-5-con-tro-va-hang/0.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-5-con-tro-va-hang/0.png?raw=true" \o "0.png?raw=true)**

**[0.png?raw=true939x494](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-5-con-tro-va-hang/0.png?raw=true" \o "0.png?raw=true)**

Chúng ta có thể khai báo Pointer to const bằng cách đặt từ khóa const như sau:

int **const** \*ptr = NULL*;*

Nhưng đây là cách khai báo dễ nhầm lẫn nên mình vẫn thích dùng cách cũ hơn:

**const** int \*ptr = NULL*;*

##### Const pointer

Const pointer là loại con trỏ chỉ gán được địa chỉ một lần khi khởi tạo, điều này có nghĩa sau khi trỏ đến vùng nhớ nào đó thì nó không thể trỏ đi nơi khác được. Để khai báo const pointer, chúng ta cần đặt từ khóa con giữa dấu \* và tên con trỏ.

**int** **value** = 5;

**int** \***const** ptr = &**value**;

Cũng giống như const variable, const pointer cần được khởi tạo ngay sau khi khai báo, và địa chỉ được gán cho const pointer sẽ không thể thay đổi về sau.

int value1 = 5*;*

int value2 = 10*;*

int \***const** ptr = &value1*;*

ptr = &value2*; //compile error*

Xét lại đoạn chương trình này:

**int** **value** = 5;

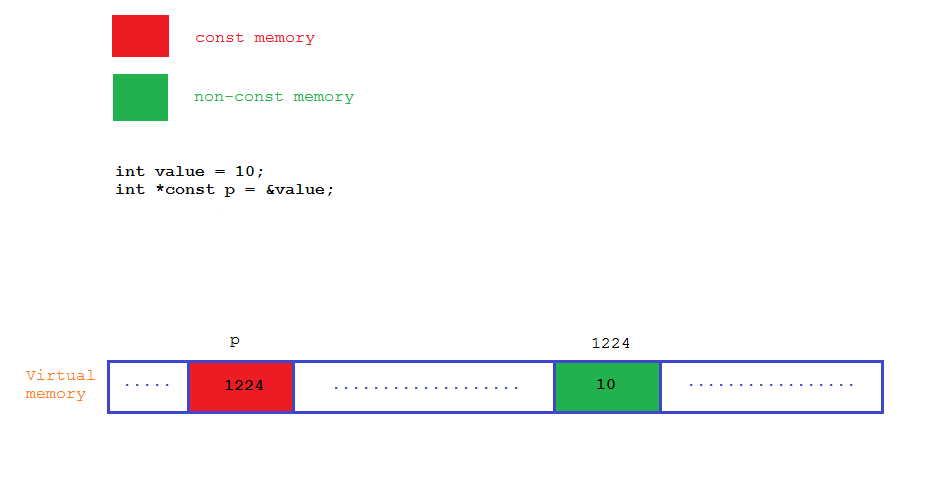
**int** \***const** ptr = &**value**;

Lúc này, con trỏ ptr chính nó là hằng, nhưng vùng nhớ mà nó trỏ đến (biến value) lại không phải hằng, nên con trỏ ptr hoàn toàn có thể thay đổi giá trị của biến value.

int value = 5;

int \***const** **ptr** = &value;

\***ptr** = 10; //it's ok, change value **of** non-**const** memory area

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-5-con-tro-va-hang/1.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-5-con-tro-va-hang/1.png?raw=true" \o "1.png?raw=true)**

**[1.png?raw=true939x494](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-5-con-tro-va-hang/1.png?raw=true" \o "1.png?raw=true)**

Như vậy, const pointer có đầy đủ chức năng đọc và ghi giá trị lên vùng nhớ.

##### Const pointer to const

Chúng ta hoàn toàn có thể kết hợp cả 2 loại con trỏ trên lại với nhau để tạo thành loại con trỏ mới gọi là **const pointer to const**. Loại con trỏ này sẽ có chức năng read-only, và nó cũng không thể trỏ đến vùng nhớ khác sau khi đã khởi tạo.

**int** **value** = 5;

**const** **int** \***const** ptr = &**value**;

&ptr = 10; *//compile error*

**int** otherValue = 10;

ptr = &otherValue; *//compile error*

### Tổng kết

Trong bài học này, chúng ta đã được biết thêm một số khái niệm về con trỏ có liên quan đến hằng. Khi sử dụng từ khóa const, chúng ta có thể tạo ra 2 loại con trỏ tùy vào vị trí đặt từ khóa const. Pointer to const là loại con trỏ chỉ có chức năng đọc nội dung vùng nhớ mà nó trỏ đến. Const pointer cũng tương tự như một biến hằng thông thường, vì giá trị mà nó nắm giữ là địa chỉ và nó là hằng, nên địa chỉ của con trỏ này sẽ không bị thay đổi (không thể trỏ đến vùng nhớ khác sau khi khởi tạo).

### Bài tập cơ bản

Trong số các đoạn code dưới đây, đoạn code nào có thể biên dịch được?

Code 1:

**#include <iostream>**

**#include <cstring>**

**using** **namespace** std;

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

**char** str[] = "Le Tran Dat";

**char** \***const** p\_str = str;

**for**(**int** i = 0; i < strlen(str); p\_str++)

{

\*p\_str = ' ';

}

cout << p\_str << endl;

**return** 0;

}

Code 2:

**#include <iostream>**

**#include <cstring>**

**using** **namespace** std;

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

**char** str[] = "Le Tran Dat";

**const** **char** \*p\_str = str;

**for**(**int** i = 0; i < strlen(str); i++)

{

\*(p\_str + i) = ' ';

}

cout << p\_str << endl;

**return** 0;

# 8.6 Con trỏ void

#### Chào các bạn đang theo dõi khóa học lập trình trực tuyến ngôn ngữ C++.

### Void pointers

Con trỏ kiểu **void**, có thể gọi là con trỏ tổng quát, là một kiểu dữ liệu đặc biệt của con trỏ. Con trỏ kiểu **void** có thể trỏ đến bất kỳ đối tượng nào (với bất kỳ kiểu dữ liệu nào) có địa chỉ cụ thể trên bộ nhớ ảo. Cách khai báo con trỏ kiểu **void** cũng giống với các con trỏ có kiểu dữ liệu được xây dựng sẵn:

void \***ptr**; // **ptr** **is** a void pointer

Bây giờ, mình có thể gán địa chỉ của các biến có kiểu dữ liệu khác nhau cho con trỏ ptr:

void \*ptr*;*

int iValue*;*

float fValue*;*

double dValue*;*

string str*;*

int iArr[10]*;*

ptr = &iValue*;*

ptr = &fValue*;*

ptr = &dValue*;*

ptr = &str*;*

ptr = iArr*;*

Chúng ta còn có thể cho con trỏ **void** trỏ đến những con trỏ khác:

void \*ptr*;*

int \*iArr = new int[10]*;*

ptr = iArr*;*

delete[] iArr*;*

Cũng như mọi kiểu con trỏ khác, con trỏ kiểu void cũng có kích thước 4 bytes khi chạy trên nền tảng 32 bits, hoặc 8 bytes nếu chạy trên nền tàng 64 bits. Tuy nhiên, con trỏ kiểu **void** không xác định được kiểu dữ liệu của vùng nhớ mà nó trỏ tới, chúng ta không thể truy xuất trực tiếp nội dung thông qua toán tử **dereference** được. Do đó, con trỏ kiểu void cần phải được ép kiểu một cách rõ ràng sang con trỏ có kiểu dữ liệu khác trước khi sử dụng toán tử dereference cho vùng nhớ mà con trỏ đang nắm giữ.

Ví dụ:

**int** value = 5;

**void** \*vPtr = &value;

**int** \*iPtr = **static\_cast**<**int** \*> (vPtr);

cout << \*iPtr << endl;

Lúc này, vPtr và iPtr đều trỏ vào địa chỉ của biến value, nhưng chúng ta chỉ có thể sử dụng toán tử **dereference** lên con trỏ iPtr chứ không thể sử dụng cho con trỏ vPtr.

Điều gì xảy ra nếu chúng ta ép sai kiểu dữ liệu?

Thử với ví dụ sau:

**int** value = 5;

**void** \*vPtr = &value;

**int** \*iPtr = **static\_cast**<**int** \*> (vPtr);

cout << \*iPtr << endl;

**int64\_t** \* i64Ptr = **static\_cast**<**int64\_t** \*> (vPtr);

cout << \*i64Ptr << endl;

Kết quả:

5  
-5233161171908952059

Như các bạn thấy, sử dụng sai kiểu dữ liệu đi kèm với đó là kết quả không mong muốn. Do đó, chúng ta nên hạn chế sử dụng con trỏ kiểu void.

**Vậy chúng ta sử dụng con trỏ kiểu void cho mục đích gì?**

Chúng ta thường sử dụng con trỏ kiểu **void** khi mà dữ liệu bên trong vùng nhớ đó không quan trọng. Ví dụ mình muốn copy dữ liệu từ dãy vùng nhớ này sang dãy vùng nhớ khác mà không cần quan tâm định dạng của chúng.

Ví dụ:

**void** \*ptr = **operator** **new** (100);

Dòng lệnh trên sử dụng toán tử new để cấp phát 100 bytes trên Heap partition và lưu địa chỉ của vùng nhớ đó bên trong con trỏ ptr.

Con trỏ void cũng thường được sử dụng làm tham số của hàm khi muốn input của hàm là con trỏ có kiểu dữ liệu bất kỳ. Chúng ta sẽ tìm hiểu vấn đề này trong những bài học tiếp theo.

Việc giải phóng một vùng nhớ trên Heap bằng tên con trỏ kiểu void cũng có thể gây ra lỗi vì hệ điều hành không tính được kích thước vùng nhớ cần thu hồi là bao nhiêu.

### Tổng kết

Trên thực tế, chúng ta nên tránh sử dụng con trỏ kiểu void trừ những lúc thực sự cần thiết để tránh gây ra những sai sót không đáng có cho chương trình.

# 8.7 Con trỏ trỏ đến con trỏ

#### Chào các bạn đang theo dõi khóa học lập trình trực tuyến ngôn ngữ C++.

Trong bài học này, chúng ta sẽ cùng nhau tìm hiểu một khái niệm nâng cao của con trỏ: "Con trỏ trỏ đến con trỏ".

### Pointer to pointer

**Pointer to pointer** là một loại con trỏ dùng để lưu trữ địa chỉ của biến con trỏ.

Mình lấy ví dụ về việc sử dụng con trỏ thông thường:

**int** **value**;

**int** \*ptr = &**value**;

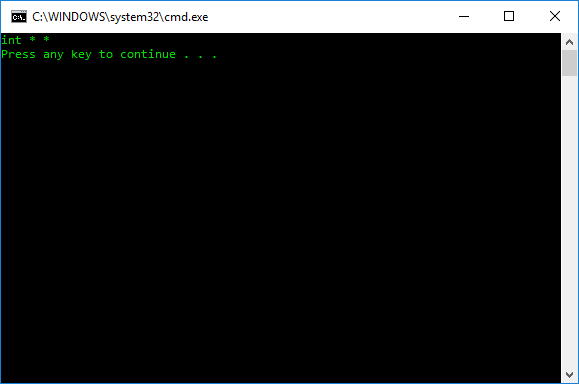
Chúng ta gán được địa chỉ của biến value cho con trỏ **ptr** vì biến **value** là biến kiểu **int**, và sử dụng toán tử **address-of** cho biến value sẽ trả về giá trị kiểu (int \*) giống với kiểu dữ liệu của con trỏ ptr.

Như vậy, nếu chúng ta muốn **Pointer to pointer** trỏ đến được một **pointer** khác, trước hết chúng ta cần xem kiểu dữ liệu khi sử dụng toán tử **address-of** cho con trỏ sẽ trả về giá trị kiểu gì.

**int** \*ptr = NULL;

cout << **typeid**(&ptr).name() << endl;

Kết quả:



Như chúng ta thấy, chúng ta cần khai báo biến có kiểu dữ liệu (int \*\*) để có thể gán địa chỉ của con trỏ kiểu (int \*) cho nó. Let's try:

int \*ptr = NULL*;*

int \*\*p\_to\_p = &ptr*;*

Con trỏ p\_to\_p được gọi là một **Pointer to pointer.**

Cũng tương tự như khi sử dụng con trỏ thông thường, chúng ta có thể sử dụng toán tử **dereference** cho một **Pointer to pointer.**

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

**int** value = 100;

**int** \*ptr = &value;

**int** \*\*p\_to\_p = &ptr;

cout << p\_to\_p << endl; *//print address of ptr*

cout << \*p\_to\_p << endl; *//print address which hold by ptr*

cout << \*\*p\_to\_p << endl; *//print value at address which hold by ptr*

**return** 0;

}

Bản chất của **Pointer to pointer** vẫn là một **pointer**, nên khi truy xuất giá trị của p\_to\_p chúng ta lấy được địa chỉ mà nó trỏ đến (địa chỉ của biến ptr).

p\_to\_p; *//là &ptr*

Khi chúng ta sử dụng 1 toán tử **dereference** cho 1 **pointer to pointer**, cũng đồng nghĩa chúng ta đang truy xuất đến giá trị tại địa chỉ mà con trỏ ptr nắm giữ (địa chỉ đang được lưu trữ trong biến ptr).

\*p\_to\_p; *//là ptr*

Và khi sử dụng 2 toán tử **dereference** cho 1 **pointer to pointer,** có thể viết lại như sau:

\**(\*p\_to\_p)*; *//là \*ptr*

Chúng ta có thể thấy việc sử dụng **pointer to pointer** cũng tương tự như việc đi hỏi tìm một người bạn mà không biết nhà nó ở đâu, chỉ biết nhà của những người biết về nó. Vậy là chúng ta đi hỏi từng người một.

Ví dụ chúng ta là A, đang cần gặp C nhưng không biết nó ở đâu, chúng ta hỏi (sử dụng toán tử dereference) chú B thì chú B bảo đến địa chỉ mà C đang ở, chúng ta đến địa chỉ mà chú B nắm giữ và truy xuất vào đó là sẽ tìm được thằng C.

Tóm tắt lại ví dụ trên, chúng ta có thể viết:

A giữ địa chỉ nhà chú B => A = &B;  
Chú B biết địa chỉ nhà thằng C => B = &C;

Như vậy:

(\*A) tương đương (\*(&B)) tương đương &C;

\*(\*A) tương đương \*(\*(&B)) tương đương C;

Áp dụng lại cho ví dụ:

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

**int** value = 100;

**int** \*ptr = &value;

**int** \*\*p\_to\_p = &ptr;

cout << p\_to\_p << endl; *//print address of ptr*

cout << \*p\_to\_p << endl; *//print address which hold by ptr*

cout << \*\*p\_to\_p << endl; *//print value at address which hold by ptr*

**return** 0;

}

Chúng ta có thể viết:

p\_to\_p giữ địa chỉ của ptr => p\_to\_p = &ptr*;*

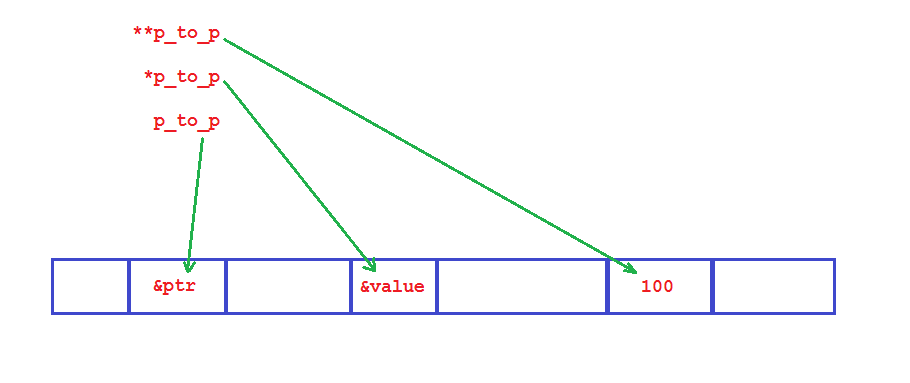
ptr giữ địa chỉ của value => ptr = &value*;*

Như vậy:

(\*p\_to\_p) tương đương ptr tương đương &value;

\*\*p\_to\_p tương đương \*ptr tương đương value;

Kết quả:

**[[](https://raw.githubusercontent.com/nguyenchiemminhvu/CPP-Tutorial/master/8-con-tro/8-7-con-tro-tro-den-con-tro/1.png)](https://raw.githubusercontent.com/nguyenchiemminhvu/CPP-Tutorial/master/8-con-tro/8-7-con-tro-tro-den-con-tro/1.png" \o "1.png)**

**[1.png910x383](https://raw.githubusercontent.com/nguyenchiemminhvu/CPP-Tutorial/master/8-con-tro/8-7-con-tro-tro-den-con-tro/1.png" \o "1.png)**

Chúng ta không thể gán trực tiếp như sau:

**int** \*\*p\_to\_p = &&**value**; *//not valid*

Vì p\_to\_p là lvalue, &&value là rvalue. [**https://msdn.microsoft.com/en-us/library/f90831hc.aspx**](https://msdn.microsoft.com/en-us/library/f90831hc.aspx)

Và cũng tương tự như những con trỏ khác, Pointer to pointer có thể gán giá trị NULL:

**int** \*\*p\_to\_p = NULL;

### Array of pointers

**Pointer to pointer** có thể được dùng để quản lý mảng một chiều kiểu con trỏ (int \*[]). Ví dụ:

int main() {

int \*p1 = NULL;

int \*p2 = NULL;

int \*p3 = NULL;

int \*p[] = { p1, p2, p3 };

int \*\*p\_to\_p = p;

return 0;

}

Trong trường hợp này, p\_to\_p[0] tương đương với p[0].

Thông thường, chúng ta sẽ sử dụng **pointer to pointer** để quản lý vùng nhớ được cấp phát trên **Heap** cho mảng một chiều chứa các con trỏ.

**int** \*p1 = NULL;

**int** \*p2 = NULL;

**int** \*p3 = NULL;

**int** \*\*p\_to\_p = **new** **int**\*[3];

p\_to\_p[0] = p1;

p\_to\_p[1] = p2;

p\_to\_p[2] = p3;

**delete**[] p\_to\_p;

Tương tự như con trỏ kiểu (int \*) dùng để trỏ đến mảng các phần tử kiểu int, con trỏ kiểu (int \*\*) dùng để trỏ đến mảng các phần tử kiểu (int \*).

### 2D dynamically allocated array

Một cách sử dụng khác của **Pointer to pointer** là dùng để quản lý mảng hai chiều được cấp phát trên Heap.

Với mảng hai chiều cấp phát trên **Stack**, chúng ta chỉ cần khai báo như sau:

int arr2D[5][10];

Nhưng với mảng hai chiều cấp phát trên **Heap** sẽ rắc rối hơn.

Chúng ta biết rằng, mảng hai chiều là một tập hợp của các mảng một chiều có cùng kích thước. Chúng ta cũng đã biết cách cấp phát vùng nhớ cho mảng một chiều trên **Heap** bằng cách dùng toán tử **new** đi kèm với toán tử [ ]. Ví dụ:

**int** \*arr1 = **new** **int**[10];

**int** \*arr2 = **new** **int**[10];

*//........*

Như vậy, một mảng các con trỏ được dùng để quản lý tập hợp các mảng một chiều này sẽ tạo thành mảng 2 chiều. Ví dụ:

**int** \*pToArrPtr[3];

**for**(**int** i = 0; i < 3; i++)

{

pToArrPtr[i] = **new** **int**[5];

}

Kết quả của đoạn chương trình này cho chúng ta một vùng nhớ có kích thước (3 x 5) phần tử kiểu int. Và chúng ta có thể truy xuất từng giá trị thông qua con trỏ **pToArrPtr**:

**for**(**int** i = 0; i < 3; i++)

{

**for**(**int** j = 0; j < 5; j++)

{

cin >> pToArrPtr[i][j];

}

}

cout << "--------------------------------" << endl;

**for**(**int** i = 0; i < 3; i++)

{

**for**(**int** j = 0; j < 5; j++)

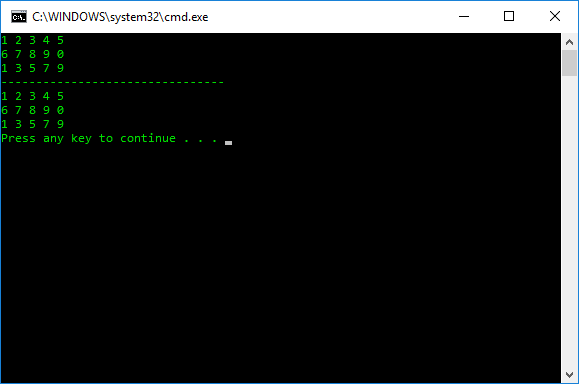
{

cout << pToArrPtr[i][j] << " ";

}

cout << endl;

}



Kết quả hoàn toàn giống với mảng hai chiều thông thường. Nhưng lúc này, 3 con trỏ pToArrPtr[0] và pToArrPtr[1] và pToArrPtr[2] vẫn là biến được cấp phát trên **Stack**. Để chuyển những con trỏ quản lý các mảng một chiều con này sang **Heap**, chúng ta cần sử dụng **Pointer to pointer**. Dưới đây là toàn bộ chương trình mẫu cho việc cấp phát và giải phóng vùng nhớ 2 chiều hoạt động tương tự như mảng hai chiều thông thường:

**#include <iostream>**

**using** **namespace** std;

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

**int** \*\*pToArrPtr;

*//Cấp phát vùng nhớ cho 3 con trỏ kiểu (int \*)*

pToArrPtr = **new** **int**\*[3];

*//Mỗi con trỏ kiểu (int \*) sẽ quản lý 5 phần tử kiểu int*

**for** (**int** i = 0; i < 3; i++)

{

pToArrPtr[i] = **new** **int**[5];

}

**for** (**int** i = 0; i < 3; i++)

{

**for** (**int** j = 0; j < 5; j++)

{

cin >> pToArrPtr[i][j];

}

}

cout << "--------------------------------" << endl;

**for** (**int** i = 0; i < 3; i++)

{

**for** (**int** j = 0; j < 5; j++)

{

cout << pToArrPtr[i][j] << " ";

}

cout << endl;

}

*//Giải phóng vùng nhớ cho từng dãy vùng nhớ mà 3 con trỏ đang quản lý*

**for** (**int** i = 0; i < 3; i++)

{

**delete**[] pToArrPtr[i];

}

*//Giải phóng cho 3 biến con trỏ chịu sự quản lý của pToArrPtr*

**delete**[] pToArrPtr;

**return** 0;

}

### Pointer to pointer to pointer to ...

Chúng ta có thể khai báo những con trỏ có dạng như sau:

int \*\*\*ptrX3*;*

int \*\*\*\*\*\*ptrX6*;*

Tuy nhiên, việc thao tác với những con trỏ như thế này khá phức tạp và rất ít gặp trong thực tế nên mình không đề cập trong bài học này.

### Tổng kết

**Pointer to pointer** là một phần nâng cao của con trỏ. Việc thao tác cấp phát và giải phóng vùng nhớ khá phức tạp. Do đó, các bạn mới học có thể hoàn toàn bỏ qua bài học này. Mình cũng khuyên các bạn nên tránh sử dụng **Pointer to pointer** trừ khi không còn giải pháp nào thay thế.

# 8.8 Con trỏ và hàm

#### Chào các bạn đang theo dõi khóa học lập trình trực tuyến ngôn ngữ C++.

### Con trỏ và tham số của hàm

Chúng ta đã tìm hiểu về 2 kiểu tham số của hàm:

* Hàm có tham số nhận giá trị: giá trị truyền vào hàm có thể là giá trị của biến, một hằng số hoặc một biểu thức toán học...
* Hàm có tham số kiểu tham chiếu: giá trị truyền vào cho hàm là tên biến, và tham số của hàm sẽ tham chiếu trực tiếp đến vùng nhớ của biến đó.

Chúng ta còn có thêm một kiểu truyền dữ liệu vào cho hàm nữa, đó là truyền địa chỉ vào hàm (**Pass arguments by address**). Do đó, kiểu tham số của hàm có thể nhận giá trị là địa chỉ phải là con trỏ.

**void** **foo**(**int** \*iPtr)

{

cout << "Int value at " << iPtr << " is " << \*iPtr << endl;

}

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

{

**int** iValue = 10;

foo(&iValue);

system("pause");

**return** 0;

}

Trong đoạn chương trình trên, sau khi truyền địa chỉ của biến iValue vào hàm foo, tham số iPtr bây giờ sẽ giữ địa chỉ của biến iValue, và chúng ta có thể sử dụng toán tử dereference cho con trỏ iPtr. Kết quả in ra màn hình trên máy tính của mình là:

**Int** value **at** 0xBFBA144C is 10

Nếu vùng nhớ tại địa chỉ được sử dụng làm đối số cho hàm không phải là hằng, chúng ta có thể thay đổi giá trị của vùng nhớ đó ngay bên trong hàm thông qua toán tử **dereference**:

**void** **changeValue**(**int** \*iPtr)

{

\*iPtr = 10;

}

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

{

**int** iValue = 5;

cout << "Value = " << iValue << endl;

changeValue(&iValue);

cout << "Value = " << iValue << endl;

system("pause");

**return** 0;

}

Kết quả in ra:

Value = 5

Value = 10

Như vậy, chúng ta có thể hoán vị giá trị của 2 số nguyên thông qua hàm như sau:

**void** **swapIntValue**(**int** \*ptr1, **int** \*ptr2)

{

**int** temp = \*ptr1;

\*ptr1 = \*ptr2;

\*ptr2 = temp;

}

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

{

**int** value1 = 2;

**int** value2 = 5;

cout << "Before swap: " << value1 << " " << value2 << endl;

swapIntValue(&value1, &value2);

cout << "After swap : " << value1 << " " << value2 << endl;

system("pause");

**return** 0;

}

Kết quả:

**Before swap:** 2 5

After **swap** : 5 2

Như các bạn thấy, con trỏ khi làm tham số cho hàm cũng có khả năng thay đổi giá trị của vùng nhớ không phải hằng như con trỏ thông thường thông qua toán tử **dereference**.

Chúng ta còn có thể truyền địa chỉ của mảng một chiều vào cho tham số kiểu con trỏ của hàm. Ví dụ:

**void** **printArray**(**int** \*arr, **int** length)

{

**for** (**int** i = 0; i < length; i++)

{

cout << arr[i] << " ";

}

cout << endl;

}

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

{

**int** iArr[] = { 3, 2, 5, 1, 7, 10, 32 };

printArray(iArr, **sizeof**(iArr) / **sizeof**(**int**));

system("pause");

**return** 0;

}

Lưu ý, chúng ta không thể biết chính xác kích thước của mảng một chiều thông qua con trỏ, do đó, chúng ta cần tính toán trước kích thước của mảng trước khi truyền vào cho hàm.

##### Sử dụng Pointer to const để làm tham số cho hàm

Như các bạn đã biết, **Pointer to const** là loại con trỏ chỉ có chức năng để đọc (read-only). Do đó, sử dụng **Pointer to const** làm tham số cho hàm sẽ đảm bảo rằng giá trị tại vùng nhớ được truyền vào cho hàm sẽ không bị thay đổi.

**void** **printArray**(**const** **int** \*arr, **int** length)

{

**for** (**int** i = 0; i < length; i++)

{

cout << arr[i] << " ";

}

cout << endl;

}

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

{

**int** arr[] = {};

**int** length = **sizeof**(arr) / **sizeof**(**int**);

printArray(arr, length);

system("pause");

**return** 0;

}

Lúc này, chúng ta có thể đảm bảo rằng giá trị của các phần tử trong mảng arr sẽ không bị thay đổi bởi hàm printArray.

##### 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");

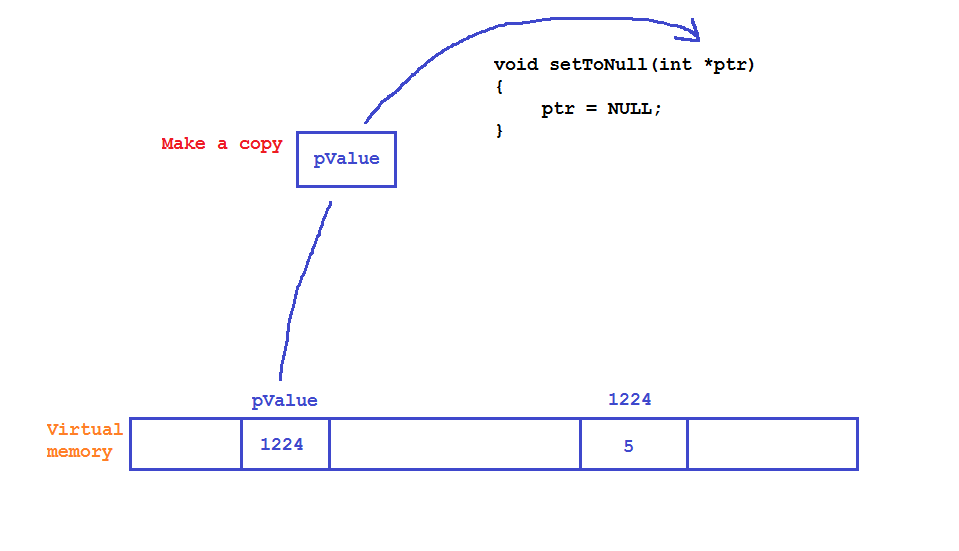
**return** 0;

}

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)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-8-con-tro-va-ham/0.png?raw=true" \o "0.png?raw=true)**

**[0.png?raw=true959x551](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-8-con-tro-va-ham/0.png?raw=true" \o "0.png?raw=true)**

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:

**void** **setToNull**(**int** \*&ptr)

{

ptr = NULL;

}

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

{

**int** value = 5;

**int** \*pValue = &value;

cout << "pValue point to " << pValue << endl;

setToNull(pValue);

**if**(pValue == NULL)

cout << "pValue point to NULL" << endl;

**else**

cout << "pValue point to " << pValue << endl;

**return** 0;

}

Kết quả của đoạn chương trình này cho thấy con trỏ pValue sau khi truyền vào hàm setToNull đã được gán giá trị NULL. Do tham số con trỏ của hàm setToNull là một tham chiếu kiểu (int \*), nó sẽ tham chiếu đến đối số được truyền vào, trong trường hợp này, tham số tham chiếu con trỏ ptr có cùng địa chỉ với pValue, việc thay đổi giá trị mà ptr nắm giữ cũng làm thay đổi giá trị của pValue.

### Con trỏ và kiểu trả về của hàm

Chúng ta đã cùng tìm hiểu 2 kiểu giá trị trả về của hàm có kiểu trả về:

* Hàm trả về giá trị.
* Hàm trả về tham chiếu.

Bây giờ, chúng ta sẽ cùng tìm hiểu một số vấn đề về kiểu giá trị trả về của hàm là địa chỉ (**return by address**).

Khi nói về việc trả về địa chỉ từ hàm, chúng ta hiểu rằng đó là địa chỉ của những biến hoạt động bên trong hàm. Địa chỉ này sẽ được trả về cho lời gọi hàm, và địa chỉ này thường được tiếp tục sử dụng bằng cách gán nó lại cho 1 con trỏ. Do đó, kiểu trả về của hàm cũng phải là kiểu con trỏ.

Ví dụ:

**int** \* **createAnInteger**(**int** value = 0)

{

**int** myInt = value;

**return** &myInt;

}

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

{

**int** \*pInt = createAnInteger(10);

cout << \*pInt << endl;

**return** 0;

}

Sau khi nhìn vào kết quả, chúng ta thấy có vẻ chương trình đã cho ra kết quả như mong muốn:

10

Nhưng thực chất, đoạn chương trình trên đã gây ra lỗi nghiêm trọng. Lý do là biến myInt được khai báo bên trong hàm là biến cục bộ, được cấp phát bằng kỹ thuật **Automatic memory allocation**, và vùng nhớ được cấp phát cho biến myInt được lưu trữ trên phân vùng **Stack** của bộ nhớ ảo. Do đó, ngay sau khi ra khỏi hàm, vùng nhớ của biến myInt đã bị hệ điều hành thu hồi, nhưng địa chỉ của biến myInt trước đó đã được trả về cho lời gọi hàm, nên con trỏ pInt trong hàm main được gán một địa chỉ của một vùng nhớ không thuộc quyền quản lý của chương trình hiện hành nữa.

Như mình đã nói, nếu không may, một chương trình khác yêu cầu cấp phát vùng nhớ ngay tại địa chỉ của biến myInt lúc chưa bị hủy, nội dung bên trong vùng nhớ này sẽ bị các chương trình khác thay đổi, dẫn đến việc sử dụng toán tử **dereference** đến vùng nhớ đó không cho ra kết quả như ban đầu nữa. Các bạn có thể chạy đoạn chương trình sau để kiểm chứng:

**int** \* **createAnInteger**(**int** value = 0)

{

**int** myInt = value;

**return** &myInt;

}

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

{

**int** \*pInt = createAnInteger(10);

cout << "Print immediately: " << \*pInt << endl;

\_sleep(1000);

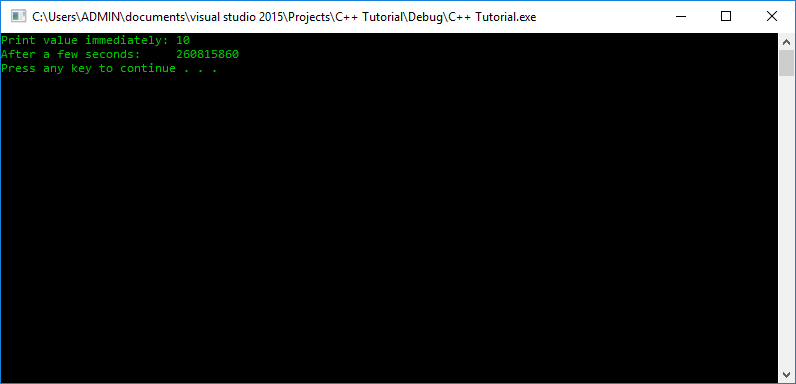
cout << "After a fews seconds: " << \*pInt << endl;

system("pause");

**return** 0;

}

Kết quả trên máy tính của mình:

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-8-con-tro-va-ham/1.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-8-con-tro-va-ham/1.png?raw=true" \o "1.png?raw=true)**

**[1.png?raw=true796x384](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-8-con-tro-va-ham/1.png?raw=true" \o "1.png?raw=true)**

Như các bạn thấy, chỉ sau thời điểm vùng nhớ của biến myInt bị hủy mới có 1 giây mà đã có chương trình khác sử dụng vùng nhớ đó, làm cho giá trị in ra màn hình console không còn như ban đầu nữa. Và nếu không may hơn nữa, nếu chương trình khác sử dụng cơ chế đồng bộ của kỹ thuật multithreading lên vùng nhớ này, việc dereference vào vùng nhớ đó cũng có thể gây crash chương trình.

Nguyên nhân của những hệ quả mà mình vừa kể ra đều là do vùng nhớ được cấp phát trên Stack thông qua kỹ thuật **Automatic memory allocation** sẽ bị thu hồi tự động bởi hệ điều hành. Để giải quyết vấn đề này, chúng ta cần sử dụng phân vùng Heap để có thể tự quản lý thời điểm giải phóng vùng nhớ để trả lại cho hệ điều hành quản lý.

**int** \* **createAnInteger**(**int** value = 0)

{

**return** **new** **int**(value);

}

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

{

**int** \*pInt = createAnInteger(10);

cout << "Print immediately: " << \*pInt << endl;

\_sleep(5000);

cout << "After a few seconds: " << \*pInt << endl;

system("pause");

**return** 0;

}

Kết quả lúc này đã được đảm bảo do chúng ta biết rằng vùng nhớ cấp phát trên Heap chỉ bị hệ điều hành thu hồi khi toàn bộ chương trình kết thúc.

### Tổng kết

Trong bài học này, chúng ta đã biết cách truyền tham số là địa chỉ (hoặc con trỏ) vào cho hàm, và trả về địa chỉ cho lời gọi hàm. Bên cạnh đó, chúng ta cũng đã biết được một số vấn đề phát sinh khi sử dụng các kỹ thuật này. Vẫn còn nhiều vấn đề cần phải nói khi sử dụng con trỏ, chúng ta sẽ cùng tiếp tục tìm hiểu trong các bài học tiếp theo.

### Bài tập cơ bản

Xét đoạn chương trình của ví dụ trên.

**#include <iostream>**

**using** **namespace** std;

**int** \* **createAnInteger**(**int** value = 0)

{

**return** **new** **int**(value);

}

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

{

**int** \*pInt = createAnInteger(10);

cout << "Print immediately: " << \*pInt << endl;

\_sleep(5000);

cout << "After a few seconds: " << \*pInt << endl;

system("pause");

**return** 0;

}

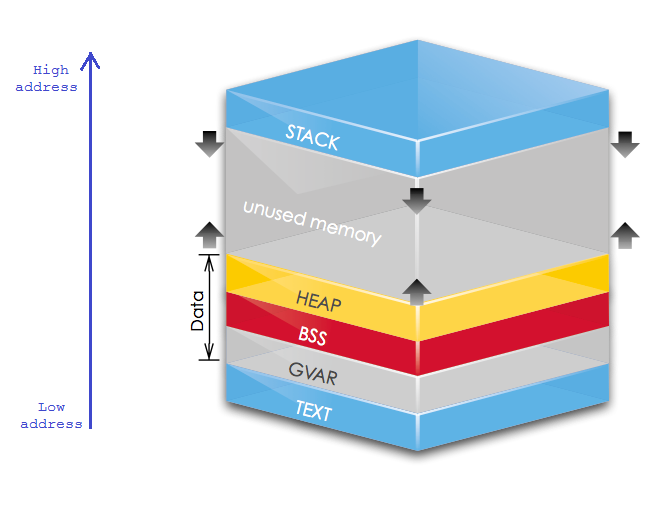
Đoạn chương trình trên cho ra kết quả đúng, giá trị được in ra khi sử dụng toán tử dereference để truy xuất không bị thay đổi theo thời gian, nhưng nó lại phát sinh một vấn đề khác. Đó là vấn đề gì?

# 8.9 Con trỏ hàm

#### Chào các bạn đang theo dõi khóa học lập trình trực tuyến ngôn ngữ C++.

Tiếp tục tìm hiểu về con trỏ trong ngôn ngữ lập trình C++, trong bài học này, mình sẽ giới thiệu đến các bạn một loại con trỏ mới có chức năng khá đặc biệt.

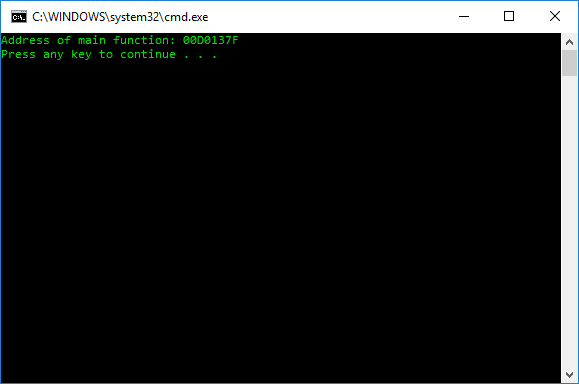
Như chúng ta đã biết, con trỏ có chức năng lưu trữ địa chỉ của một vùng nhớ nào đó trên bộ nhớ ảo. Tuy nhiên, bộ nhớ ảo được chia làm nhiều phân vùng khác nhau.

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-9-con-tro-ham/0.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-9-con-tro-ham/0.png?raw=true" \o "0.png?raw=true)**

**[0.png?raw=true670x523](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-9-con-tro-ham/0.png?raw=true" \o "0.png?raw=true)**

Như trong hình, hầu hết toàn bộ phân vùng của bộ nhớ ảo đều dùng để lưu trữ dữ liệu (biến đơn, giá trị, chuỗi kí tự, ...). Nhưng dữ liệu lưu trên bộ nhớ ảo là cái được tạo ra sau khi chương trình nào đó được thực thi và nó xin cấp phát vùng nhớ trên bộ nhớ ảo để sử dụng. Trước đó, chương trình hoạt động dựa trên các dòng lệnh mà lập trình viên đưa ra sử dụng cú pháp của ngôn ngữ lập trình nào đó. Và trước khi chạy chương trình, mã nguồn (đã được biên dịch thành mã máy) đang lưu trữ trong máy tính (có thể là trên ổ cứng) cũng phải được hệ điều hành load lên RAM và quản lý bằng cách đưa vào bộ nhớ ảo. Vậy mã nguồn của chương trình sẽ lưu ở đâu trên bộ nhớ ảo? Đó là tại phân vùng Text (**Text segment**) hay còn gọi là phân vùng Code (**Code segment**). Tất cả các lệnh, các hàm... của chương trình sẽ được đưa vào phân vùng này, trong đó có cả hàm **main** nếu đó là chương trình C++. Như các bạn đã biết, một chương trình C++ sẽ có duy nhất một hàm **main** đóng vai trò là điểm bắt đầu của chương trình đó. Như vậy, sau khi được load mã nguồn C++ đã được biên dịch lên bộ nhớ ảo, hệ điều hành sẽ tìm đến vị trí (địa chỉ) của hàm **main** và chuyển mã nguồn đến cho CPU xử lý.

Mình có thể cho các bạn xem địa chỉ của hàm **main** của một chương trình C++ ở trên máy mình như hình bên dưới:



Như các bạn thấy, hàm **main** (hay bất kỳ hàm nào khác trong chương trình) có một địa chỉ xác định trên bộ nhớ ảo. Do đó, chúng ta có thể sử dụng con trỏ để trỏ đến địa chỉ của hàm **main**. Tuy nhiên, chúng ta cần lưu ý đến kiểu dữ liệu khai báo cho con trỏ phải tương thích với kiểu dữ liệu của vùng nhớ. Ví dụ, con trỏ kiểu **int** dùng để trỏ đến vùng nhớ kiểu **int**, con trỏ trỏ đến hằng (**Pointer to const**) dùng để trỏ đến vùng nhớ hằng... Và để trỏ đến địa chỉ của một hàm, chúng ta cần sử dụng con trỏ hàm (**Function pointer** hoặc có thể gọi là **Pointer to function**).

### Function pointers

Khi nhìn vào một hàm (function), ví dụ:

**int** **foo**()

{

**return** 0;

}

Chúng ta có thể nói hàm này có định danh là **foo**, kiểu trả về là int, hàm foo không nhận đối số. Đó là những gì chúng ta thấy được trong quá trình biên soạn mã nguồn chương trình. Thuộc tính địa chỉ của hàm chỉ được sinh ra khi chương trình đã được chạy.

**int** **foo**() *// code of foo start at memory address 0x01001492*

{

**return** 0;

}

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

{

**int** n = foo();

**return** 0;

}

Như vậy, khi trong hàm **main** chạy đến dòng lệnh gọi hàm **foo**, hệ điều hành sẽ tìm đến địa chỉ của hàm **foo** trên bộ nhớ ảo và chuyển mã lệnh của hàm **foo** cho CPU tiếp tục xử lý. Để in ra địa chỉ của hàm **foo**, chúng ta có thể làm như sau:

**int** **foo**()

{

**return** 0;

}

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

{

cout << foo << endl;

**return** 0;

}

Kết quả:

013D1492

Như các bạn thấy, khi muốn thực thi một hàm, chúng ta cần thêm cặp dấu ngoặc để truyền đối số vào cho hàm (nếu hàm không có tham số thì để trống). Nếu chúng ta không sử dụng cặp dấu ngoặc, sử dụng tên hàm trả về địa chỉ của hàm trên bộ nhớ ảo. Và địa chỉ này có thể được gán con một con trỏ có kiểu dữ liệu tương ứng (**function pointer**).

##### Function pointers syntax

Cú pháp của một con trỏ hàm có nhiều điểm khác biệt so với cách khai báo con trỏ thông thường.

<return\_**type**> (\*<name\_of\_pointer>)( <data\_type\_of\_parameters> );

Mình lấy ví dụ, để trỏ đến hàm foo trong ví dụ trên, chúng ta cần khai báo con trỏ hàm như sau:

int (\*pFoo) ()*;*

Trong đó, **int** là kiểu trả về của hàm **foo**, **pFoo** là tên của con trỏ, và hàm **foo** không có tham số nên phần trong ngoặc mình bỏ trống. Một ví dụ khác, mình có hàm như bên dưới:

**void** swapValue(int &**value**1, int &**value**2)

{

int temp = **value**1;

**value**1 = **value**2;

**value**2 = temp;

}

Hàm **swapValue** có không có kiểu trả về, và nó nhận vào 2 tham số đều có kiểu tham chiếu **int**. Như vậy, mình có thể khai báo một con trỏ hàm dùng để trỏ đến hàm **swapValue** như sau:

void(\*pSwap) (int &, int &)*;*

##### Gán địa chỉ của hàm cho Function pointers

Sau khi đã có được con trỏ hàm được khai báo tương ứng với hàm, chúng ta có thể gán địa chỉ của hàm cho chúng:

**void** **swapValue**(**int** &value1, **int** &value2)

{

**int** temp = value1;

value1 = value2;

value2 = temp;

}

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

{

**void**(\*pSwap) (**int** &, **int** &) = swapValue;

cout << pSwap << endl;

cout << swapValue << endl;

**return** 0;

}

**Lưu ý, khi cần lấy địa chỉ của hàm, chúng ta chỉ sử dụng duy nhất tên hàm, không đặt thêm cặp dấu ngoặc vào.**

Chỉ có con trỏ được khai báo có kiểu dữ liệu trả về và danh sách tham số phù hợp mới trỏ đến hàm được.

*// function prototypes*

**int** **foo**();

**double** **goo**();

**int** **hoo**(**int** x);

*// function pointer assignments*

**int** (\*funcPtr1)() = foo; *// okay*

**int** (\*funcPtr2)() = goo; *// wrong -- return types don't match!*

**double** (\*funcPtr4)() = goo; *// okay*

funcPtr1 = hoo; *// wrong -- fcnPtr1 has no parameters, but hoo() does*

**int** (\*funcPtr3)(**int**) = hoo; *// okay*

##### Sử dụng Function pointers

Sau khi đã nắm giữ được địa chỉ của hàm, con trỏ hàm có thể được sử dụng như hàm thông qua toán tử **dereference**. Ví dụ:

**void** **swapValue**(**int** &value1, **int** &value2)

{

**int** temp = value1;

value1 = value2;

value2 = temp;

}

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

{

**void**(\*pSwap) (**int** &, **int** &) = swapValue;

**int** a = 1, b = 5;

cout << "Before: " << a << " " << b << endl;

(\*pSwap)(a, b);

cout << "After: " << a << " " << b << endl;

**return** 0;

}

Lưu ý, tham số mặc định của hàm không áp dụng được cho con trỏ hàm, vì tham số mặc định được **compiler** xác định tại thời điểm **compile** chương trình, còn con trỏ hàm được sử dụng tại thời điểm chương trình đang chạy.

##### Sử dụng con trỏ hàm làm tham số

Một con trỏ hàm cũng là một biến con trỏ, do đó chúng ta có thể sử dụng con trỏ hàm là tham số của một hàm nào đó. Khi tham số của hàm là con trỏ hàm, chúng ta sẽ truyền đối số là địa chỉ của một hàm. Hàm được sử dụng làm đối số của hàm có thể gọi là **callback function**.

Mình lấy ví dụ về hàm **selectionSort** dùng để sắp xếp dữ liệu trong mảng số nguyên theo thứ tự tăng dần:

**#include <algorithm> *// use for std::swap***

**void** **selectionSort**(**int** \*arr, **int** length)

{

**for** (**int** i\_start = 0; i\_start < length; i\_start++)

{

**int** minIndex = i\_start;

**for** (**int** i\_current = i\_start + 1; i\_current < length; i\_current++)

{

**if** (arr[minIndex] > arr[i\_current])

{

minIndex = i\_current;

}

}

swap(arr[i\_start], arr[minIndex]); *// std::swap*

}

}

*//................*

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

{

**int** arr[] = { 1, 4, 2, 3, 6, 5, 8, 9, 7 };

**int** length = **sizeof**(arr) / **sizeof**(**int**);

cout << "Before sorted: ";

printArray(arr, length);

selectionSort(arr, length);

cout << "After sorted: ";

printArray(arr, length);

**return** 0;

}

Kết quả:

Before sorted: 1 4 2 3 6 5 8 9 7

After sorted: 1 2 3 4 5 6 7 8 9

Bây giờ đặt ra trường hợp chúng ta muốn có sắp xếp mảng một chiều sử dụng thuật toán **selectionSort** nhưng sắp xếp theo thứ tự giảm dần. Như vậy, chúng ta cần đến 2 hàm **selectionSort** để đáp ứng cho 2 trường hợp mình kể trên. Trong khi cả 2 hàm **selectionSort** này chỉ khác nhau về toán tử so sánh tại phép so sánh ở mệnh đề if trong vòng lặp.

Để giải quyết vấn đề này, đầu tiên mình cần có 2 hàm thực hiện công việc so sánh như sau:

bool ascending(int left, int right)

{

return left > right;

}

bool descending(int left, int right)

{

return left < right;

}

Hai hàm này chỉ có tác dụng thay thế phép so sánh trong mệnh đề if. Nếu chúng ta cần sắp xếp giá trị trong mảng theo thứ tự tăng dần, chúng ta sẽ thay thế hàm **ascending** vào mệnh đề if trong hàm **selectionSort** như sau:

**void** selectionSort(**int** \*arr, **int** length)

{

**for** (**int** i\_start = 0; i\_start < length; i\_start++)

{

**int** minIndex = i\_start;

**for** (**int** i\_current = i\_start + 1; i\_current < length; i\_current++)

{

**if** (ascending(arr[minIndex], arr[i\_current])) *// replace comparison expression by ascending function*

{

minIndex = i\_current;

}

}

swap(arr[i\_start], arr[minIndex]); *// std::swap*

}

}

Lúc này, chúng ta vẫn còn cần thêm một hàm **selectionSort** khác và thay thế biểu thức so sánh trong mệnh đề if bằng hàm **descending** để có thể sắp xếp mảng theo chiều giảm dần. Chúng ta cần thiết kế lại hàm **selectionSort** này sao cho người dùng có thể tùy chọn việc sắp xếp theo thứ tự tăng dần hay giảm dần theo từng thời điểm khác nhau. Chúng ta sẽ thêm vào tham số thứ 3 cho hàm **selectionSort** là một con trỏ hàm dùng để trỏ đến hàm **ascending** hoặc **descending** tùy vào lời gọi hàm **selectionSort**. Do hàm **ascending** và **descending** có cấu trúc kiểu trả về và tham số hoàn toàn giống nhau, nên chúng ta có thể sử dụng chung một kiểu con trỏ hàm. Mình định nghĩa con trỏ hàm dùng làm tham số thứ 3 của hàm **selectionSort** như sau:

bool (\*comparisonFunc)(int, int)*;*

Bây giờ, chúng ta sẽ sửa lại hàm **selectionSort** thành phiên bản sử dụng con trỏ hàm:

**bool** **ascending**(**int** left, **int** right)

{

**return** left > right;

}

**bool** **descending**(**int** left, **int** right)

{

**return** left < right;

}

**void** **selectionSort**(**int** \*arr, **int** length, **bool** (\*comparisonFunc)(**int**, **int**))

{

**for** (**int** i\_start = 0; i\_start < length; i\_start++)

{

**int** minIndex = i\_start;

**for** (**int** i\_current = i\_start + 1; i\_current < length; i\_current++)

{

**if** (comparisonFunc(arr[minIndex], arr[i\_current])) *// use function pointer as ascending or descending function*

{

minIndex = i\_current;

}

}

swap(arr[i\_start], arr[minIndex]); *// std::swap*

}

}

Lúc này, giả sử mình muốn sắp xếp mảng theo thứ tự giảm dần, mình sẽ sử dụng lời gọi hàm như sau:

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

{

**int** arr[] = { 1, 4, 2, 3, 6, 5, 8, 9, 7 };

**int** length = **sizeof**(arr) / **sizeof**(**int**);

cout << "Before sorted: ";

printArray(arr, length);

selectionSort(arr, length, descending);

cout << "After sorted: ";

printArray(arr, length);

**return** 0;

}

Mình sử dụng địa chỉ của hàm **descending** làm đối số cho tham số thứ 3 của hàm **selectionSort**. Như vậy, hàm **descending** sẽ được sử dụng bên trong hàm **selectionSort** và mảng của chúng ta sẽ sắp xếp theo thứ tự giảm dần.

Before sorted: 1 4 2 3 6 5 8 9 7

After sorted: 9 8 7 6 5 4 3 2 1

Nếu muốn đổi ngược lại thứ tự của mảng khi sắp xếp, chúng ta chỉ cần thay đối số thứ 3 của hàm **selectionSort** là địa chỉ của hàm **ascending**:

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

{

**int** arr[] = { 1, 4, 2, 3, 6, 5, 8, 9, 7 };

**int** length = **sizeof**(arr) / **sizeof**(**int**);

cout << "Before sorted: ";

printArray(arr, length);

selectionSort(arr, length, ascending);

cout << "After sorted: ";

printArray(arr, length);

**return** 0;

}

Với việc đặt thêm tham số thứ 3 của hàm **selectionSort** là 1 con trỏ hàm, chúng ta có thể thiết kế thêm nhiều tùy chọn cho điều kiện sắp xếp mảng một chiều khác nhau và vẫn có thể sử dụng cho hàm **selectionSort**. Ví dụ mình muốn thêm một kiểu sắp xếp có điều kiện khác là mọi số chẵn trong mảng sẽ đứng trước, các số lẻ trong mảng sẽ đứng sau, và phần chẵn hay phần lẻ đều được sắp xếp tăng dần, mình sẽ làm như sau:

bool evensFirst(int left, int right)

{

//**if** left is even and right is odd, not need to swap

**if** ((left % 2 == 0) && (right % 2 != 0))

**return** false;

//**if** left is odd and right is even, swap this couple

**if** ((left % 2 != 0) && (right % 2 == 0))

**return** true;

**return** ascending(left, right);

}

Và mình chỉ cần sử dụng địa chỉ của hàm này làm đối số:

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

{

**int** arr[] = { 1, 4, 2, 3, 6, 5, 8, 9, 7 };

**int** length = **sizeof**(arr) / **sizeof**(**int**);

cout << "Before sorted: ";

printArray(arr, length);

selectionSort(arr, length, evensFirst);

cout << "After sorted: ";

printArray(arr, length);

**return** 0;

}

Kết quả:

Before sorted: 1 4 2 3 6 5 8 9 7

After sorted: 2 4 6 8 1 3 5 7 9

Như các bạn thấy, sử dụng con trỏ hàm trong trường hợp này mang lại cho chúng ta một cách sử dụng hàm sắp xếp hiệu quả hơn.

##### Tham số mặc định của tham số con trỏ hàm

Chúng ta có thể cung cấp cho tham số con trỏ hàm một địa chỉ hàm cụ thể, và hàm đó sẽ được gọi mặc định nếu chúng ta không cung cấp đối số cho tham số con trỏ hàm. Ví dụ:

**void** selectionSort(**int** \*arr, **int** length, **bool** (\*comparisonFunc)(**int**, **int**) = ascending)

{

**for** (**int** i\_start = 0; i\_start < length; i\_start++)

{

**int** minIndex = i\_start;

**for** (**int** i\_current = i\_start + 1; i\_current < length; i\_current++)

{

**if** (comparisonFunc(arr[minIndex], arr[i\_current])) *// use function pointer as ascending or descending function*

{

minIndex = i\_current;

}

}

swap(arr[i\_start], arr[minIndex]); *// std::swap*

}

}

Như vậy, lời gọi hàm **selectionSort** với 2 đối số sẽ được mặc định là sắp xếp mảng tăng dần.

### std::function in C++11

Chuẩn C++11 cung cấp cho chúng ta một cách thay thế cho việc sử dụng con trỏ hàm bằng cách sử dụng kiểu dữ liệu **function** thuộc thư viện **functional**. Thư viện này cũng được định nghĩa trong namespace **std** nên chúng ta cần có dòng lệnh using namespace std; hoặc có thể khai báo là std::function.

Cú pháp khai báo biến kiểu std::function như sau:

std::**function**< <**return\_type**>([list **of** parameters]) > **varName**;

Ví dụ:

std::function< **bool**(**int**, **int**) > comparisonFunc;

Mình lấy ví dụ về việc sử dụng kiểu dữ liệu std::function như sau:

**#include <iostream>**

**#include <functional>**

**using** **namespace** std;

**void** **addOne**(**int** &value)

{

value++;

}

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

{

function<**void**(**int** &)> func = addOne;

**int** number = 5;

func(number);

cout << "New value: " << number << endl;

**return** 0;

}

Như các bạn thấy, sử dụng kiểu dữ liệu std::function cũng tương tự như sử dụng con trỏ hàm, chỉ khác nhau về cách khai báo.

### Tổng kết

Con trỏ hàm (**function pointers**) thường được sử dụng khi chúng ta có các hàm có cùng kiểu trả về và danh sách tham số. Đặt con trỏ hàm làm tham số của hàm cũng là một cách sử dụng con trỏ hàm khá phổ biến.

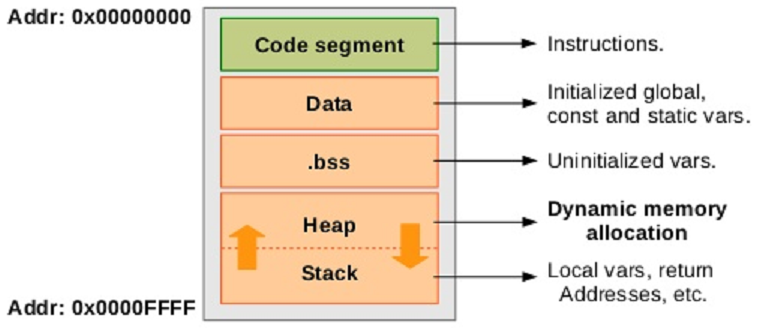
Vì con trỏ hàm có cú pháp khai báo khó nhớ hơn kiểu std::function, mình khuyến khích các bạn sử dụng kiểu std::function được định nghĩa bên trong thư viện **functional** của chuẩn C++11.

# 8.10 Phân loại các vùng nhớ (stack & heap ...)

#### Chào các bạn đang theo dõi khóa học lập trình trực tuyến ngôn ngữ C++.

Sau khi tìm hiểu một số khái niệm cơ bản về con trỏ, cấp phát động, ... chúng ta đã thấy được dãy địa chỉ bộ nhớ ảo được chia thành nhiều phân vùng khác nhau và được sử dụng cho những mục đích khác nhau. Trong bài học này, mình sẽ cùng các bạn tổng hợp lại chức năng của một số phân vùng trên bộ nhớ ảo.

Dưới đây là hình ảnh minh họa cho thứ tự các phân vùng trên bộ nhớ ảo:

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-10-cac-phan-vung-tren-bo-nho-ao/0.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-10-cac-phan-vung-tren-bo-nho-ao/0.png?raw=true" \o "0.png?raw=true)**

**[0.png?raw=true760x328](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-10-cac-phan-vung-tren-bo-nho-ao/0.png?raw=true" \o "0.png?raw=true)**

##### Code segment

Code segment (text segment) là nơi mà lưu trữ các mã lệnh đã được biên dịch của các chương trình máy tính. Những mã lệnh trong phân vùng này sẽ được chuyển đến CPU xử lý khi cần thiết. Code segment chỉ chịu sự chi phối của hệ điều hành, các tác nhân khác không thể can thiệp trực tiếp đến phân vùng này. Việc đưa các mã lệnh đã được biên dịch của chương trình lên phân vùng code segment là công việc đầu tiên mà hệ điều hành cần làm khi chúng ta chạy chương trình.

##### Data segment

Data segment (initialized data segment) là phân vùng mà hệ điều hành sử dụng để khởi tạo giá trị cho các biến kiểu static, biến toàn cục (global variable) của các chương trình.

##### BSS segment

BSS segment (uninitialized data segment) cũng được dùng để lưu trữ các biến kiểu static, biến toàn cục (global variable) nhưng chưa được khởi tạo giá trị cụ thể.

##### Heap segment

Heap segment (free srote segment) được sử dụng để cấp phát bộ nhớ thông qua kỹ thuật Dynamic memory allocation.

Để sử dụng kỹ thuật cấp phát bộ nhớ động, ngôn ngữ C++ đã hổ trợ sẵn cho chúng ta toán tử new. Ví dụ:

**new** **int**; *//allocate 4 bytes on Heap segment*

**new** **int**[10]; *//allocate (4 \* 10) bytes on Heap segment*

Toán tử new sau khi thực thi thành công sẽ trả về địa chỉ của vùng nhớ được cấp phát trên heap, chúng ta có thể sử dụng con trỏ có kiểu dữ liệu phù hợp để lưu trữ địa chỉ trả về này, và con trỏ cũng là công cụ duy nhất giúp chúng ta có thể xác định được vị trí vùng nhớ được cấp phát là ở đâu, và cũng thông qua con trỏ để chúng ta có thể giải phóng vùng nhớ đã được cấp phát.

**int** \*pInt = **new** **int**;

**int** \*pArr = **new** **int**[10];

Chúng ta không cần biết rõ cơ chế quản lý bộ nhớ Heap như thế nào, mà chỉ cần biết rằng bộ nhớ được cấp phát trên Heap sẽ không tự giải phóng cho đến khi nào toàn bộ chương trình đang chạy kết thúc. Do đó, nếu chương trình có thời gian chạy quá lâu mà không được giải phóng các vùng nhớ một cách hợp lý, điều này sẽ làm ảnh hưởng đến việc cấp phát bộ nhớ động cho các chương trình khác.

Mình có thể kể ra một số ưu điểm và nhược điểm đáng chú ý khi sử dụng phân vùng Heap như sau:

* Việc cấp phát bộ nhớ trên Heap chậm hơn các phân vùng khác.
* Vùng nhớ đã được cấp phát sẽ vẫn thuộc quyền kiểm soát của chương trình đang chạy cho đến khi chúng được giải phóng, hoặc nhận được tín hiệu kết thúc chương trình.
* Vùng nhớ được cấp phát phải được quản lý bởi ít nhất 1 con trỏ.
* Toán tử dereference truy xuất đến vùng nhớ chậm hơn các biến thông thường.
* Phân vùng Heap có dung lượng lớn nhất, nên chúng ta có thể sử dụng một cách thoải mái hơn các phân vùng khác.

##### Stack segment

Call Stack (thường được gọi là Stack) được dùng để cấp phát bộ nhớ cho tham số của các hàm (function parameters) và biến cục bộ (local variables). Call Stack được thực hiện theo cấu trúc dữ liệu stack, do đó, trước khi nói về phân vùng Stack trên bộ nhớ ảo mình sẽ trình bày cho các bạn về cấu trúc dữ liệu stack trước.

**Stack data structure**

Stack là một cơ chế tổ chức dữ liệu. Các bạn cũng từng làm việc với một kiểu tổ chức dữ liệu khá phổ biến là mảng một chiều. Mỗi cấu trúc dữ liệu sẽ tổ chức dữ liệu dưới một cơ chế khác nhau để sử dụng hiệu quả trong từng công việc cụ thể. Bây giờ chúng ta xem xét cấu trúc dữ liệu stack.

Dưới đây là một hình ảnh minh họa cho một stack trong đời sống hằng ngày:

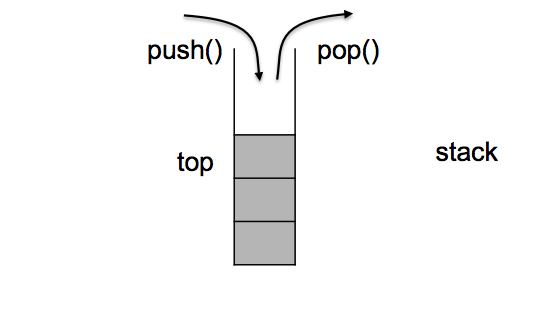


Những đĩa CD này được đặt chồng lên nhau. Khi nhìn vào chồng đĩa CD này, chúng ta chỉ có thể thực hiện 3 công việc:

(1) Nhìn vào đĩa CD trên cùng của chồng đĩa.  
(2) Lấy ra một đĩa CD nằm trên cùng.  
(3) Đặt thêm một đĩa CD lên trên cùng của chồng đĩa.

Do đó, chúng ta có thể nhận thấy ngay việc tổ chức dữ liệu theo cơ chế stack gặp nhiều hạn chế hơn so với tổ chức dữ liệu theo mảng một chiều.

Khi sử dụng mảng một chiều, chúng ta có thể truy cập vào bất kì phần tử nào bên trong mảng bằng cách đưa ra chỉ số của phần tử. Nhưng đối với stack thì không được. Chúng ta chỉ có thể thao tác với phần tử nằm trên cùng (ngoài cùng). Chúng ta thường nói stack hoạt động theo cơ chế "Last-in, first-out". Có nghĩa là phần tử nào được thêm vào mảng sau cùng thì sẽ được lấy ra đầu tiên.



Ví dụ:

Stack ban đầu của chúng ta là

--------------------------------

| 4 7 2 5

--------------------------------

Thêm vào phần tử có giá trị là 3

--------------------------------

| 4 7 2 5 3

--------------------------------

Thêm vào phần tử có giá trị 9

--------------------------------

| 4 7 2 5 3 9

--------------------------------

Lấy một phần tử ra khỏi stack

--------------------------------

| 4 7 2 5 3

--------------------------------

**Call Stack segment**

Call stack segment cũng hoạt động dựa trên cơ chế tổ chức dữ liệu như stack. Khi bắt gặp một dòng lệnh khai báo biến, nếu biến đó là biến cục bộ hoặc tham số hàm, nó sẽ được cấp phát tại địa chỉ lớn nhất hiện tại trên Stack. Khi một biến cục bộ hoặc tham số của hàm ra khỏi phạm vi khối lệnh, nó sẽ được đưa ra khỏi Stack.

Để kiểm chứng điều này, các bạn có thể chạy thử đoạn chương trình sau:

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

{

**int** n1, n2, n3, n4, n5;

cout << "Address of " << &n1 << endl;

cout << "Address of " << &n2 << endl;

cout << "Address of " << &n3 << endl;

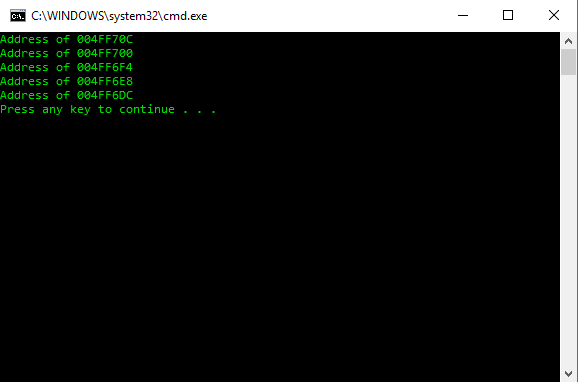
cout << "Address of " << &n4 << endl;

cout << "Address of " << &n5 << endl;

**return** 0;

}

Đoạn chương trình này khai báo lần lượt 5 biến cục bộ liên tiếp nhau. Nếu trong trường hợp tại thời điểm khai báo, chỉ có chương trình này được CPU xử lý, chúng ta sẽ thấy địa chỉ của 5 biến cục bộ này có địa chỉ liên tiếp nhau.



Địa chỉ sau cách địa chỉ trước đó đúng bằng kích thước của kiểu dữ liệu **int**.

Như vậy, lần lượt biến n1 n2 n3 n4 và n5 được cấp phát tại những địa chỉ tiếp theo (từ thấp đến cao) trên phân vùng Stack, và khi ra khỏi hàm main, lần lượt biến n5 n4 n3 n2 và n1 sẽ bị đưa ra khỏi Stack.

**Stack overflow**

Phân vùng Stack có kích thước khá hạn chế. Trên hệ điều hành Windows mà mình đang sử dụng, Call Stack chỉ có kích thước khoảng 1MB. Nếu chúng ta cố gắng cho chương trình cấp phát vùng nhớ trên Stack vượt quá kích thước của Stack, chúng ta gọi đó là hiện tượng tràn bộ nhớ phân vùng Stack (Stack overflow).

**Một số ưu và nhược điểm có thể nhận thấy khi sử dụng phân vùng Stack**

* Việc cấp phát bộ nhớ trên Call Stack khá nhanh.
* Nhìn vào mã nguồn chương trình, chúng ta có thể biết được thời điểm cấp phát và hủy vùng nhớ của biến trên Stack.
* Kích thước vùng nhớ cấp phát trên phân vùng Stack phải được khai báo rõ ràng trước khi biên dịch.
* Vùng nhớ trên phân vùng Stack có thể được truy cập trực tiếp thông qua định danh.
* Kích thước của phân vùng Stack khá hạn chế.

### Tổng kết

Trong bài học này, chúng ta đã cùng tìm hiểu qua một số phân vùng bộ nhớ trên dãy địa chỉ bộ nhớ ảo. Còn một phân vùng nữa thuộc vùng dịa chỉ nhỏ nhất, đứng trước Code segment là phân vùng dành cho hệ điều hành. Vì hệ điều hành cũng là một chương trình (nhưng thuộc về hệ thống) nên nó cũng cần được load lên bộ nhớ ảo như những chương trình thông thường. Điều đặc biệt là phân vùng này ngăn chặn mọi hành vi truy cập từ phía người dùng, do đó mình không đề cập đến trong bài học này.

# 8.11 Các vấn đề thường gặp khi sử dụng con trỏ

#### Chào các bạn đang theo dõi khóa học lập trình trực tuyến ngôn ngữ C++.

Trong bài học này, chúng ta sẽ cùng tìm hiểu về những lỗi thường gặp ở những bạn mới học lập trình khi sử dụng con trỏ trong chương trình. Những lỗi này thường xuất hiện do các bạn mới học chưa hiểu rõ cách quản lý vùng nhớ trong chương trình, do cách tổ chức chương trình chưa hợp lý, hoặc do sơ ý... Trong đó, một số lỗi không nghiêm trọng sẽ không gây ảnh hưởng nhiều đến hoạt động của chương trình, một số lỗi nghiêm trọng có thể làm chương trình phải kết thúc ngay lập tức. Dù lỗi gây có có nghiêm trọng hay không, chúng ta chủ động ngăn chặn thì vẫn tốt hơn.

##### Con trỏ trỏ đến vùng nhớ nằm ngoài phạm vi chương trình đang quản lý

Mình có đoạn chương trình như sau:

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

{

*//allocate memory on Heap*

**int** \*p = **new** **int**;

*//p point to somewhere*

p -= 10000;

*//dereference to the area of other program*

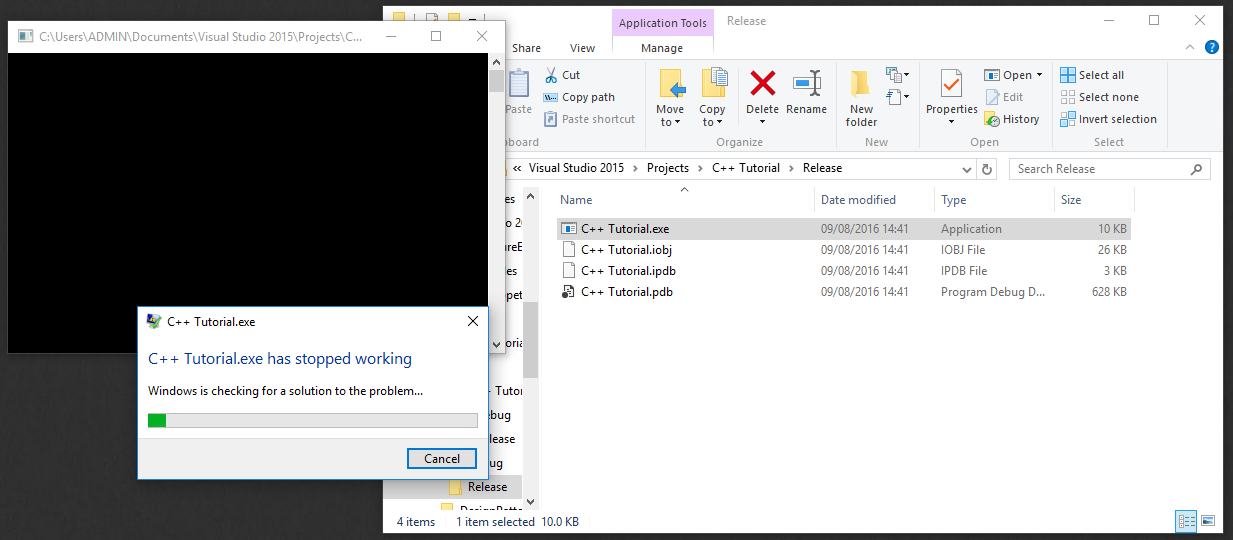
\*p = 1;

system("pause");

**return** 0;

}

Đầu tiên, chương trình cấp phát một vùng nhớ có kích thước **4 bytes** trên phân vùng **Heap** và cho con trỏ nắm giữ địa chỉ trả về của toán tử **new**. Sau đó, mình lại cho con trỏ p trỏ lung tung trên bộ nhớ ảo và cố tình thay đổi nội dung bên trong vùng nhớ mới trỏ đến. Khi để ở chế độ **Debug**, Visual Studio sẽ chặn chương trình của chúng ta lại và đưa ra cảnh báo. Nhưng nếu chúng ta chuyển sang chế độ **Release** và **build** ra sản phẩm thành một phần mềm, chạy phần mềm này sẽ gây **crash** và phần mềm phải kết thúc ngay lập tức.

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-11-cac-van-de-thuong-gap-khi-su-dung-con-tro/0.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-11-cac-van-de-thuong-gap-khi-su-dung-con-tro/0.png?raw=true" \o "0.png?raw=true)**

**[0.png?raw=true1233x540](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-11-cac-van-de-thuong-gap-khi-su-dung-con-tro/0.png?raw=true" \o "0.png?raw=true)**

Thật ra cũng không có ai rãnh rỗi đến mức gán địa chỉ sai cho con trỏ, điều mà mình muốn nói ở đây là khi chương trình chúng ta viết bị **crash** khi chạy, có thể lỗi này đến từ việc gán địa chỉ không hợp lý.

Trong thực tế, lỗi này sẽ xuất hiện dưới một hình thức khác. Ví dụ:

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

{

**int** \*p = **new** **int**[10];

cout << p << endl;

**delete**[] p;

cout << p << endl;

\_sleep(10000);

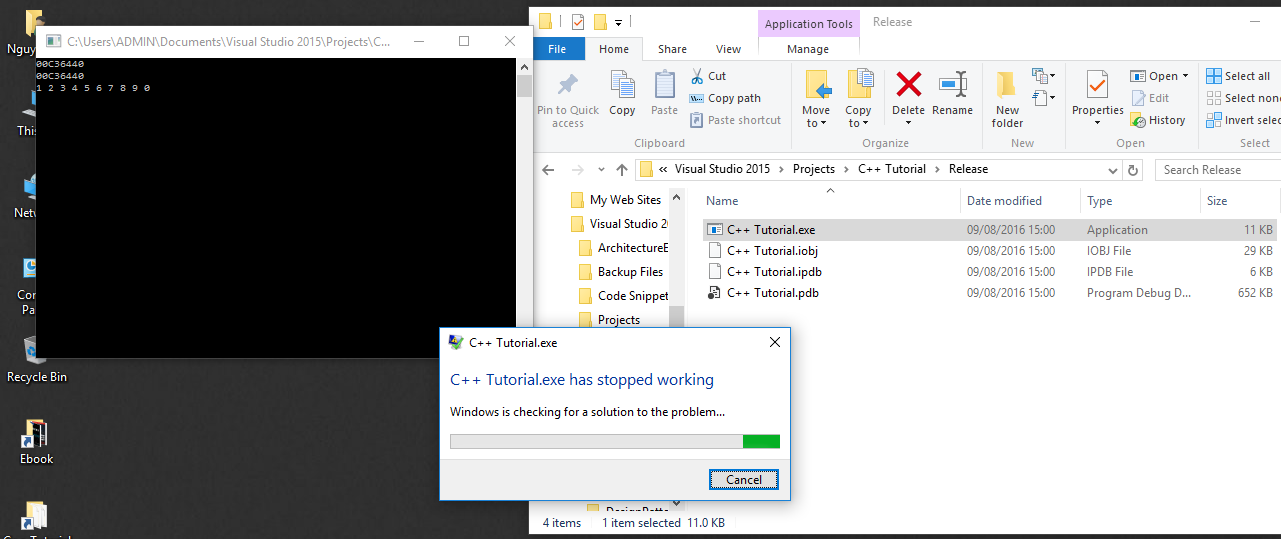
**for** (**int** i = 0; i < 10; i++)

cin >> p[i];

**return** 0;

}

Chương trình này yêu cầu cấp phát 1 dãy địa chỉ có kích thước **40 bytes trên Heap**. Sau đó in ra địa chỉ của vùng nhớ vừa được cấp phát thành công. Tiếp theo, mình không muốn sử dụng vùng nhớ này nữa nên mình trả lại cho hệ điều hành thông qua toán tử **delete**. Trên thực tế, có thể mình sẽ không giải phóng vùng nhớ ngay lập tức mà sử dụng xong rồi mới giải phóng nó đi. Sau khi giải phóng vùng nhớ đó, mình in ra lại địa chỉ mà con trỏ p đang nắm giữ thì thấy p vẫn đang trỏ đến địa chỉ mà mình đã giải phóng. Vậy là chúng ta đã gặp phải lỗi mà mình đưa ra ở ví dụ đầu tiên trong bài học này, đó là con trỏ trỏ đến một vùng nhớ không chịu sử quản lý của chương trình. Lúc này, mình chưa **dereference** đến vùng nhớ đó ngay lập tức mà vẫn cho chương trình tiếp tục thực thi. Cuối cùng, mình không nhớ rằng vùng nhớ ban đầu đã được giải phóng ở đâu đó nên vẫn tiếp tục sử dụng bằng cách nhập dữ liệu vào đó thông qua con trỏ p.

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-11-cac-van-de-thuong-gap-khi-su-dung-con-tro/1.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-11-cac-van-de-thuong-gap-khi-su-dung-con-tro/1.png?raw=true" \o "1.png?raw=true)**

**[1.png?raw=true1281x539](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-11-cac-van-de-thuong-gap-khi-su-dung-con-tro/1.png?raw=true" \o "1.png?raw=true)**

Kết quả là phần mềm đang chạy thì bị **crash** vì có thể hệ điều hành đã cấp phát vùng nhớ đã được giải phóng cho phần mềm khác sử dụng. Khi chạy chương trình dưới chế độ Debug trong Visual Studio, lỗi này có thể không phát hiện được do nó vẫn chạy bình thường mà không có thông báo mà cũng không bị crash. Điều này làm chúng ta tưởng rằng chương trình hoạt động tốt, và **build** nó ra thành phần mềm lỗi.

Để khắc phục trường hợp này, chúng ta nên cho con trỏ quản lý vùng nhớ được cấp phát trỏ về **NULL** ngay sau khi giải phóng vùng nhớ đó.

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

{

**int** \*p = NULL;

p = **new** **int**[10];

cout << p << endl;

**delete**[] p;

p = NULL;

cout << p << endl;

\_sleep(10000);

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

**for** (**int** i = 0; i < 10; i++)

cin >> p[i];

}

**return** 0;

}

Bất cứ khi nào sử dụng vùng nhớ thông qua con trỏ, chúng ta cũng nên kiểm tra xem con trỏ có khác NULL hay không. Nếu con trỏ khác NULL thì chúng ta hiểu rằng vùng nhớ đó vẫn chưa được giải phóng. Đây chỉ là một cách quy ước mình tự đặt ra giúp cách viết chương trình của mình an toàn hơn, cách của mình có thể khác với một số lập trình viên khác.

Nhưng lỗi này còn có thể xuất hiện dưới một hình thức khác nữa. Đó là sử dụng 2 con trỏ cùng trỏ đến một vùng nhớ trong chương trình.

**#include <iostream>**

**#include <cstring>**

**using** **namespace** std;

**char** \* **getName**(**char** \*fullname) {

**if** (fullName == NULL)

**return** NULL;

**char** \*pTemp = strrchr(fullname, ' ');

**if** (pTemp == NULL)

**return** fullname;

**else**

**return** pTemp + 1;

}

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

**char** \*fullName = **new** **char**[50];

cout << "Enter your full name: ";

cin.getline(fullName, 50);

cout << "Your last name is: ";

**char** \*name = getName(fullName);

**delete**[] fullName;

cout << name << endl;

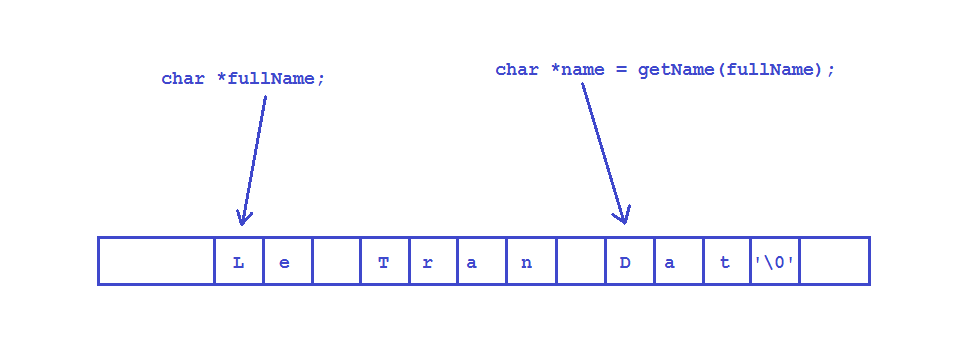
**return** 0;

}

Đoạn chương trình này thực hiện công việc yêu cầu người dùng nhập vào đầy đủ họ và tên, sau đó in ra tên mà người dùng vừa nhập vào (bỏ qua họ và tên đệm). Nhưng trước đó, mình cần yêu cầu cấp phát một vùng nhớ trên **Heap** đủ để người dùng nhập vào họ tên.

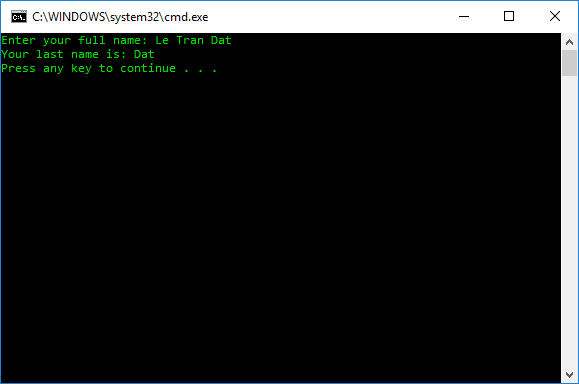
Cách hoạt động của hàm getName như sau:

(1) Nhận vào đối số là địa chỉ của một địa chỉ của một chuỗi kí tự, trong trường hợp này là full name của người dùng.  
(2) Sử dụng hàm **strrchr** trong thư viện **cstring** để trả về địa chỉ xuất hiện kí tự khoảng trắng cuối cùng trong chuỗi kí tự.  
(3) Nếu không có kí tự khoảng trắng thì trả về địa chỉ đầu tiên của chuỗi kí tự (cho rằng người dùng chỉ nhập tên chứ không nhập họ và tên đệm), nếu có xuất hiện khoảng trắng thì trả về địa chỉ của phần tử đứng sau khoảng trắng.

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-11-cac-van-de-thuong-gap-khi-su-dung-con-tro/2.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-11-cac-van-de-thuong-gap-khi-su-dung-con-tro/2.png?raw=true" \o "2.png?raw=true)**

**[2.png?raw=true958x360](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/8-con-tro/8-11-cac-van-de-thuong-gap-khi-su-dung-con-tro/2.png?raw=true" \o "2.png?raw=true)**

Như chúng ta thấy, địa chỉ của tên người dùng được trả về từ hàm **getName** cũng thuộc phạm vi vùng nhớ được cấp phát và đang được quản lý thông qua con trỏ **fullName**. Tuy nhiên, trong đoạn chương trình trên, lập trình viên này đã nghĩ rằng sau khi sử dụng xong hàm getName thì không cần sử dụng đến con trỏ **fullName** nữa, vậy là **delete** luôn vùng nhớ mà con trỏ **fullName** đang nắm giữ, dẫn đến việc con trỏ **name** đã trỏ tới một vùng nhớ không còn thuộc quyền quản lý của chương trình nữa. Và kết quả cho ra không đúng với mong đợi:



Tuy nhiên, khi mình chạy chương trình trên, kết quả vẫn đúng. Đó là do mình sử dụng vùng nhớ đó ngay sau khi trả lại cho hệ điều hành. Hệ điều hành lúc này vẫn chưa tác động gì đến vùng nhớ đã được giải phóng, hoặc nếu đã có cấp phát cho chương trình khác thì chương trình đó vẫn chưa thay đổi nội dung trong phạm vi này. Bây giờ mình giả sử chúng ta thực thi công việc khác, sau một vài giây sau mới cần in ra kết quả thì sẽ dễ phát hiện lỗi hơn:

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

**char** \*fullName = **new** **char**[50];

cout << "Enter your full name: ";

cin.getline(fullName, 50);

cout << "Your last name is: ";

**char** \*name = getName(fullName);

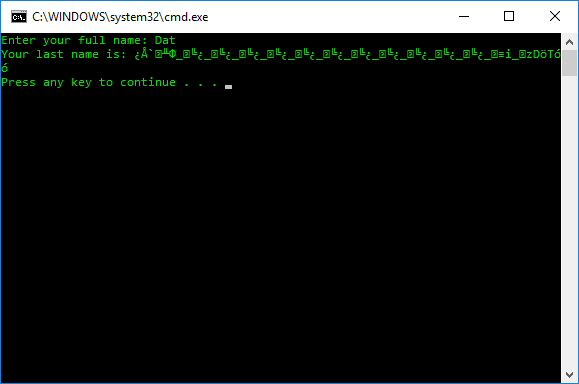
**delete**[] fullName;

\_sleep(5000);

cout << name << endl;

**return** 0;

}



Để khắc phục trường hợp này, chúng ta cần xác định rằng khi nào thực sự không còn sử dụng đến vùng nhớ nào đó thì mới giải phóng. Sửa lại đoạn chương trình trên như sau:

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

**char** \*fullName = **new** **char**[50];

cout << "Enter your full name: ";

cin.getline(fullName, 50);

cout << "Your last name is: ";

**char** \*name = getName(fullName);

cout << name << endl;

**delete**[] fullName;

fullName = NULL;

**return** 0;

}

Hoặc chúng ta vẫn muốn sử dụng tiếp vùng nhớ quản lý bởi con trỏ **fullName** mà không muốn sử dụng đến con trỏ **name** nữa, chúng ta nên sửa lại như sau:

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

**char** \*fullName = **new** **char**[50];

cout << "Enter your full name: ";

cin.getline(fullName, 50);

cout << "Your last name is: ";

**char** \*name = getName(fullName);

cout << name << endl;

name = NULL;

*//keep using fullName*

*//and then deallocate it*

**delete**[] fullName;

**return** 0;

}

Con trỏ **name** chỉ trỏ đến địa chỉ bên trong vùng nhớ được cấp phát cho con trỏ **fullName**, nên chúng ta không nên sử dụng toán tử **delete** cho con trỏ **name**.

Trường hợp con trỏ trỏ đến vùng nhớ không chịu sự quản lý của chương trình cũng thường xuất hiện khi trả về địa chỉ của biến cục bộ trong hàm.

**int** \* newIntValue(**int** **value** = 0)

{

**int** n = **value**;

**return** &n;

}

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

**int** \*pInt = newIntValue(0);

**return** 0;

}

Như các bạn đã biết, biến cục bộ sẽ bị đưa ra khỏi **Stack** khi ra khỏi phạm vi khối lệnh. Dó đó, địa chỉ của biến n trong hàm **newIntValue** vẫn được trả về trước khi bị hủy. Chúng ta nên thay bằng **Dynamic memory allocation:**

**int** \* **newIntValue**(**int** value = 0)

{

**return** **new** **int**(value);

}

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

{

**int** \*p = newIntValue(0);

**delete** p;

**return** 0;

}

##### memory leak

**Memory leak** là trường hợp cấp phát vùng nhớ cho chương trình (thường là cấp phát trên **Heap**) nhưng vùng nhớ không được sử dụng hoặc không được giải phóng. Điều này làm giảm dung lượng bộ nhớ có thể sử dụng được cho những chương trình khác, khiến các chương trình hoạt động chậm hơn hoặc có thể làm crash chương trình.

Đây là một ví dụ thường gặp ở những lập trình viên mới học về kỹ thuật **Dynamic memory allocation:**

**int** \*ptr = **new** **int**[10];

*//................*

ptr = NULL;

Trong đoạn chương trình này, lập trình viên tự ý cho con trỏ ptr trỏ đi nơi khác. Điều này dẫn đến việc vùng nhớ được cấp phát trước đó không thể quản lý được nữa. Muốn quản lý một vùng nhớ được cấp phát trên Heap, chúng ta cần sử dụng ít nhất một con trỏ. Nhưng trong trường hợp này, không còn con trỏ nào được dùng để quản lý vùng nhớ đã được cấp phát. Do đó, vùng nhớ được cấp phát chỉ có thể được giải phóng khi toàn bộ chương trình kết thúc.

Để khắc phục trường hợp này, chúng ta cần có một con trỏ khác thay thế vị trí của con trỏ **ptr** trước khi cho con trỏ **ptr** trỏ đi nơi khác:

int \*ptr = new int[10]*;*

//.................

int \*pTemp = ptr*;*

ptr = NULL*;*

Việc **resize** kích thước của vùng nhớ cũng có thể gây ra lỗi **memory leak** nếu sơ ý:

**void** **resizeArray**(**int** \*&p, **int** oldLength, **int** newLength)

{

**int** \*pTemp = p;

p = allocateArray(newLength);

*//copy data*

**if**(oldLength < newLength)

{

**for**(**int** i = 0; i < oldLength; i++)

{

p[i] = pTemp[i];

}

}

**else**

{

**for**(**int** i = 0; i < newLength; i++)

{

p[i] = pTemp[i];

}

}

}

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

**int** length = 10;

**int** \*p = **new** **int**[length];

**int** newLength = 20;

resizeArray(p, length, newLength);

**return** 0;

}

Khi cấp phát lại vùng nhớ, con trỏ p được gán vào địa chỉ mới, vùng nhớ ban đầu được con trỏ **pTemp** quản lý, nhưng khi ra khỏi hàm thì con trỏ pTemp bị hủy (vì **pTemp** cũng là biến cục bộ, được cấp phát trên **Stack**). Như vậy, vùng nhớ cũ không còn được quản lý nữa.

Chúng ta nên sửa lại đoạn chương trình trên như sau:

**void** **resizeArray**(**int** \*&p, **int** oldLength, **int** newLength)

{

**int** \*pTemp = p;

p = allocateArray(newLength);

*//copy data*

**if**(oldLength < newLength)

{

**for**(**int** i = 0; i < oldLength; i++)

{

p[i] = pTemp[i];

}

}

**else**

{

**for**(**int** i = 0; i < newLength; i++)

{

p[i] = pTemp[i];

}

}

**delete**[] pTemp;

}

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

**int** length = 10;

**int** \*p = **new** **int**[length];

**int** newLength = 20;

resizeArray(p, length, newLength);

**delete**[] p;

**return** 0;

}

Còn một trường hợp thường thấy nữa, đó là việc sử dụng sai toán tử delete cho con trỏ:

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

**int** \*p = **new** **int**[10];

**delete** p;

**return** 0;

}

Như các bạn thấy, chúng ta yêu cầu cấp phát một dãy vùng nhớ cho 10 phần tử kiểu **int**, nhưng khi giải phóng thì sử dụng toán tử **delete** để giải phóng một biến đơn. Visual Studio không báo lỗi cho trường hợp này, do đó lỗi này thường cũng khó nhận ra. Chúng ta nên sửa lại như sau:

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

**int** \*p = **new** **int**[10];

**delete**[] p;

**return** 0;

}

### Tổng kết

Trong bài học này, chúng ta đã cùng tìm hiểu một số nguyên nhân gây ra lỗi khi sử dụng con trỏ trong ngôn ngữ C++. Đây là một số lỗi thường gặp ở những người mới học lập trình C++. Như các bạn thấy, việc quản lý vùng nhớ một cách thủ công khá là phức tạp. Trong chuẩn C++ mới đã có hổ trợ cho chúng ta **Smart Pointer** giúp chúng ta tránh được những lỗi thường gặp này. Chúng ta sẽ tìm hiểu về **Smart Pointer** trong những bài học sau.

# 9.0 Kiểu liệt kê (enum)

#### Chào các bạn đang theo dõi khóa học lập trình trực tuyến ngôn ngữ C++.

Trong bài học ngày hôm nay, chúng ta sẽ cùng tìm hiểu về từ khóa **enum**, cách sử dụng, và quan trọng nhất là tại sao chúng ta sử dụng **enum** trong ngôn ngữ C++.

### Enumarated types

Kiểu liệt kê là một trong số các kiểu dữ liệu do người lập trình tự định nghĩa. **Tại sao chúng ta cần định nghĩa kiểu dữ liệu mới?** Trong quá trình lập trình, những kiểu dữ liệu được định nghĩa sẵn trong ngôn ngữ lập trình có thể không mang lại ý nghĩa phù hợp. Ví dụ mình muốn sử dụng các giá trị từ 1 đến 7 để đại diện cho 7 ngày trong tuần (1 đại diện cho ngày chủ nhật, 7 đại diện cho thứ 7), như vậy mình cần ít nhất là 7 biến để lưu trữ các giá trị này:

**const** **int** SUNDAY = 1;

**const** **int** MONDAY = 2;

**const** **int** TUESDAY = 3;

**const** **int** WEDNESDAY = 4;

**const** **int** THURSDAY = 5;

**const** **int** FRIDAY = 6;

**const** **int** SATURDAY = 7;

Mình không sử dụng mảng một chiều trong trường hợp này vì:

int DAYS\_OF\_WEEK[7] = { 1, 2, 3, 4, 5, 6, 7 };

Những con số cụ thể không mang lại ý nghĩa cho người đọc mã nguồn chương trình. Việc sử dụng tên của các biến hằng số sẽ giúp chương trình chúng ta rõ ràng hơn.

Nhưng việc khai báo các hằng số như trên vẫn có một số nhược điểm:

* Có thể khai báo thiếu sót một vài giá trị khi danh sách các hằng số là quá nhiều.
* Có thể khai báo không theo một quy luật (hay thứ tự) nhất định khiến chúng ta khó tìm trong chương trình. Ví dụ:

**const** **int** WEDNESDAY = 4;

**const** **int** SUNDAY = 1;

**const** **int** TUESDAY = 3;

**const** **int** FRIDAY = 6;

**const** **int** MONDAY = 2;

**const** **int** SATURDAY = 7;

**const** **int** THURSDAY = 5;

* Có một số hằng số không liên quan đến nhau nhưng được khai báo gần nhau khiến chúng ta dễ rối. Ví dụ:

**const** float PI = 3.14;

**const** float ACCELERATION\_OF\_GRAVITY = 9.8;

**const** int MAX\_SIZE\_OF\_ARRAY = 255;

*//..............*

Như vậy, muốn khắc phục một số nhược điểm trên, chúng ta cần tìm cách để tập hợp các hằng số có ý nghĩa tương đương nhau thành những nhóm hằng số riêng biệt. Kiểu liệt kê sẽ giúp chúng ta thực hiện điều này.

##### Công dụng của kiểu liệt kê

Như mình đã trình bày ở trên, kiểu liệt kê có tác dụng giúp thay thế các con số (giá trị cụ thể) bằng những cái tên có ý nghĩa, và nó còn giúp chúng ta tập hợp các giá trị có ý nghĩa liên quan với nhau thành từng nhóm. Mỗi nhóm hằng số này khi đưa vào kiểu liệt kê sẽ trở thành một kiểu dữ liệu (người ta thường gọi enumeration là một kiểu dữ liệu trong C++ vì nó có cách khai báo tương tự như khai báo biến, chứ mình thấy nó giống một group của các giá trị hơn).

##### Cú pháp khai báo kiểu liệt kê

Để định nghĩa một kiểu liệt kê mới, chúng ta sử dụng từ khóa **enum** theo cấu trúc sau:

**enum** <**name\_of\_enumeration**>

{

//list all **of** values inside this block

//each enumerator is separated by a comma, not a semicolon

};

Việc khai báo kiểu dữ liệu mới (như kiểu **enum**) không yêu cầu chương trình cấp phát bộ nhớ, lúc nào chúng ta sử dụng kiểu **enum** vừa đã được định nghĩa để tạo ra biến kiểu **enum** thì chương trình mới cấp phát bộ nhớ.

Mỗi giá trị trong block của kiểu enum cách nhau bởi một dấu phẩy (đối với giá trị cuối cùng thì không cần sử dụng dấu phẩy).

##### Khai báo kiểu liệt kê

Mình đã nói về lý thuyết của kiểu **enum** (kiểu liệt kê) xong, chắc bây giờ các bạn cũng đang tò mò muốn biết cuối cùng thì khai báo và sử dụng nó như thế nào. Dưới đây là một ví dụ:

**enum** **DaysOfWeek**

{

SUNDAY,

MONDAY,

TUESDAY,

WEDNESDAY,

THURSDAY,

FRIDAY,

SATURDAY

};

Như các bạn thấy, sau khi định nghĩa một kiểu enum xong thì kết thúc nó là một dấu chấm phẩy, vì đây cũng là một câu lệnh. Về mặt cơ bản, chúng ta phải đặt toàn bộ câu lệnh trên cùng một dòng:

**enum** **DaysOfWeek** { **SUNDAY**, **MONDAY**, **TUESDAY**, **WEDNESDAY**, **THURSDAY**, **FRIDAY**, **SATURDAY** };

Nhưng **compiler** vẫn hiểu được một câu lệnh nằm trên nhiều dòng nên mình chọn cách viết ở trên (tách thành nhiều dòng) để phần định nghĩa của mình rõ ràng hơn.

Như vậy là chúng ta đã có một kiểu dữ liệu mới cho chương trình. Các bạn có thể gọi **DaysOfWeek** là một kiểu dữ liệu (kiểu enum hay kiểu liệt kê) hoặc có thể gọi là tên của một nhóm các giá trị cũng như chúng ta hay đi chơi với bạn bè theo nhóm nhỏ rồi đặt tên cho nhóm vậy.

Trong một chương trình, chúng ta có thể có nhiều khai báo kiểu **enum** khác nhau. Ví dụ mình khai báo thêm vài kiểu **enum** khác:

**enum** **DaysOfWeek**

{

SUNDAY,

MONDAY,

TUESDAY,

WEDNESDAY,

THURSDAY,

FRIDAY,

SATURDAY

};

**enum** **Color**

{

RED,

GREEN,

BLUE,

WHITE

};

**enum** **Animal**

{

CAT,

DOG,

HORSE,

MONKEY,

CHICKEN

};

Như vậy là chương trình của chúng ta có 3 kiểu dữ liệu mới (3 nhóm giá trị mới), mỗi kiểu enum này hoàn toàn không liên quan gì đến nhau, chỉ có các giá trị bên trong mỗi kiểu enum mới có liên quan đến nhau về mặt ý nghĩa.

**Nhưng có thấy giá trị nào đâu?**

Khi nhìn vào bên trong khối lệnh định nghĩa của kiểu enum có tên **Color**, chúng ta chỉ thấy những những danh từ như RED, GREEN, BLUE... mà không hề thấy những con số. Thực chất, những danh từ này đã được gắn cho một giá trị cụ thể, và những cái danh từ mà chúng ta nhìn thấy sẽ đại diện cho những giá trị đó. Sử dụng những danh từ để thay thế cho những con số sẽ giúp người đọc chương trình dễ hiểu hơn (chứ không giúp chương trình chạy nhanh hơn).

##### Enumerator values

Bây giờ mình sẽ làm một chương trình mẫu để show cho các bạn xem những giá trị được đặt trong block của một kiểu enum mình tự định nghĩa:

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

**enum** Alphabet

{

LETTER\_A,

LETTER\_B,

LETTER\_C,

LETTER\_D,

LETTER\_E

};

cout << LETTER\_A << endl;

cout << LETTER\_B << endl;

cout << LETTER\_C << endl;

cout << LETTER\_D << endl;

cout << LETTER\_E << endl;

**return** 0;

}

Khi mình chạy chương trình, kết quả xuất hiện trên console là:



Như vậy, không cần chúng ta trực tiếp gán giá trị cho các tên hằng số, **compiler** đã tự động khởi tạo giá trị cho chúng, bắt đầu với giá trị 0 và tăng dần. Các bạn cũng đã thấy rằng, sau khi định nghĩa xong 1 kiểu enum thì chúng ta có thể sử dụng các tên gọi bên trong enum như những hằng số. Vì những giá trị hằng số này là giá trị kiểu integer (**int**), nên chúng ta cũng có thể gán chúng cho những biến kiểu **int** khác. Ví dụ:

**enum** **Alphabet**

{

LETTER\_A,

LETTER\_B,

LETTER\_C,

LETTER\_D,

LETTER\_E

};

int iValue = LETTER\_A;

Bên cạnh việc tự động gán giá trị cho từng phần tử được liệt kê, chúng ta cũng có thể chủ động thay đổi giá trị cho chúng (**nhưng chỉ có thể thay đổi giá trị trong phần khai báo**), một enum sau khi đã định nghĩa xong thì không thể thay đổi những giá trị của danh sách các phần tử nữa.

**enum** **Direction**

{

UP = 1, //assigned 1 by programmer

DOWN = 3, //assigned 3 by programmer

LEFT, //assigned 4 by compiler

RIGHT //assigned 5 by compiler

};

cout << UP << " " << DOWN << " " << LEFT << " " << RIGHT << endl;

Đoạn chương trình này sẽ in ra:

1 3 4 5

Như vậy, **compiler** sẽ tự động gán giá trị cho các phần tử không được khởi tạo giá trị. Ngoại trừ phần tử đầu tiên trong **enum**, những hằng số khác sẽ được gán giá trị bằng phần tử trước nó cộng thêm 1.

**Lưu ý: những hằng số trong cùng một enum có thể có cùng giá trị với nhau.**

Best practice: Don’t assign specific values to your enumerators.

Rule: Don’t assign the same value to two enumerators in the same enumeration unless there’s a very good reason.

##### Sử dụng kiểu enum đã định nghĩa như một kiểu dữ liệu thông thường

Như mình trình bày ở trên, từ khóa enum trong C++ giúp chúng ta định nghĩa một kiểu dữ liệu mới cho chương trình. Tuy nó chỉ là tập hợp danh sách các hằng số có ý nghĩa tương quan với nhau, nhưng bản chất nó vẫn là một kiểu dữ liệu (kiểu liệt kê) nên chúng ta có thể sử dụng chúng để tạo ra các biến. Ví dụ:

**enum** **Color**

{

COLOR\_BLACK,

COLOR\_RED,

COLOR\_BLUE,

COLOR\_GREEN,

COLOR\_WHITE,

COLOR\_CYAN,

COLOR\_YELLOW

};

Color backgroundColor;

Bây giờ, chúng ta đã có một biến kiểu **Color**. Biến **backgroundColor** chỉ có tác dụng lưu trữ giá trị của một trong số tất cả các hằng số đã được liệt kê bên trong kiểu **Color**. Việc thực hiện gán các giá trị khác kiểu **Color** sẽ gây ra lỗi về mặt cú pháp.

Color backgroundColor = 5; *//error*

Bây giờ mình sẽ chọn ra bất kì một hằng số thuộc kiểu Color để gán cho biến backgroundColor.

Color backgroundColor = COLOR\_GREEN;

Các bạn cần lưu ý rằng, biến kiểu enum chỉ có thể được gán giá trị là một trong số các hằng đã khai báo bên trong kiểu dữ liệu của chính nó, không thể sử dụng hằng của kiểu enum khác. Ví dụ:

**enum** **Test**

{

TEST1,

TEST2,

TEST3

};

**enum** **Color**

{

COLOR\_BLACK,

COLOR\_RED,

COLOR\_BLUE,

COLOR\_GREEN,

COLOR\_WHITE,

COLOR\_CYAN,

COLOR\_YELLOW

};

Color backgroundColor = TEST1; //this line makes an error

Compiler sẽ thông báo lỗi: "a value of type Test cannot be used to initialize an entity of type Color".

##### Những ví dụ về việc sử dụng enum

Sau khi đã gán giá trị cho biến kiểu enum, biến này sẽ mang giá trị là một số nguyên, và chúng ta có thể sử dụng biến này để in ra, tính toán, so sánh, truyền vào hàm theo kiểu giá trị, ... và còn nhiều mục đích khác.

Trên thực tế, chúng ta thường sử dụng kiểu enum để đưa ra lựa chọn hàm hoặc phương thức để thực thi. Ví dụ:

**enum** ItemTypes

{

LAPTOP,

DESKTOP,

MOBILE,

NETWORK

};

**void** **showAllLaptop**() {

}

**void** **showAllDesktop**() {

}

**void** **showAllMobile**() {

}

**void** **showAllNetworkItem**() {

}

**void** **showProducts**(ItemTypes type) {

**switch** (type)

{

**case** LAPTOP:

showAllLaptop();

**break**;

**case** DESKTOP:

showAllDesktop();

**break**;

**case** MOBILE:

showAllMobile();

**break**;

**case** NETWORK:

showAllNetworkItem();

**break**;

**default**:

**break**;

}

}

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

ItemTypes type = LAPTOP;

showProducts(type);

**return** 0;

}

Vì biến kiểu **enum** lưu trữ giá trị số nguyên, nên mình có thể đưa vào làm biểu thức mệnh đề cho câu lệnh **switch-case**. Dựa trên loại Item mà người dùng chọn, ứng dụng của chúng ta sẽ trả về thông tin của toàn bộ sản phẩm hiện có trong kho hàng.

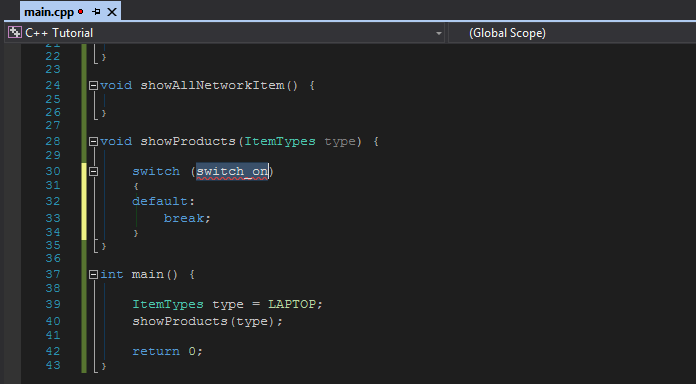
Thông thường, sau mỗi nhãn case chúng ta sẽ đặt một giá trị số nguyên ứng với mỗi trường hợp, nhưng bây giờ chúng ta có thể thay thế những con số bằng các định danh của enum. Trước đây khi làm việc với câu lệnh switch-case, chúng ta có thể bỏ sót một số trường hợp cần xem xét nếu số lượng các trường hợp là quá nhiều. Nhưng khi sử dụng Visual studio 2015 và kiểu enum, chúng ta sẽ tránh được sự thiếu sót này. Dưới đây là cách mà Visual studio 2015 hổ trợ cho kiểu enum:

Trong hàm **showProducts**, mình gõ câu lệnh **switch** nhưng sử dụng gợi ý của **Visual studio**.

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/9-kieu-du-lieu-tu-dinh-nghia/9-0-kieu-liet-ke/3.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/9-kieu-du-lieu-tu-dinh-nghia/9-0-kieu-liet-ke/3.png?raw=true" \o "3.png?raw=true)**

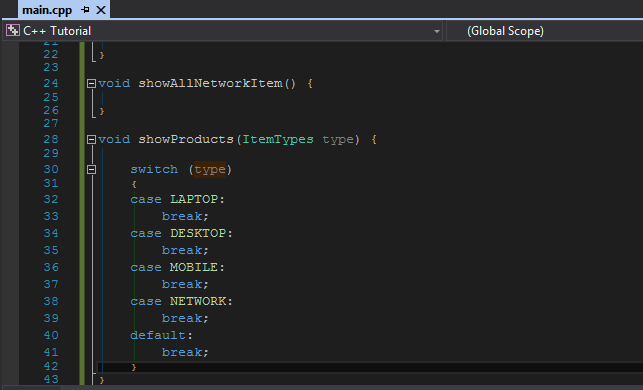
**[3.png?raw=true753x374](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/9-kieu-du-lieu-tu-dinh-nghia/9-0-kieu-liet-ke/3.png?raw=true" \o "3.png?raw=true)**

Sau khi chọn vào gợi ý của lệnh **switch**, IDE phát sinh code cho chúng ta như sau:

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/9-kieu-du-lieu-tu-dinh-nghia/9-0-kieu-liet-ke/4.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/9-kieu-du-lieu-tu-dinh-nghia/9-0-kieu-liet-ke/4.png?raw=true" \o "4.png?raw=true)**

**[4.png?raw=true696x384](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/9-kieu-du-lieu-tu-dinh-nghia/9-0-kieu-liet-ke/4.png?raw=true" \o "4.png?raw=true)**

Lúc này, các bạn chỉ cần gõ thay thế từ switch\_on bằng tên của biến **enum** rồi nhấn **phím mũi tên sang phải**, IDE sẽ liệt kê tất cả các **case** ứng với tất cả giá trị được định nghĩa bên trong kiểu của biến enum đó.



Đây cũng là một ưu điểm của Visual studio 2015. Bây giờ chúng ta trở lại với bài học.

Thêm một ví dụ khác cũng có thể có ích. Xét đoạn chương trình dưới đây:

**bool** **initialize**() {

*//init all component*

*//if something wrong, return false*

**return** false;

}

**bool** **loadResource**() {

*//load data from file*

*//if something wrong, return false*

**return** false;

}

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

**if** (initialize() == false) {

**return** -1;

}

**if** (loadResource() == false) {

**return** -2;

}

**bool** isRunning = true;

**while** (isRunning) {

*//Application event loop*

**return** -3;

}

**return** 0;

}

Đoạn chương trình này sử dụng những giá trị âm để biểu diễn các lỗi có thể xảy ra. Mỗi chức năng trong chương trình gây ra lỗi thì chương trình sẽ trả về một giá trị khác nhau. Tuy nhiên, những con số cụ thể thường không mang nhiều ý nghĩa, do đó chúng ta thay thế chúng bằng kiểu enum để code của chúng ta rõ ràng hơn.

**enum** **ReturnValue**

{

SUCCESS = 0,

INITIALIZE\_ERROR = -1,

LOAD\_RESOURCE\_ERROR = -2,

RUN\_TIME\_ERROR = -3

};

int main() {

**if** (initialize() == false) {

**return** INITIALIZE\_ERROR;

}

**if** (loadResource() == false) {

**return** LOAD\_RESOURCE\_ERROR;

}

bool isRunning = true;

**while** (isRunning) {

//Application event loop

**return** RUN\_TIME\_ERROR;

}

**return** SUCCESS;

}

Khi chúng ta viết một game với nhân vật có nhiều trạng thái khác nhau, mỗi trạng thái sẽ khiến nhân vật phản ứng bằng một hành động tương ứng, chúng ta có thể làm như sau:

**enum** BossState

{

IDLING,

RUNNING,

JUMPING,

DYING

};

BossState state;

**void** **initBoss**()

{

*//init something*

state = IDLING;

}

**void** **attack**()

{

*//............*

}

**void** **activated**()

{

*//............*

}

**void** **updateAnimation**(BossState state)

{

**switch**(state)

{

**case** IDLING:

standStill();

**break**;

**case** RUNNING:

setRunningAnimation();

**break**;

**case** JUMPING:

setJumpingAnimation();

**break**;

**case** DYING:

setDyingAnimation();

**break**;

**default**;

**break**;

}

}

Như các bạn thấy, kiểu enum được áp dụng khá thường xuyên trong thực tế. Trên đây chỉ là một vài ví dụ minh họa cho việc sử dụng kiểu enum thường gặp.

##### Phạm vi sử dụng kiểu enum

Trong một file chương trình, phạm vi sử dụng của một khai báo enum cũng tương tự như phạm vi sử dụng khi khai báo biến. Nếu chúng ta muốn sử dụng kiểu enum tại tất cả các khối lệnh trong chương trình, chúng ta nên khai báo kiểu enum phía trên cùng của các khối lệnh (giống như khai báo biến toàn cục). Ví dụ:

**enum** ItemTypes

{

LAPTOP,

DESKTOP,

MOBILE,

NETWORK

};

**void** **foo**()

{

cout << MOBILE << endl;

}

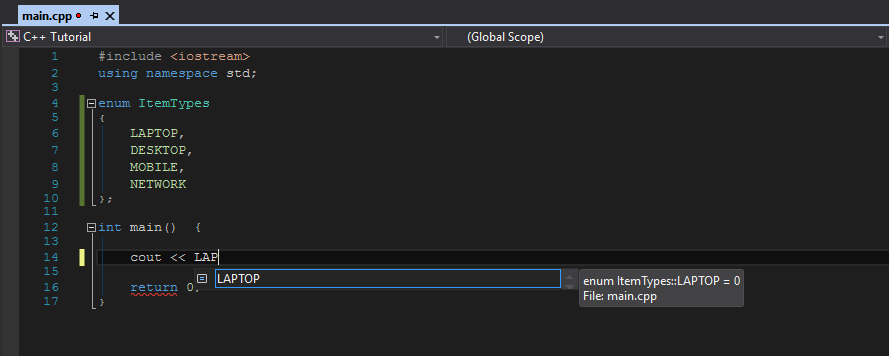
**int** **main**() {

cout << LAPTOP << endl;

cout << DESKTOP << endl;

**return** 0;

}

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/9-kieu-du-lieu-tu-dinh-nghia/9-0-kieu-liet-ke/1.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/9-kieu-du-lieu-tu-dinh-nghia/9-0-kieu-liet-ke/1.png?raw=true" \o "1.png?raw=true)**

**[1.png?raw=true889x356](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/9-kieu-du-lieu-tu-dinh-nghia/9-0-kieu-liet-ke/1.png?raw=true" \o "1.png?raw=true)**

Vì kiểu **ItemTypes** khai báo bên ngoài các khối lệnh, nên chúng ta có thể sử dụng tại các khối lệnh của các hàm bên dưới. Một trường hợp khác, khi mà enum chỉ được định nghĩa trong một khối lệnh của hàm nào đó:

void foo()

{

**enum** **ItemTypes**

{

LAPTOP,

DESKTOP,

MOBILE,

NETWORK

};

cout << MOBILE << endl;

}

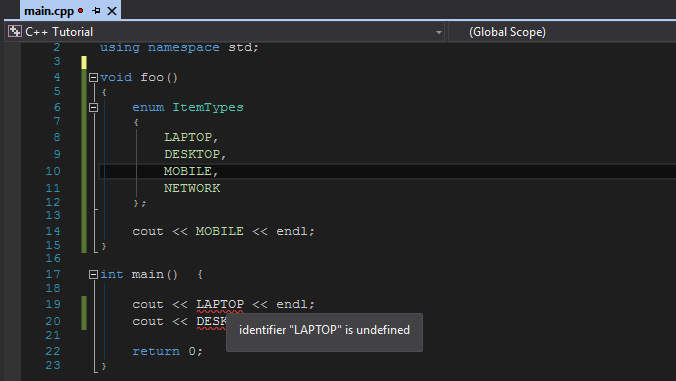
int main() {

cout << DESKTOP << endl; //error

**return** 0;

}

**Compiler** sẽ thông báo lỗi DESKTOP trong hàm main chưa được định nghĩa, vì chúng ta không thể truy xuất các giá trị của kiểu **ItemTypes** trong hàm foo từ hàm main được.



Thông thường, chúng ta nên định nghĩa kiểu dữ liệu enum bên ngoài các khối lệnh, vì việc khai báo kiểu enum mới không yêu cầu cấp phát bộ nhớ nên không hề ảnh hưởng đến tài nguyên của hệ thống.

Việc định nghĩa kiểu enum bên ngoài các khối lệnh không những có thể sử dụng bất cứ đâu trong chương trình mà còn có thể sử dụng tại file chương trình khác của dự án. Chúng ta sẽ tìm hiểu về cách quản lý dự án với nhiều file C++ trong các bài học sau.

### Enum class

Việc sử dụng kiểu enum tự định nghĩa đã giúp chúng ta tổ chức chương trình rõ ràng, dễ đọc hơn. Nhưng khi một chương trình có nhiều enum được định nghĩa, sẽ có nhiều giá trị trùng nhau giữa các enum khác nhau, và nó có thể làm cho chương trình không có ý nghĩa gì mặc dù đã sử dụng kiểu enum.

**enum** **Color**

{

RED,

GREEN,

BLUE

};

**enum** **Fruit**

{

APPLE,

BANANA

};

int main() {

Color color = GREEN;

Fruit fruit = BANANA;

**if** (color == fruit)

cout << "It's the same" << endl;

**else**

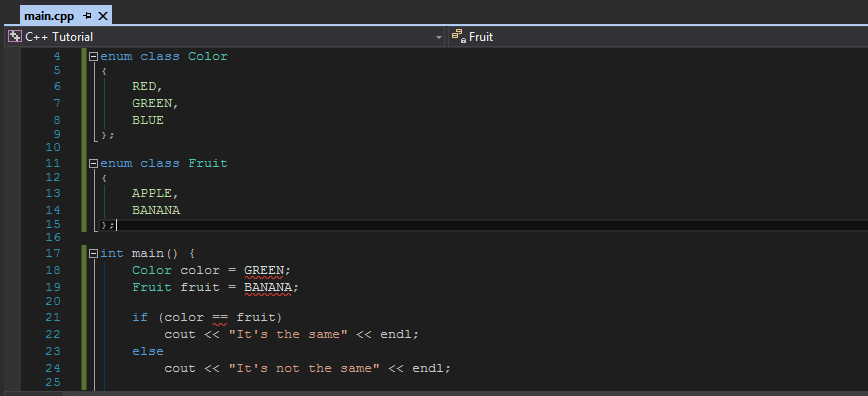
cout << "It's not the same" << endl;

**return** 0;

}

Trong hàm main, **compiler** sẽ so sánh **color và fruit** như 2 giá trị số nguyên, khi Color::GREEN và Fruit::BANANA đều được gán giá trị là 2 thì color và fruit được compiler cho là bằng nhau. Đây là một trường hợp ngoài ý muốn khi sử dụng kiểu enum. Điều này xảy ra vì enum Color và enum Fruit có thể truy cập đồng thời trong cùng một khối lệnh.

Điều chúng ta mong muốn lúc này là biến của kiểu Color chỉ được so sánh với giá trị trong enum Color, và biến kiểu Fruit chỉ có thể so sánh với giá trị trong enum Fruit. Chuẩn C++11 đã hổ trợ cho chúng ta một khái niệm enum mới, đó là enum class (có thể gọi là scoped enumeration). Mình sử dụng lại ví dụ trên nhưng thay thế enum bằng enum class:

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/9-kieu-du-lieu-tu-dinh-nghia/9-0-kieu-liet-ke/6.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/9-kieu-du-lieu-tu-dinh-nghia/9-0-kieu-liet-ke/6.png?raw=true" \o "6.png?raw=true)**

**[6.png?raw=true868x396](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/9-kieu-du-lieu-tu-dinh-nghia/9-0-kieu-liet-ke/6.png?raw=true" \o "6.png?raw=true)**

Như các bạn thấy, compiler đã thông báo lỗi ở câu lệnh if, đồng thời cũng thông báo lỗi ở 2 câu lệnh gán giá trị cho 2 biến color và fruit. Vì việc sử dụng enum class cần phải cung cấp thêm cho compiler biết là giá trị của enum đó được định nghĩa bên trong enum nào, phải cung cấp cho compiler một cái tên của kiểu enum đứng trước giá trị chúng ta muốn sử dụng (ví dụ Color::RED).

Các bạn đã được biết toán tử "::" là toán tử chỉ phạm vi truy cập. Khi mình sử dụng Color::RED có nghĩa là RED được định nghĩa bên trong khối lệnh của Color (không cần biết Color là gì, chúng ta chỉ biết RED nằm trong Color). Việc sử dụng enum class đòi hỏi lập trình viên phải chỉ ra đích danh của kiểu enum chứa giá trị cần sử dụng. Mình sửa lại đoạn chương trình trên như sau:

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

Color color = Color::GREEN;

Fruit fruit = Fruit::BANANA;

**if** (color == fruit)

cout << "It's the same" << endl;

**else**

cout << "It's not the same" << endl;

**return** 0;

}

Đến đây, 2 phép gán đã có thể thực thi được nhưng câu lệnh if thì còn lỗi. Đó là do compiler đã phát hiện ra 2 biến này có 2 kiểu dữ liệu khác nhau, 1 cái là của kiểu Color trong khi cái kia là kiểu Fruit. Compiler không chấp nhận điều này nên đưa ra thông báo lỗi.

Thậm chí khi các bạn sử dụng biến kiểu enum class để so sánh với một số nguyên cũng không được cho phép.

**if** *(color == 2)* *// error*

{

}

Chỉ có một cách duy nhất để sử dụng enum class là sử dụng giá trị trong chính enum của nó.

**if**(**color** == **Color**::GREEN)

{

*//OK*

}

Nếu các bạn sử dụng compiler C++11 trở lên, không có lý do nào mà các bạn sử dụng kiểu enum thông thường thay vì sử dụng enum class.

### Tổng kết

Trong bài học này, chúng ta đã tìm hiểu một số khái niệm về kiểu dữ liệu tự định nghĩa bằng từ khóa enum:

* Cú pháp khai báo, giá trị khởi tạo cho các thành phần của kiểu enum.
* Một số cách sử dụng enum thường gặp.
* Phân biệt enum và enum class trong chuẩn C++11.
* Visual studio hổ trợ cho chúng ta liệt kê tất cả các giá trị cần so sánh trong mệnh đề switch-case. Điều này có nghĩa chúng ta nên sử dụng switch-case thay vì if-else khi cần phân loại biến kiểu enum.

Sử dụng kiểu enum không làm cho chương trình của các bạn chạy nhanh hơn, cũng không làm cho chương trình của các bạn ngắn gọn hơn, nó chỉ có tác dụng duy nhất là làm cho chương trình của các bạn rõ ràng hơn.

# 9.1 Structs

#### Chào các bạn đang theo dõi khóa học lập trình trực tuyến C++.

Như các bạn đã biết, việc lập trình ứng dụng phần mềm chỉ đơn giản là tạo ra chương trình máy tính dùng để giải quyết một vấn đề nào đó trong cuộc sống. Để giải quyết được vấn đề, chúng ta cần cung cấp cho chương trình một lượng dữ liệu cần thiết, các câu lệnh sẽ xử lý các dữ liệu được đưa vào và cho ra kết quả mà người dùng mong muốn.

Dữ liệu chúng ta thu thập được trong cuộc sống cần được biểu diễn theo một định dạng nào đó mà máy tính có thể hiểu được, và ngôn ngữ lập trình cung cấp cho chúng ta điều này, đó chính là kiểu dữ liệu. Mỗi kiểu dữ liệu sẽ có một định dạng khác nhau, và khi tạo ra những thực thể từ những kiểu dữ liệu khác nhau, chúng sẽ có định dạng khác nhau dựa trên kiểu dữ liệu mô tả chúng. Ví dụ kiểu số thực (float, double) sẽ có định dạng phần thập phân trong khi kiểu số nguyên (int, long, ...) thì không có.

Việc chọn kiểu dữ liệu phù hợp để lưu trữ dữ liệu cần xử lý là rất quan trọng. Nhưng số lượng kiểu dữ liệu mà một ngôn ngữ lập trình (trong đó có C++) hổ trợ sẵn là khá hạn chế trong khi có những dữ liệu đặc tả cho vấn đề trong cuộc sống lại rất phức tạp. Ví dụ mình muốn biểu diễn một vài thông tin cá nhân của mình trên máy tính, mình có thể làm như sau:

std::string name;

std::string currentJob;

std::string homeAddress;

**int** birthYear;

**int** birthMonth;

**int** birthDay;

**float** height;

**float** weight;

*//...............*

Nhìn vào đoạn chương trình trên, các bạn có thể thấy mình cần sử dụng đến 3 kiểu dữ liệu khác nhau để tạo ra 8 biến chỉ để lưu trữ một lượng thông tin cá nhân không đầy đủ của một cá thể nào đó. Những biến này hoàn toàn độc lập với nhau, giả sử những biến này được khai báo tại những vị trí khác nhau trong chương trình sẽ rất khó quản lý. Và sẽ rắc rối hơn nếu chúng ta muốn lưu trữ thông tin cá nhân của nhiều hơn một người, lúc này chúng ta cần khai báo thêm 8 biến tương tự như trên, hoặc sử dụng mảng một chiều như sau:

std::string name[10];

std::string currentJob[10];

std::string homeAddress[10];

**int** birthYear[10];

**int** birthMonth[10];

**int** birthDay[10];

**float** height[10];

**float** weight[10];

*//...............*

Như các bạn thấy, việc quản lý chương trình trở nên phức tạp so với những người mới học lập trình. Do đó, việc tự định nghĩa một kiểu dữ liệu mới phù hợp cho đặc thù của chương trình của mỗi người là điều cần thiết. Rất may mắn, ngôn ngữ C++ hổ trợ chúng ta tự định nghĩa kiểu dữ liệu mới từ những kiểu dữ liệu built-in. Kiểu dữ liệu mới mà chúng ta sẽ định nghĩa được tạo thành từ một hoặc một nhóm kiểu dữ liệu xây dựng sẵn để tạo ra một tập hợp các biến thuộc cùng nhóm, những biến cùng nhóm này dùng để lưu trữ các dữ liệu có liên quan với nhau trong kiểu dữ liệu mới. Chúng ta gọi kiểu dữ liệu tập hợp này là **struct**.

### Struct

Một struct (viết tắt của structure) cho phép chúng ta nhóm nhiều biến của nhiều kiểu dữ liệu khác nhau để lưu trữ một tập hợp các dữ liệu cần thiết cho việc mô tả một đơn vị nào đó.

##### Khai báo struct

Để khai báo một cấu trúc mới (kiểu dữ liệu mới), chúng ta sử dụng từ khóa **struct**. Mặc dù một struct là một kiểu dữ liệu do lập trình viên tự định nghĩa, nó cũng cần được khai báo theo một cú pháp nhất định để compiler có thể hiểu được. Dưới đây là cú pháp để tạo ra một struct mới:

**struct** <**name\_of\_new\_type**>

{

<variables>;

};

Trong đó:

* struct là từ khóa mà ngôn ngữ C++ cung cấp.
* name\_of\_new\_type sẽ là tên của kiểu dữ liệu mới. Sau khi khai báo xong một struct, chúng ta có thể dùng tên struct để khai báo biến như những kiểu dữ liệu thông thường.
* variables là danh sách các biến dùng để lưu trữ dữ liệu phù hợp với yêu cầu lưu trữ dữ liệu của một đơn vị nào đó.

Mình lấy một ví dụ để các bạn có thể dễ hình dung hơn:

**#include <iostream>**

**#include <string>**

**struct** VietNamPeople

{

\_\_int32 ID;

std::string name;

\_\_int16 age;

**float** height;

**float** weight;

**bool** isStudent;

};

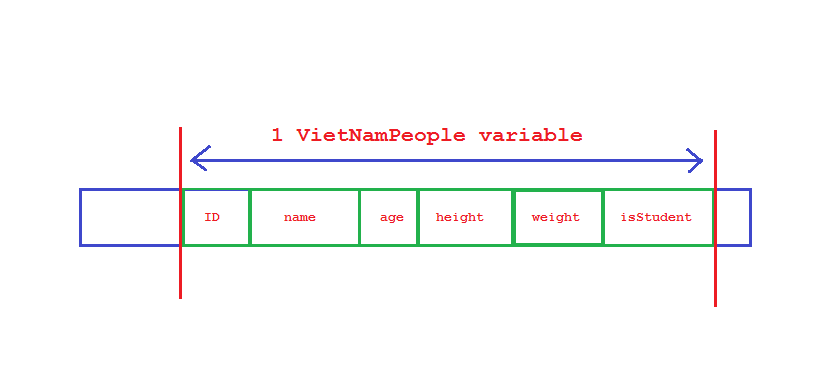
Như vậy, mình vừa định nghĩa xong một struct có tên là VietNamPeople, struct này bây giờ được coi là một kiểu dữ liệu mới là một tập hợp các biến ID, name, age, height, weight và isStudent. Những biến này được đặt vào chung một nhóm và mỗi biến sẽ lưu trữ một phần thông tin của một đơn vị là một con người Việt Nam.

Các bạn cần lưu ý về phạm vi sử dụng của kiểu dữ liệu tự định nghĩa cũng tương tự phạm vi sử dụng của biến trong chương trình, nhưng việc định nghĩa kiểu dữ liệu không yêu cầu hệ điều hành cấp phát bộ nhớ nên hoàn toàn không làm ảnh hưởng đến tài nguyên của hệ thống. Do đó, chúng ta nên định nghĩa kiểu dữ liệu mới cho phạm vi toàn cục (global scope) để kiểu dữ liệu mới này có thể được sử dụng trong toàn bộ file, thậm chí là sử dụng trong những file mã nguồn khác trong cùng project.

Khi các biến được đặt trong struct, chúng ta gọi chúng là trường dữ liệu của struct (fields). Một trường dữ liệu là một thành phần trong tập hợp các biến lưu trữ dữ liệu cần thiết cho một đơn vị. Ví dụ kiểu dữ liệu VietNamPeople có 6 trường dữ liệu là ID, name, age, height, weight và isStudent. Compiler có thể hiểu rằng khi một biến được tạo ra từ kiểu dữ liệu VietNamPeople, ví dụ:

VietNamPeople leTranDat*;*

thì lúc này, leTranDat chỉ là một cái tên của một đơn vị được tạo thành từ tập hợp các trường dữ liệu ID, name, age, height, weight và isStudent mà mình đã định nghĩa cho kiểu dữ liệu VietNamPeople. Các bạn có thể hình dung như thế này:

**[[](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/9-kieu-du-lieu-tu-dinh-nghia/9-1-structs/0.png?raw=true)](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/9-kieu-du-lieu-tu-dinh-nghia/9-1-structs/0.png?raw=true" \o "0.png?raw=true)**

**[0.png?raw=true825x390](https://github.com/nguyenchiemminhvu/CPP-Tutorial/blob/master/9-kieu-du-lieu-tu-dinh-nghia/9-1-structs/0.png?raw=true" \o "0.png?raw=true)**

Đây chỉ là hình ảnh minh họa cho việc tổ chức dữ liệu các trường bên trong một biến kiểu VietNamPeople sau khi được tạo ra. Trên thực tế các trường sẽ có kích thước khác so với dự đoán (khi đến phần nâng cao của struct mình sẽ trình bày về vấn đề này), nhưng đối với các bạn mới học thì chưa cần quan tâm những điều này.

Chúng ta có thể sử dụng kiểu dữ liệu VietNamPeople mà mình định nghĩa ở trên để tạo ra nhiều biến struct khác nhau:

VietNamPeople leTranDat*;*

VietNamPeople dayNhauHoc*;*

VietNamPeople ngoDoanTuan*;*

//.....................

Như những biến thông thường, các biến struct này sẽ được cấp phát bộ nhớ tùy vào cách chọn kỹ thuật cấp phát.

##### Khởi tạo cho biến struct

Khởi tạo giá trị cho các trường dữ liệu trong một biến struct rắc rối hơn khởi tạo giá trị cho biến thông thường một chút. Ngôn ngữ C++ đã hổ trợ cho chúng ta một cách nhanh hơn là sử dụng một **initializer list**. Nó cho phép các bạn khởi tạo một hoặc nhiều trường dữ liệu trong một khai báo biến struct. Ví dụ:

**struct** Employee

{

\_\_int32 ID;

std::string name;

\_\_int32 age;

\_\_int32 year\_of\_exp;

};

*//................*

Employee leTranDat = { 1, "Le Tran Dat", 28, 5 };

Các trường dữ liệu được khởi tạo lần lượt từ trên xuống dưới như trong phần định nghĩa struct có tên Employee. Lúc này, biến leTranDat sẽ chứa các thông tin được khởi tạo lần lượt là: ID = 1, name = "Le Tran Dat", age = 28 và year\_of\_exp = 5.

Nếu initializer list không cung cấp đủ dữ liệu cho các trường dữ liệu, giá trị mặc định sẽ được dùng để khởi tạo. Ví dụ:

Employee newEmp = { 1, "new employee" }; //age = 0, year\_of\_exp = 0 by default

Một tập hợp các giá trị của một biến struct được gọi là một Record (bản ghi).

##### Truy cập các trường dữ liệu của biến struct

Xem xét về struct Employee mình đã định nghĩa ở trên:

struct Employee

{

\_\_int32 ID*;*

std::string name*;*

\_\_int32 age*;*

\_\_int32 year\_of\_exp*;*

}*;*

Kiểu dữ liệu Employee mô tả rằng mỗi biến kiểu Employee được tạo ra sẽ bao gồm 4 trường dữ liệu là ID, name, age và year\_of\_exp. Như vậy, bất kỳ biến nào có kiểu Employee đều có đủ 4 trường dữ liệu trên.

Muốn truy xuất đến các trường dữ liệu của một biến struct, chúng ta sử dụng **member selection operator** (dấu chấm). Dưới đây là một ví dụ:

Employee leTranDat*;*

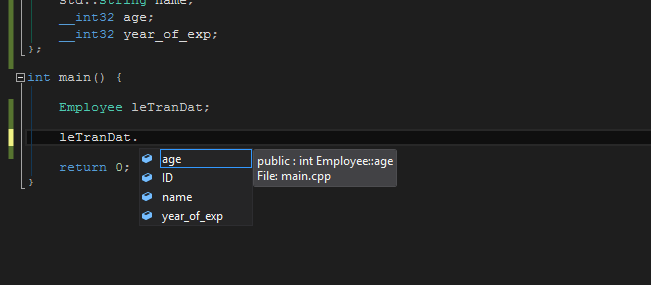
leTranDat.ID = 1*;*

leTranDat.name = "Le Tran Dat"*;*

leTranDat.age = 28*;*

leTranDat.year\_of\_exp = 5*;*

Visual studio sẽ hổ trợ chúng ta liệt kê tất cả các trường dữ liệu của một biến struct khi sử dụng **member selection operator**.



Các trường dữ liệu của một biến struct cũng là những biến thông thường, nhưng nó được gói gọn bên trong một biến struct, nên chúng ta phải sử dụng tên của biến struct và **member selection operator** để truy xuất đến chúng.

Như vậy, thông qua tên biến struct, các trường dữ liệu được nhóm lại giúp chúng ta biết được trường dữ liệu đó được dùng cho đơn vị nào, điều này giúp chúng ta dễ dàng tổ chức chương trình ở quy mô lớn hơn.

Vì các trường dữ liệu của một biến struct cũng là những biến thông thường, chúng ta cũng có thể sử dụng chúng để tính toán, so sánh, ...

Employee leTranDat = { 1, "Le Tran Dat", 28, 5 };

Employee juniorEmp = { 2, "New employee", 25, 1 };

**if** (leTranDat.year\_of\_exp > juniorEmp.year\_of\_exp)

{

std::cout << leTranDat.name << " has more experience than " << juniorEmp.name << std::endl;

}

Các trường dữ liệu của một struct sẽ tồn tại cùng với biến struct cho đến khi biến struct ra khỏi phạm vi sử dụng và bị hủy. Do đó, khi biến struct còn tồn tại, chúng ta vẫn có thể truy xuất đến các trường dữ liệu của nó.

##### Nhập và xuất dữ liệu cho biến struct

Cũng tương tự như nhập xuất dữ liệu cho biến thông thường, chỉ khác là chúng ta cần sử dụng thêm tên biến struct và **member selection operator** để compiler biết chúng ta nhập xuất cho trường dữ liệu của đơn vị nào. Ví dụ:

Employee emp;

*//Input*

std::cout << "Enter ID: ";

std::cin >> emp.ID;

std::cout << "Enter name: ";

std::getline(std::cin, emp.name);

std::cout << "Enter age: ";

std::cin >> emp.age;

std::cout << "Enter year of experience: ";

std::cin >> emp.year\_of\_exp;

*//Output*

std::cout << "===================================" << std::endl;

std::cout << emp.ID << std::endl;

std::cout << emp.name << std::endl;

std::cout << emp.age << std::endl;

std::cout << emp.year\_of\_exp << std::endl;

std::cout << "===================================" << std::endl;

##### Structs và function

Một ưu điểm khi sử dụng struct là chúng ta không cần truyền tất cả các trường dữ liệu của một đơn vị nào đó mà chỉ cần sử dụng một biến struct làm tham số cho hàm. Ví dụ:

**struct** Vector2D

{

**float** x;

**float** y;

};

**void** **printVector2D**(Vector2D vec)

{

std::cout << "(" << vec.x << "," << vec.y << ")" << std::endl;

}

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

{

Vector2D vec = { 1, 4 };

printVector2D(vec);

**return** 0;

}

Trong ví dụ trên, mình sử dụng kiểu truyền dữ liệu giá trị nên tham số Vector2D vec của hàm printVector2D không làm thay đổi giá trị gốc của đối số. Các bạn cũng có thể thử truyền đối số là biến struct theo kiểu tham chiếu hoặc con trỏ, về mặt cơ bản, biến struct cũng là một biến có địa chỉ cụ thể nên chúng ta làm hoàn toàn tương tự như biến thông thường.

void normalize(Vector2D &vec)

{

float length = sqrt((vec.x \* vec.x) + (vec.y \* vec.y));

vec.x = vec.x / length;

vec.y = vec.y / length;

}

int main()

{

Vector2D vec = { 1, 4 };

printVector2D(vec);

normalize(vec);

printVector2D(vec);

**return** 0;

}

Kiểu struct cũng có thể được dùng làm kiểu trả về của hàm. Ví dụ:

Vector2D addTwoVector(Vector2D vec1, Vector2D **vec2**)

{

Vector2D result = { vec1.x + **vec2**.x, vec1.y + **vec2**.y };

**return** result;

}

**int** main()

{

Vector2D vec1 = { 1, 2 };

Vector2D **vec2** = { 2, 2 };

Vector2D result = addTwoVector(vec1, **vec2**);

**return** 0;

}

Chúng ta có thể làm như trên vì ngôn ngữ C++ cho phép chúng ta gán biến struct cho một biến cùng kiểu struct khác. Ví dụ:

Vector2D vec1 = { 1, 2 };

Vector2D vec2 = vec1;

Nhưng chúng ta không nên sử dụng phép gán trực tiếp như vậy, vì có thể trong struct còn có các yếu tố phức tạp khác như con trỏ, hoặc struct khác, ... và dễ gây ra sai sót. Quay trở lại với chủ đề mình đang trình bày.

Chúng ta còn có thể định nghĩa các hàm bên trong phần định nghĩa của struct. Ví dụ:

struct Vector2D

{

**float** x;

**float** y;

**void** normalize()

{

**float** length = sqrt(x \* x + y \* y);

x = x / length;

y = y / length;

}

};

Trong ví dụ trên, hàm normalize được định nghĩa trong cùng khối lệnh của struct Vector2D, nên nó có thể trực tiếp truy cập đến biến x và y và thao tác với chúng. Nhưng x và y của struct Vector2D vẫn đang còn ở mức khái niệm, chỉ khi nào struct Vector2D được dùng để khai báo biến, biến x và y cũng như hàm normalize mới được tạo ra. Như mình đã nói ở trên, việc định nghĩa một kiểu dữ liệu mới chỉ là định nghĩa những dữ liệu sẽ tồn tại trong biến struct nếu nó được tạo ra.

Hàm normalize được tạo ra nhưng chỉ được sử dụng khi một biến struct cụ thể gọi đến nó bằng **member selection operator**.

int main()

{

Vector2D vec = { 1, 4 };

vec.normalize();

printVector2D(vec);

**return** 0;

}

Hàm normalize trong struct Vector2D chỉ có thể được gọi thông qua một biến struct cụ thể. Như vậy, khi chúng ta muốn chuẩn hóa một Vector2D, chúng ta không cần truyền biến struct kiểu Vector2D vào hàm normalize(Vector2D) nữa mà chỉ cần gọi hàm normalize được định nghĩa trong chính nó. Đây cũng là một ưu điểm khi sử dụng struct.

Mình lấy thêm một ví dụ nữa về hàm được định nghĩa bên trong struct để các bạn dễ hình dung:

struct Vector2D

{

**float** x;

**float** y;

**void** setPosition(**float** X, **float** Y)

{

x = X;

y = Y;

}

**void** normalize()

{

**float** length = sqrt(x \* x + y \* y);

x = x / length;

y = y / length;

}

};

Mình vừa thêm vào struct Vector2D hàm setPosition(float, float), lúc này mình không cần sử dụng **initializer list** để khởi tạo cho một biến kiểu Vector2D nữa, mà mình sẽ gọi hàm setPosition(float, float).

int main()

{

Vector2D vec;

vec.setPosition(1, 4);

printVector2D(vec);

**return** 0;

}

Điều này cũng làm tăng thêm ý nghĩa cho mã nguồn chương trình, giúp code của các bạn dễ đọc hơn. Chúng ta sẽ còn nói đến việc định nghĩa hàm bên trong struct trong những bài học tiếp theo.

##### Nested structs

Struct là một tập hợp các kiểu dữ liệu dùng để tạo nên một kiểu dữ liệu mới, và một kiểu struct cũng là một kiểu dữ liệu, nên chúng ta có thể sử dụng một kiểu struct khác để làm một trường dữ liệu cho struct cần tạo ra. Ví dụ:

struct **Birthday**

{

\_\_int32 day*;*

\_\_int32 month*;*

\_\_int32 year*;*

}*;*

struct Employee

{

\_\_int32 ID*;*

std::string name*;*

**Birthday birthday;**

\_\_int32 year\_of\_exp*;*

}*;*

Trong trường hợp này, mình thay thế trường dữ liệu age bằng một trường dữ liệu kiểu Birthday đã được định nghĩa ở trên. Chúng ta có thể khởi tạo giá trị cho nested struct trên như sau:

Employee emp = { 1, "Le Tran Dat", {1, 2, 2000}, 5 };

Để truy xuất đến giá trị thực của trường dữ liệu birthday, chúng ta cần sử dụng thêm 1 lần **member selection operator**.

Employee emp = { 1, "Le Tran Dat", {1, 2, 2000}, 5 };

std::cout << emp.ID << std::endl;

std::cout << emp.name << std::endl;

std::cout << emp.birthday.day << "/" << emp.birthday.month << "/" <<emp.birthday.year << std::endl;

std::cout << emp.year\_of\_exp << std::endl;

### Tổng kết

Trong bài học này, chúng ta đã cùng nhau tìm hiểu cách để tạo ra một kiểu dữ liệu mới bằng từ khóa struct cung cấp bởi ngôn ngữ C++, một số thao tác cơ bản với biến struct.

Struct là một khái niệm quan trọng trong ngôn ngữ C/C++, hiểu được structs là một bước quan trọng để tiếp cận hướng phát triển chương trình theo mô hình hướng đối tượng. Struct giúp chúng ta tổ chức chương trình hiệu quả hơn.

# 9.2 Structs and pointer

#### Chào các bạn đang theo dõi khóa học lập trình trực tuyến ngôn ngữ C++.

Chúng ta cùng tiếp tục tìm hiểu về kiểu dữ liệu tự định nghĩa thông qua từ khóa struct mà ngôn ngữ C++ hỗ trợ. Trong bài học này, mình sẽ trình bày về kiểu struct khi sử dụng kết hợp với con trỏ.

Như các bạn đã học trong bài trước, sau khi chúng ta tự định nghĩa một struct, compiler sẽ coi tên gọi của struct đó như là một kiểu dữ liệu. Điều này có nghĩa khi chúng ta sử dụng các kiểu dữ liệu built-in để tạo ra các biến, tham chiếu hoặc con trỏ thì chúng ta cũng có thể sử dụng kiểu struct để tạo ra biến struct, tham chiếu struct và con trỏ kiểu struct (**Pointer to struct**).

### Pointer to struct

Đầu tiên, chúng ta cùng định nghĩa một kiểu dữ liệu theo ý muốn. Dưới đây, mình định nghĩa một kiểu dữ liệu có tên là Letter:

**struct** **Letter**

{

};

Trong struct Letter mình chưa định nghĩa các trường dữ liệu, lúc này Letter là một kiểu dữ liệu rỗng. Nhưng ngôn ngữ C++ vẫn đặt kích thước của kiểu Letter này là 1 bytes.

Mục đích là để đảm bảo địa chỉ của 2 biến được tạo ra sẽ có địa chỉ khác nhau. Tuy nhiên, định nghĩa ra một struct rỗng không có tác dụng gì trong chương trình, chúng ta cùng thêm vào một số trường dữ liệu cho Letter:

**struct** **Letter**

{

char from[50];

char to[50];

};

Một lá thư sẽ có thông tin về người gửi và người nhận, nên mình thêm vào 2 trường dữ liệu kiểu C-style string dùng để lưu thông tin mà người dùng điền vào một lá thư.

Mình vừa định nghĩa xong một kiểu dữ liệu mới để phục vụ cho chương trình của mình. Bây giờ chúng ta cùng tạo ra một đơn vị từ kiểu dữ liệu trên (mình thao tác luôn trong hàm main):

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

{

Letter myLetter;

**return** 0;

}

Với mỗi biến kiểu Letter được tạo ra, chương trình sẽ yêu cầu cấp phát 100 bytes (50 bytes cho trường dữ liệu from và 50 bytes cho trường dữ liệu to), và chắc chắn rồi, biến đó sẽ có một địa chỉ xác định được thông qua toán tử address-of.

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

{

Letter myLetter;

std::cout << "Address of myLetter: " << &myLetter << std::endl;

std::cout << "Address of from field: " << &myLetter.from << std::endl;

**return** 0;

}

Ở đoạn chương trình trên, mình in ra địa chỉ của biến myLetter, đồng thời in ra luôn địa chỉ của trường dữ liệu from của biến myLetter. Kết quả cho thấy 2 địa chỉ được in ra có giá trị hoàn toàn giống nhau. Điều này có nghĩa địa chỉ của trường dữ liệu đầu tiên trong một biến struct cũng là địa chỉ của biến struct đó. Các bạn có thể liên hệ struct với mảng một chiều trong C/C++, khi mảng một chiều mà tập hợp các phần tử có cùng kiểu dữ liệu được bao bọc bởi tên mảng một chiều và địa chỉ của mảng một chiều cũng là địa chỉ của phần tử đầu tiên trong mảng, một biến struct sẽ bao gồm tập hợp các trường dữ liệu mà địa chỉ của biến struct sẽ là địa chỉ của trường dữ liệu được khai báo đầu tiên trong struct.

Và như các bạn cũng đã học về con trỏ (Pointer), kiểu dữ liệu của con trỏ dùng để xác định kiểu dữ liệu của vùng nhớ mà con trỏ có thể trỏ đến. Vậy thì để cho con trỏ trỏ đến một địa chỉ của biến kiểu struct, chúng ta cần có một con trỏ cùng kiểu struct với biến được trỏ đến.

Letter myLetter*;*

Letter \*pLetter = &myLetter*;*

Dù kích thước của kiểu dữ liệu struct có lớn bao nhiêu, biến con trỏ cũng chỉ có kích thước 4 bytes trên hệ điều hành 32 bits và kích thước 8 bytes trên hệ điều hành 64 bits (đủ để trỏ đến toàn bộ địa chỉ trên bộ nhớ ảo).

##### Access struct members

Trong bài học trước, các bạn đã biết cách truy cập đến các trường dữ liệu của các biến struct thông qua **member selection operator** (dấu chấm). Nhưng khi sử dụng Pointer to struct, **member selection operator** được sử dụng dưới cách viết khác. Để phân biệt sự khác nhau khi sử dụng **member selection operator** cho biến struct thông thường và một Pointer to struct, các bạn cùng xem ví dụ bên dưới:

**struct** BankAccount

{

\_\_int64 accountNumber;

\_\_int64 balance;

};

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

{

BankAccount myAccount = { 123456789, 50 }; *// $50*

BankAccount \*pAccount = &myAccount;

std::cout << "My bank account number: " << myAccount.accountNumber << std::endl;

std::cout << "My bank account number: " << pAccount->accountNumber << std::endl;

std::cout << "My balance: " << myAccount.balance << std::endl;

std::cout << "My balance: " << pAccount->balance << std::endl;

**return** 0;

}

Như các bạn thấy, kết quả của việc truy xuất giá trị thông qua tên biến struct và con trỏ kiểu struct là hoàn toàn giống nhau, và chúng đều dùng toán tử member selection. Tuy nhiên, để phân biệt biến con trỏ và biến thông thường, biến con trỏ kiểu struct sẽ truy cập đến các trường dữ liệu trong vùng nhớ bằng toán tử (->). Hai toán tử này cùng tên, chỉ khác nhau về cách biểu diễn.

##### Một số nhầm lần khi sử dụng struct và Pointer to struct

Khi mới tìm hiểu về Pointer to struct, các bạn có thể bị nhầm lẫn giữa cách khởi tạo hoặc gán giá trị cho biến struct thông thường và biến con trỏ struct.

struct **BankAccount**

{

\_\_int64 accountNumber*;*

\_\_int64 **balance;**

}*;*

int main()

{

**BankAccount** myAccount = { 12345, 50 }*;*

**BankAccount** \*pAccount = { 12345, 50 }*; //error*

return 0*;*

}

Đoạn chương trình trên báo lỗi vì biến con trỏ chỉ nhận giá trị là địa chỉ. Tuy nhiên, lỗi này có thể thấy dễ dàng vì Visual studio đưa ra thông báo lỗi ngay. Dưới đây là cách gán giá trị đúng khi mình sử dụng toán tử dereference cho biến con trỏ struct để thay đổi giá trị bên trong vùng nhớ:

struct **BankAccount**

{

\_\_int64 accountNumber*;*

\_\_int64 **balance;**

}*;*

int main()

{

**BankAccount** myAccount = { 0, 0 }*;*

**BankAccount** \*pAccount = &myAccount*;*

\*pAccount = { 12345, 50 }*;*

std::cout << pAccount->accountNumber << " " << pAccount->**balance** << std::endl*;*

return 0*;*

}

Hoặc một cách khác là chúng ta cấp phát vùng nhớ cho biến con trỏ struct, và dereference đến đó để gán giá trị cho nó:

**BankAccount** \*pAccount = new **BankAccount;**

\*pAccount = { 12345, 50 }*;*

Và các bạn lưu ý khi sử dụng biến kiểu con trỏ struct thì chúng ta sử dụng toán tử member selection này (->). Có một số bạn nhầm lẫn giữa biến con trỏ struct và trường dữ liệu kiểu con trỏ. Ví dụ:

struct **BankAccount**

{

char \*name*;*

\_\_int64 accountNumber*;*

\_\_int64 **balance;**

}*;*

Mình thêm vào struct một trường dữ liệu kiểu con trỏ char nhưng việc truy xuất đến trường dữ liệu này không có gì thay đổi khi mình sử dụng biến struct thông thường.

BankAccount myAccount = { "Le Tran Dat", 12345, 50 };

std::cout << myAccount.name << std::endl;

std::cout << myAccount.accountNumber << std::endl;

std::cout << myAccount.balance << std::endl;

Sẽ phức tạp hơn một chút khi các bạn sử dụng các nested struct. Ví dụ:

**struct** BankAccount

{

Date registrationDate;

\_\_int64 accountNumber;

\_\_int64 balance;

};

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

{

BankAccount \*pAccount = **new** BankAccount;

\*pAccount = { {2, 5, 2016}, 12345, 50 };

std::cout << pAccount->registrationDate.year << std::endl;

**return** 0;

}

Như các bạn thấy, từ biến con trỏ pAccount truy xuất vào các trường dữ liệu bên trong thì mình dùng toán tử (->), nhưng trường dữ liệu Date trong struct BankAccount là biến thông thường, nên mình dùng dấu chấm để truy xuất dữ liệu ngày đăng kí.

Trên đây là một số vấn đề thường gặp khi sử dụng con trỏ và kiểu struct. Tuy nhiên, những lỗi này không phải là lỗi nghiêm trọng do compiler sẽ thông báo chính xác vị trí lỗi cho lập trình viên xử lý.

# Phân biệt Syntax-error, Runtime-error and Logical-error?

**Syntax-error**: Lỗi cú pháp được phát hiện ngay khi biên dịch chương trình. Trình biên dịch sẽ thông báo lỗi tại cửa sổ Output.

|  |  |
| --- | --- |
| 1  2  3  4  5 | #include <stdio.h>  **void** main()  {  **printf**("Hello world")  } |

Output

error C2143: syntax error : missing ';' before '}

**Runtime-error**: Chương trình đã được biên dịch thành công, gặp lỗi khi chạy chương trình do đầu vào hoặc đầu ra có giá trị không mong muốn.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | #include <stdio.h>  **void** main()  {  **int** a, b = 5, c = 0;      a = b/c;  } |

Output

Unhandled exception at 0x012313a0 in example.exe: 0xC0000094: Integer division by zero.

**Logical-error**: Chương trình đã được biên dịch và chạy không gặp lỗi Runtime-error. Nhưng kết quả đầu ra không đúng theo yêu cầu do logic xử lý bài toán bị sai.

# Complier and Linker C/C++ là gì?

Complier và Linker là 2 component trong process viết 1 chương trình C/C++.

Bước 1: Soạn thảo mã nguồn .c và .h (header)

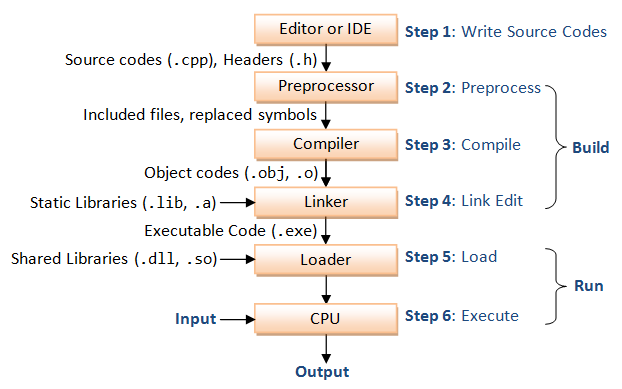
Bước 2: Tiền xử lí (pre-process) source code dựa vào macro và preprocessor directives. Tiền xử lí sẽ thay thế các giá trị được định nghĩa bởi #define, #include,…. trước khi biên dịch

Bước 3: Complier biên dịch source code đã được tiền xử lí thành các object code (.obj)

Bước 4: Linker link object code với (các object code khác, static lib, dynamic lib) để tạo ra file .EXE

Bước 5: Load chương trình .EXE vào bộ nhớ

Bước 6: Run .EXE với giá trị nhập vào và output đúng như mong muốn.

Complier and Linker C/C++

# Vòng lặp vô hạn trong ngôn ngữ C/C++ là gì?

## Vòng lặp vô hạn là gì?

Vòng lặp vô hạn là chuỗi các câu lệnh được thực hiện lặp đi lặp lại không có điểm dừng. Nguyên nhân có thể là:

* Vòng lặp chưa có điều kết thúc vòng lặp
* Vòng lặp có điều điều kiện để kết thúc vòng lặp nhưng điều kiện này không bao giờ được đáp ứng.
* Hoặc có thể một logic nào đó dẫn đến lặp vô hạn (ví dụ: lệnh goto có thể tạo ra vòng lặp)
* Chương trình bị dính vòng lặp vô hạn, chương trình có thể bị gây treo PC hoặc bị crash (nếu trong vòng lặp khai bao biến hoặc cấp phát vùng nhớ mà không giải phóng vùng nhớ).
* Do vậy, cần tránh để xảy ra vòng lặp vô hạn trong chương trình.
* Trong một số trường hợp (requirement), lập trình  viên không xác định chính xác được số lần lặp để xử lí 1 logic nào đó. Lúc này, lập trình viên phải sử dụng khai báo vòng lặp vô hạn [**for()**](http://vncoding.net/2015/11/04/vong-lap-for-2/), [**while(), do while()**](http://vncoding.net/2015/10/25/while-do-while/).

## Các dạng của vòng lặp vô hạn trong ngôn ngữ C

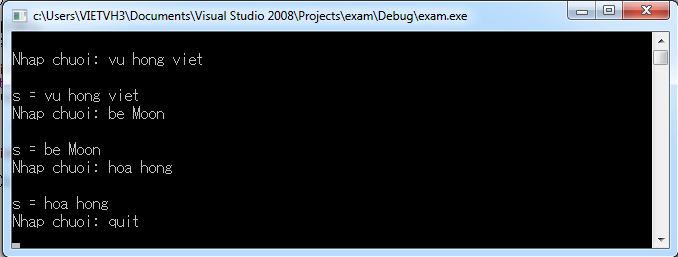
### Vòng lặp vô hạn for

Khai báo:

|  |  |
| --- | --- |
| 1  2  3  4 | **for** ( ; ; )  {      // place code here  } |

Ví dụ: Nhập và in ra chuỗi đã nhập. Việc nhập kết thúc khi nhập “quit”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include <stdio.h>  #include <string.h>  #include <conio.h>    **void** main()  {  **char** s[100];  **for** ( ; ; )      {  **printf**("\nNhap chuoi: ");  **gets**(s);  **if**(**strcmp**(s, "quit") == 0)          {  **break**;          }  **printf**("\ns = %s", s);      }      getch();  } |

Vòng lặp vô hạn for

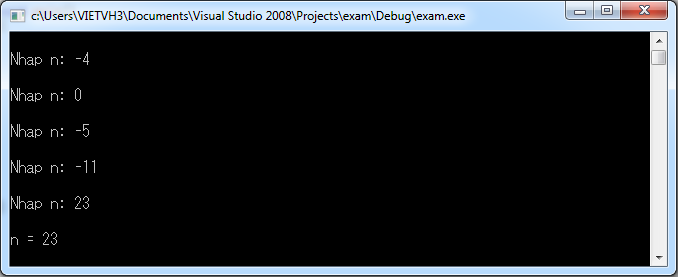
### Vòng lặp vô hạn while

Khai báo:

|  |  |
| --- | --- |
| 1  2  3  4 | **while**(1)  {      //pleace code here  } |

Ví dụ: Nhập số nguyên lớn hơn 0. Nếu nhập số <= 0, yêu cầu nhập lại.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include <stdio.h>  #include <string.h>  #include <conio.h>    **void** main()  {  **int** n;  **while** (1)      {  **printf**("\nNhap n: ");  **scanf**("%d", &n);  **if**(n > 0)          {  **break**;          }      }  **printf**("\nn = %d", n);      getch();  } |

Vòng lặp vô hạn while

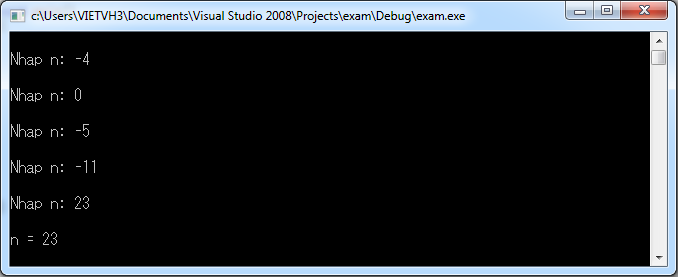
### Vòng lặp vô hạn do while

Khai báo:

|  |  |
| --- | --- |
| 1  2  3  4 | **do**  {      //place code here  } **while** (1); |

Ví dụ: Nhập số nguyên lớn hơn 0. Nếu nhập số <= 0, yêu cầu nhập lại.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #include <stdio.h>  #include <string.h>  #include <conio.h>    **void** main()  {  **int** n;  **do**      {  **printf**("\nNhap n: ");  **scanf**("%d", &n);  **if**(n > 0)          {  **break**;          }      }  **while**(1);  **printf**("\nn = %d", n);      getch();  } |

Vòng lặp vô hạn do while

# Phân biệt biến Local, Global, Extern, Static và Const

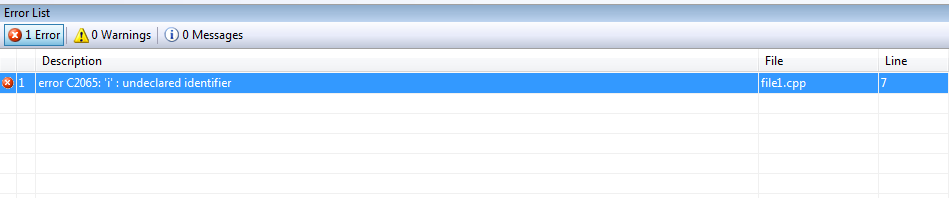
### Biến local

– Biến local là xuất hiện trong phạm vi cụ thể.  
– Biến local chỉ tồn tại trong hàm mà biến được khai báo  
– Đôi khi, biến local được gọi là biến tự động (auto) bởi vì các biến được tự động sinh ra khi hàm được thực hiện và sẽ tự động biến mất khi kết thúc hàm.  
– Từ khóa **auto** được sử dụng để ám chỉ biến cục bộ.

Ví dụ:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #include <conio.h>  #include <stdio.h>  **void** Display();  **int** main()  {      Display();  **printf**("\n i = %d", i); // error      getch();  }    **void** Display()  {  **int** i;  **for**(i = 0; i < 10; i++)  **printf**("\n %d", i);  } |

Kết quả:

Biến local

Chương trình này khi build sẽ bị lỗi do, biến i khai báo trong hàm Display(), nên phạm vị hoạt động của biến i chỉ trong hàm Display(). Hàm main() không sử dụng được biến i => gây ra lỗi.

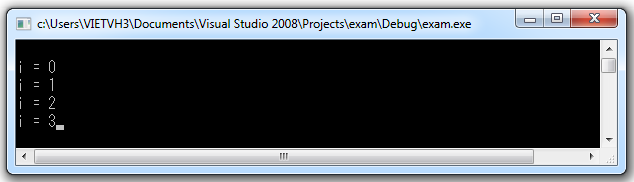
### Biến global và extern

– Biến global là biến được khai báo bên ngoài tất cả các hàm và có giá trị với tất cả các hàm trong chương trình. Tức là các hàm trong chương trình có thể sử dụng biến global để tính toán.  
– Biến global tồn tại đến khi nào chương trình kết thúc.  
– Có thể định nghĩa 1 biến global trong 1 file (.c/.cpp/.h) và truy cập biến này từ 1 file (.c/.cpp/.h) khác. Để làm điều này, biến phải được khai báo ở cả 2 file và từ khóa **extern**được thêm trong lần khai báo thứ 2.

Ví dụ: Biến toàn cục global

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | #include <conio.h>  #include <stdio.h>    **int** i = 0; // Bien Global    **void** Display();    **int** main()  {      Display();  **printf**("\ni = %d", i);      getch();  }  **void** Display()  {  **for**(i = 0; i < 3; i++)  **printf**("\ni = %d", i);  } |

Kết quả:

Biến toàn cục global

Ví dụ: Biến toàn cục extern

File header1.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #ifndef \_HEADER\_H\_  #define \_HEADER\_H\_    **extern** **int** X;  **void** display\_x();    #endif |

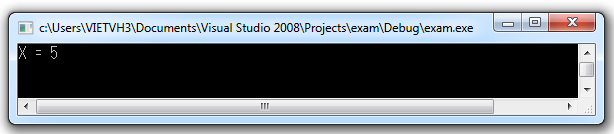
File file1.cpp

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | #include "header1.h"  #include "conio.h"    **int** X; // khai bao bien global    **void** main()  {      X = 5;      display\_x();      getch();  } |

File file2.cpp

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #include "stdio.h"  #include "header1.h"    **void** display\_x()  {  **printf**("X = %d", X);  } |

Kết quả:

Biến Extern

### Biến static

– Biến **static**có thể là global hoặc local. Cả hai đều được khai báo với từ khóa **static**đi kèm.  
– Biến **local static** là biến có thể duy trì giá trị từ lần gọi hàm thứ nhất đến các lần gọi hàm tiếp theo. Biến local static tồn tại đến khi chương trình kết thúc.  
– Khi tạo 1 biến local static trong hàm, chúng ta nên khởi tạo giá trị cho chúng. Nếu không giá trị biến được gán mặc định bằng 0.  
– Biến **global static** là biến global mà chỉ có thể truy cập từ file (.c/.cpp) mà biến đó được định nghĩa.

### Biến const

– Trong ngôn ngữ C, tiền xử lý #define được sử dụng để tạo biến với giá trị là hằng số.  
– Trong ngôn ngữ C++, xuất hiện một số vấn đề: khi sử dụng #define, tiền xử lí sẽ nhảy thẳng vào source code và thay thế biến bằng giá trị đã định nghĩa. Vì biến #define chỉ tồn tại bên trong file mà nó được định nghĩa, có thể xảy ra trường hợp định nghĩa tên biến giống nhưng khác về giá trị.  
– Định nghĩa biến hằng trong C++, chủng ta nên sử dụng từ khóa **const** đi kèm.  
– Khi sử dụng từ khóa **const**, phải khởi tạo giá trị ban đầu cho biến.

# Hàm sprintf()

int sprintf(char \*buffer, const char \*format [,argument] ...);

buff: vùng nhớ lưu trữ chuỗi output.

argument: biến tùy chọn

%d hoặc %i : in ra số nguyên

%u : in ra số nguyên không dấu

%o : in ra số hệ 8 (octal)

%x hoặc %X : in ra số hê 16 (hexa)

%f : in ra số dấu phẩy động

%c : in ra kí tự

%s : in ra string

%e hoặc %E: in dạng M\*e^x

Một số kí tự đặc biệt:

‘/n’ : kí tự xuống dòng

‘/t’ : kí tự cách ra 1 tab (4 space)

‘/r’ : kí tự trở về đầu dòng in

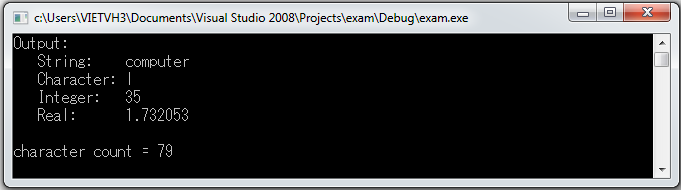
‘/b’: kí tự lùi con trỏ màn hình về sau 1 kí tự

* Hàm sprintf() được dùng để tạo ra chuỗi từ các kiểu dữ liệu nguyên thủy khác nhau (char\*, int, float,…)
* Hàm sprintf() trả về số lượng kí tự được ghi ra chuỗi buff.

Ví dụ:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | #include <stdio.h>  #include <conio.h>      **void** main()  {  **char**  buffer[200], s[] = "computer", c = 'l';  **int**   i = 35, j;  **float** fp = 1.7320534f;        // Format and print various data:      j  = **sprintf**( buffer,     "   String:    %s\n", s );      j += **sprintf**( buffer + j, "   Character: %c\n", c );      j += **sprintf**( buffer + j, "   Integer:   %d\n", i );      j += **sprintf**( buffer + j, "   Real:      %f\n", fp );      // Note: sprintf is deprecated; consider using sprintf\_s instead    **printf**( "Output:\n%s\ncharacter count = %d\n", buffer, j );        getch();  } |

Kết quả:

Hàm sprint

# Sự khác nhau giữa hàm memcpy và strcpy?

Memcpy va strcpy đều sử dụng để copy dữ liệu từ vùng nhớ này sang vùng nhớ khác. Tuy nhiên, chúng có một số điểm khác nhau:

[**strcpy()**](http://vncoding.net/2015/11/15/ham-strcpy-copy-noi-dung-cua-chuoi-nguon-sang-chuoi-dich/)

* Copy chuỗi kí tự nguồn sang chuỗi đích. Hàm strcpy copy kí tự đến khi gặp kí tự NULL thì dừng lại.

[**memcpy()**](http://vncoding.net/2015/11/15/ham-memcpy-se-copy-n-byte-tu-vung-nho-nguon-vao-vung-nho-dich/)

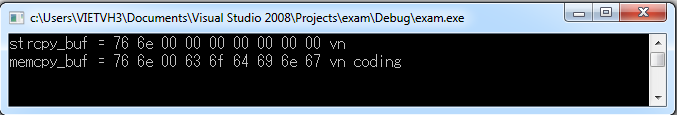
* Copy chính xác số byte dữ liệu từ vùng nhớ nguồn sang vùng nhớ đích.

Chú ý: cả 2 hàm không xử lí được trường hợp overlap buffer.

Các bạn xem thêm ví dụ dưới đây để hiểu rõ hơn về sự khác nhau giữa 2 hàm này.

|  |  |
| --- | --- |
| 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  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | #include <stdio.h>  #include <string.h>  #include <conio.h>    **void** dump(**char**\* s, **int** sz);    **void** main()  {  **char** s[]={'v', 'n', '\0', 'c', 'o', 'd', 'i', 'n', 'g'};  **char** strcpy\_buf[5];  **char** memcpy\_buf[5];    **memset**(strcpy\_buf, NULL, 9);  **memset**(memcpy\_buf, NULL, 9);    **strcpy**(strcpy\_buf, s);  **memcpy**(memcpy\_buf, s, 9);    **printf**("strcpy\_buf = ");      dump(strcpy\_buf, 9);    **printf**("memcpy\_buf = ");      dump(memcpy\_buf, 9);        getch();  }    **void** dump(**char**\* s, **int** sz)  {  **char** \*p = s;  **int** i;  **for**(i = 0; i < sz; i++)      {  **printf**("%.2x ", \*(p+i)); // hexa ASCII      }  **for**(i = 0; i < sz; i++)      {  **printf**("%c", (p+i)? \*(p+i) : ' '); // characters      }  **printf**("\n");  } |

Kết quả:

Sự khác nhau giữa memcpy và strcpy

# Sự khác nhau giữa hàm memcpy và memmove?

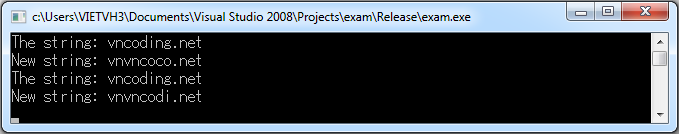
Cả 2 hàm [**memcpy**](http://vncoding.net/2015/11/15/ham-memcpy-se-copy-n-byte-tu-vung-nho-nguon-vao-vung-nho-dich/)và [**memmove**](http://vncoding.net/2015/11/15/ham-memmove-copy-n-byte-tu-src-vao-vung-nho-moi-dest/)được sử dụng để copy N byte dữ liệu từ vùng nhớ này sang  vùng nhớ khác. Tuy nhiên, trong trường  hợp vùng nhớ nguồn overlap với vùng nhớ đích:

* Hàm memmove đảm bảo việc copy dữ liệu và output là chính xác .
* Hàm memcpy KHÔNG đảm bảo việc copy dữ liệu và output là chính xác.

Ví dụ sau sẽ giúp các bạn hiểu thêm về sự khác nhau của 2 hàm trong trường hợp vùng nhớ bị overlap.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include <stdio.h>  #include <string.h>  #include <conio.h>    **char** str1[20] = "vncoding.net";    **void** main( **void** )  {  **printf**("The string: %s\n", str1);  **memcpy**(str1 + 2, str1, 6);  **printf**("New string: %s\n", str1);        strcpy\_s(str1, **sizeof**(str1), "vncoding.net");   // reset string    **printf**("The string: %s\n", str1);  **memmove**(str1 + 2, str1, 6);  **printf**("New string: %s\n", str1);      getch();  } |

Kết quả:

Sự khác nhau giữa hàm memcpy và memmove

Theo như kết quả, hàm memmove đảm bảo kết quả đúng như mong muốn (“vncodi” ghi đè lên “coding”).

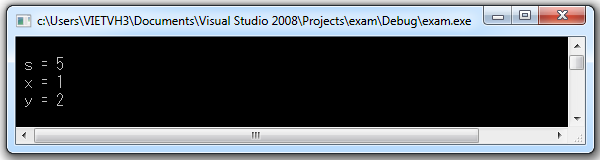
# Truyền tham trị và tham biến cho hàm

Khi một hàm mà được định nghĩa và khai báo trong chương trình thì hàm main(), các hàm khác và thậm chí hàm đó có thể gọi đến chính nó (đệ quy). Trong C/C++, có các cách truyền đối số (arguments) cho hàm như sau:

## Truyền tham trị (dùng trong cả C và C++)

Khi truyền đối số kiểu tham trị, chương trình biên dịch sẽ copy giá trị của đối số để gán cho tham số của hàm (không tác động trực tiếp đến biến số truyền vào).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include <stdio.h>  #include <conio.h>    **int** sum(**int** a, **int** b)  {      a++;      b++;  **return** (a+b);  }  **void** main()  {  **int** x =1, y = 2;  **int** s = 0;      s = sum(x, y);  **printf**("\n s = %d", s);  **printf**("\n x = %d", x);  **printf**("\n y = %d", y);      getch();  } |

Truyền tham trị cho hàm

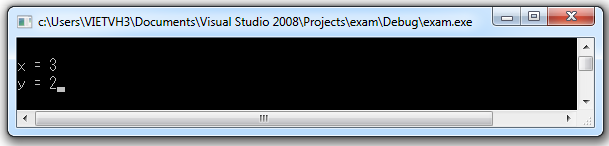
Trong ví dụ này, ta định nghĩa và khai báo hàm sum() có kiểu trả về là int, 2 tham số a,b kiểu int. Khi gặp câu lệnh gọi hàm sum() trong hàm main(). Chương trình sẽ tạo ra 2 biến Local a,b trong hàm sum(). Giá trị của đối số x,y truyền vào sẽ được copy và gán tương ứng cho 2 tham số a,b  
(a = 1,b = 2). Các câu lệnh tiếp theo trong hàm sum() chỉ thao tác trên 2 biến Local a,b. Do vậy kết quả là giá trị biến x = 1 , y = 2.

## Truyền tham biến

Phương pháp truyền tham biến là cách truyền địa chỉ của đối số cho các tham số tương ứng của hàm được gọi. Với cách truyền tham biến, giá trị của đối số truyền vào có thể bị thay đổi bởi việc gọi hàm.  
Truyền tham biến chia ra thành 2 loại : truyền con trỏ (dùng trong C và C++) , truyền tham chiếu (chỉ dùng trong C++)

### Truyền con trỏ (dùng trong C và C++)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | #include <stdio.h>  #include <conio.h>    **void** swap(**int** \*a, **int** \*b)  {  **int** temp;      temp = \*a;      \*a = \*b;      \*b = temp;  }  **void** main()  {  **int** x = 2, y = 3;      swap(&x, &y);  **printf**("\nx = %d", x);  **printf**("\ny = %d", y);      getch();  } |

Truyền con trỏ cho hàm

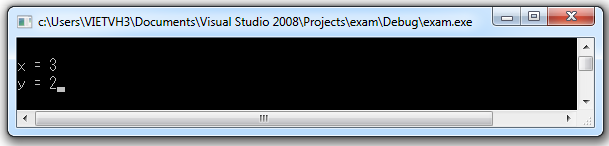
Hàm swap() làm nhiệm vụ đổi chỗ 2 biến nguyên x,y truyền vào. Đối số truyền vào ở đây là địa chỉ 2 biến x,y (&x , &y). Trong hàm swap(), biến con trỏ a,b sẽ trỏ tới địa chỉ của biến x,y (a = &x, b = &y). Và \*a và \*b chính là giá trị của 2 biến x,y. Các câu lệnh tiếp theo:

|  |  |
| --- | --- |
| 1  2  3 | temp = \*a;  \*a = \*b;  \*b = temp; |

Ta thấy, hàm swap() đã thay đổi giá trị của đối số truyền vào.

### Truyền tham chiếu (chỉ dùng trong C++)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | #include <stdio.h>  #include <conio.h>    **void** swap(**int** &a, **int** &b)  {  **int** temp;      temp = a;      a = b;      b = temp;  }  **void** main()  {  **int** x = 2, y = 3;      swap(x, y);  **printf**("\nx = %d", x);  **printf**("\ny = %d", y);      getch();  } |

Truyền tham chiếu cho hàm

Hàm swap() cũng làm nhiệm vụ hoán đổi vị trí giữa 2 biến nguyên truyền vào. Trong C++ cho phép sử dụng biến tham chiếu.  
+ Biến tham chiếu không được cấp phát bộ nhớ, không có địa chỉ riêng.  
+ Nó dùng làm bí danh cho một biến (kiểu giá trị) nào đó và nó sử dụng vùng nhớ của biến này.  
Trong hàm swap(), chương trình sẽ thực hiện lệnh gán sau:  
int &a = x;  
int &b = y;  
a,b ở đây là bí danh của biến x,y. Tức là biến a,b sẽ dùng chung vùng nhớ với biến số x,y.  
Các cậu lệnh được thao tác trên biến a,b hay cũng chính là thao tác trên biến x,y. Do vậy, hàm swap() sẽ thay đổi giá trị của đối số truyền vào.

# Detect leak memory trên Visual Studio

Leak memory được định nghĩa như việc lập trình viên quên không giải phóng ([**free**](http://vncoding.net/2015/11/15/ham-free-giai-phong-vung-nho-da-cap-phat/)hoặc delete) vùng nhớ đã cấp phát sau khi sử dụng xong. Một lượng nhỏ memory leak không được phát hiện ngay ban đầu. Lượng leak memory sẽ tăng dần theo thời gian. Điều này làm cho ứng dụng giảm performance, hoặc thậm chí gây chết (crash) chương trình khi sử dụng hết memory. Hoặc tệ hơn nữa, việc leak memory của 1 ứng dụng sử dụng hết vùng nhớ của các ứng dụng khác, làm cho các ứng dụng khác có thể bị crash.

Có rất nhiều cách để tìm ra leak memory trong 1 project C/C++ như review sourcecode, tools,…  
Ở đây tôi sẽ giới thiệu cho các bạn 1 công cụ khá đắc lực và tin tưởng trong việc phát hiện leak memory trong project C/C++.

Trong Visual Studio 2005, 2008, 2010,.. đã tích hợp Visual Studio debugger and C Run-Time (CRT) libraries cung cấp cho bạn cách phát hiện leak memory.

### Enable chức năng detect leak memory

#### Enable debug vùng nhớ heap,

Bạn cần phải #include các thư viện sau:

|  |  |
| --- | --- |
| 1  2  3 | #define \_CRTDBG\_MAP\_ALLOC  #include <stdlib.h>  #include <crtdbg.h> |

Chú ý:  
Các câu lệnh phải được giữ nguyên theo thứ tự như trên. Nếu không, việc detect leak memory sẽ không chính xác.  
– Câu lệnh “#include <crtdbg.h>”, hàm [**malloc**](http://vncoding.net/2015/11/15/ham-malloc-cap-phat-bo-nho-dong/)và [**free**](http://vncoding.net/2015/11/15/ham-free-giai-phong-vung-nho-da-cap-phat/)sẽ được map (ánh xạ) tới hàm \_malloc\_dbg và \_free\_dbg trong chế độ build **Debug**.

#### Dump thông tin của memory leak

Sau khi khai báo các câu lệnh ở mục trên, bạn có thể dump thông tin của memory leak bằng cách khai báo cậu lệnh sau vào vị trí mà chương trình chưa kết thúc.

|  |  |
| --- | --- |
| 1 | \_CrtDumpMemoryLeaks(); |

Khi chạy chương trình dưới debugger (Debug mode), hàm \_CrtDumpMemoryLeaks() sẽ hiển thị thông tin memory leak trên cửa sổ **Ouput**.  
Chú ý:  
Phải build và chạy chương trình với Debug mode

#### Format của thông tin memory leak như sau:

Detected memory leaks!

Dumping objects ->

C:\PROGRAM FILES\VISUAL STUDIO\MyProjects\leaktest\leaktest.cpp(20) : {18}

normal block at 0x00780E80, 64 bytes long.

Data: <                > CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD

Object dump complete.

– “C:\PROGRAM FILES\VISUAL STUDIO\MyProjects\leaktest\leaktest.cpp” là đường dẫn của file có memory leak.  
– (20) là dòng thứ 20 trong source code có memory leak.  
– {18} là vị trí của memory.  
– Nếu bạn không sử dụng câu lệnh #define \_CRTDBG\_MAP\_ALLOC, format của thông tin memory leak như sau:

Detected memory leaks!

Dumping objects ->

{18} normal block at 0x00780E80, 64 bytes long.

Data: <                > CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD

Object dump complete

#### Goto tới dòng code cấp phát bộ nhớ

– Double click chuột vào dòng “C:\PROGRAM FILES\VISUAL STUDIO\MyProjects\leaktest\leaktest.cpp(20) : {18}” trên Windows output  
– Hoặc select dòng “C:\PROGRAM FILES\VISUAL STUDIO\MyProjects\leaktest\leaktest.cpp(20) : {18}” trên Windows output và ấn phím F4

#### Ví dụ minh họa: Detect leak memory

Đoạn code sau minh họa việc sử dụng malloc() cấp phát bộ nhớ và không giải phóng dẫn đến leak memory.

|  |  |
| --- | --- |
| 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"    #define \_CRTDBG\_MAP\_ALLOC  #include <stdlib.h>  #include <crtdbg.h>    **void** main()  {  **int** \*pArr;  **int** i;      pArr = (**int**\*)**malloc**(5 \* **sizeof**(**int**));  **if** (pArr)      {  **for** (i = 0; i < 5; i++)          {              pArr[i] = i;          }      }      \_CrtDumpMemoryLeaks();      getch();  } |

Kết quả:

Detect leak memory trên Visual Studio 2008

# Ưu nhược điểm khi sử dụng mảng trong ngôn ngữ C/C++?

Ưu điểm: Mảng có 1 số ưu điểm như sau.

* Dễ hiểu và dễ sử dụng: chỉ cần khai báo <kiểu dữ liệu> tên mảng[kích thước].
* Truy cập đến các phần tử trong mảng nhanh: chúng ta có thể truy cập tới bất kì phần tử nào trong mảng bằng cách chỉ định chỉ số cho phần tử đó. Ví dụ: mảng A[100] gồm 100 phân tử ( từ A[0] đến A[99]), để truy xuất tới phần tử thứ i ta chỉ cần gọi giá trị A[i]. Thời gian truy cập phần tử A[0] và thời gian truy cập phần tử A[1000] là như nhau.

Hạn chế: Bên cạnh những ưu điểm, mảng còn tồn tại một số hạn chế sau.

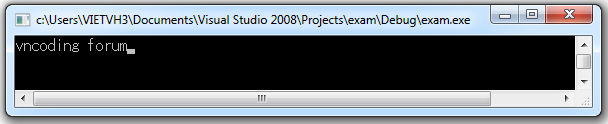
* Kích thước của mảng phải là cố định: Trong cấp phát mảng tĩnh, mảng cần được khai báo với kích thước xác định trước khi chạy chương trình. (vùng nhớ cho mảng được cấp phát khi biên dịch). Trong cấp phát động, vùng nhớ được cấp phát khi chạy chương trình. Như các bạn đã biết, vùng nhớ giành cho mỗi chương trình thường không dự đoán được trước. Nếu khai báo mảng với kích thước lớn, không sử dụng hết sẽ gây lãng phí bộ nhớ, ngược lại nếu kích thước vùng nhớ không đủ dùng, chúng ta không thể mở rộng vùng nhớ thêm được, dẫn đến buffer overrun ( tràn vùng nhớ).
* Các byte vùng nhớ cấp phát mảng được sắp xếp liên tục: trong trường hợp vùng nhớ cho chương trình đang bị phân mảnh, chương trình sẽ báo lỗi khi chúng ta khai báo hoặc cấp phát cho mảng với kích thước lớn vì lý do: không đủ vùng nhớ liên tục cho mảng ( mặc dù tổng dung lượng vùng nhớ phân mảnh là đủ).
* Việc chèn và xóa phần tử của mảng mất nhiều thời gian: vì vùng nhớ cấp phát cho mảng được sắp xếp liên tục nên việc chèn một phần tử mới vào hoặc xóa phần tử trong mảng trở lên khó khăn. Ví dụ: cho mảng A[100], chúng ta muốn chèn thêm phần tử mới vào vị trí i, tất cả các phần tử thứ i trở đi phải dịch sang vị trí kế tiếp để chèn giá trị vào vị trí thứ i. Việc xóa phần tử trong mảng cũng tương tự như vậy, dịch tất cả các phần tử từ vị trí thứ i+1 sang vị trí liền trước nó.

# Sự khác nhau giữa cấp phát tĩnh và cấp phát động?

**Cấp phát bộ nhớ tĩnh**

Khi khai báo các biến (int, float,..), mảng, chuỗi kí tự,(int a[20], char s[30]).. Trình biên dịch cấp phát vùng nhớ (static memory) cho các biến này. Vì vùng nhớ giành cho các (biến, mảng, chuỗi,..) là static memory nên việc gán con trỏ tới vùng nhớ tĩnh được gọi là cấp phát bộ nhớ tĩnh. Vùng nhớ được cấp phát trong lúc biên dịch.

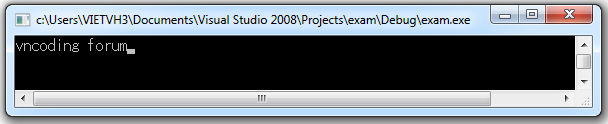
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #include <stdio.h>  #include <conio.h>    **void** main()  {  **char** s[] = "vncoding forum";  **char** \*p = &s[0];  **printf**("%s", p);      getch();  } |

Cấp phát bộ nhớ tĩnh

**Cấp phát bộ nhớ động**

Sử dụng hàm malloc(), calloc() để cấp phát bộ nhớ động. Giá trị trả về của malloc(), calloc() là con trỏ trỏ tới vùng nhớ được cấp phát. Vùng nhớ được cấp phát trong lúc chạy chương trình (runtime)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <conio.h>    **void** main()  {  **char** \*s = (**char**\*)**malloc**(30);  **if** (s)      {  **strcpy**(s, "vncoding forum");  **printf**("%s", s);  **free**(s);      }      getch();  } |

Cấp phát bộ nhớ động

# Hằng con trỏ và con trỏ hằng

**1. Hằng con trỏ – Constant pointer**

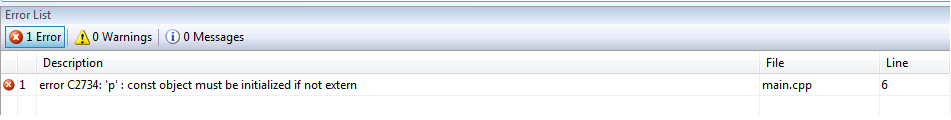
*Khai báo:*

<Kiểu dữ liệu> \* const <Tên con trỏ> = <Địa chỉ khởi tạo> ;

*Đặc điểm:*  
– Cần gán ngay giá trị địa chỉ khởi tạo cho hằng con trỏ tại câu lệnh khai báo ban đầu.  
– Không thể thay đổi địa chỉ đã được khởi gán cho hằng con trỏ ( sẽ gây ra lỗi).  
– Có thể thay đổi giá trị tại địa chỉ đã khởi gián ban đầu.

*Ví dụ 1:*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #include "stdio.h"  #include "conio.h"    **void** main()  {  **int**\* **const** p; // error  } |

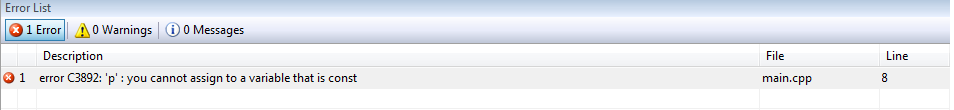
Hằng con trỏ

Chương trình dịch sẽ thông báo lỗi vì hằng con trỏ p không khởi tạo giá trị khi khai báo.

*Ví dụ 2:*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #include "stdio.h"  #include "conio.h"    **void** main()  {  **int** m = 2;  **int**\* **const** p = &m;      p++; // error  } |

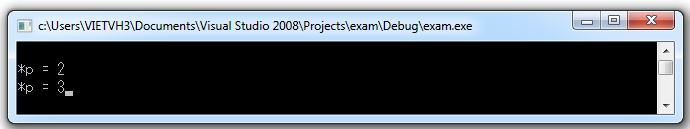
Chương trình dịch sẽ thông báo lỗi vì hằng con trỏ không thể thay đổi địa chỉ trỏ tới.

Hằng con trỏ

*Ví dụ 3:*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | #include "stdio.h"  #include "conio.h"    **void** main()  {  **int** m = 2;  **int**\* **const** p = &m;  **printf**("\n\*p = %d", \*p);      m++;  **printf**("\n\*p = %d", \*p);      getch();  } |

Câu lệnh m++ chỉ làm thay đổi giá trị tại địa chỉ &m.

Hằng con trỏ

**2. Con trỏ hằng – Pointer to constant**

*Khai báo:*

const <Kiểu dữ liệu> \* <Tên con trỏ>;

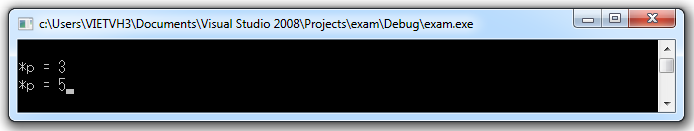
*Đặc điểm:*  
– Không được phép dùng trực tiếp con trỏ hằng để thay đổi giá trị tại vùng nhớ mà con trỏ hằng đang trỏ đến.  
– Con trỏ hằng có thể thể thay đổi địa chỉ trỏ tới (hay nói cách khác: nó có thể trỏ đến các ô nhớ khác nhau).  
*Ví dụ 4:*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | #include "stdio.h"  #include "conio.h"    **void** main()  {  **int** m = 3;  **const** **int**\* p;      p = &m;      m++;      (\*p)++; // error  } |

Câu lệnh m++ hoàn toàn hợp lệ, còn câu lệnh (\*p)++ gây ra lỗi thay đổi giá trị tại vùng nhớ.

*Ví dụ 5:*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #include "stdio.h"  #include "conio.h"    **void** main()  {  **int** m = 3, n = 5;  **const** **int**\* p;      p = &m;  **printf**("\n\*p = %d", \*p);      p = &n;  **printf**("\n\*p = %d", \*p);      getch();  } |

Con trỏ hằng

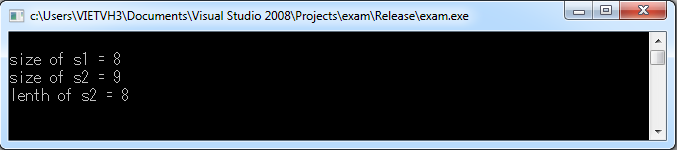
# Sự khác nhau giữa mảng và chuỗi?

* Mảng : được khai báo <kiểu dữ liệu> tên mảng[<kích thước>]. Trong đó, kiểu dữ liệu có thể là kiểu dữ liệu nguyên thủy: int, float, double, long, char,… hoặc struct, class,…
* Chuỗi là kiểu dữ liệu đặc biệt của mảng. Chuỗi là mảng kí tự (kiểu char) + kí tự NULL (‘\0’) ở phần tử cuối của chuỗi. Kí tự NULL sẽ được trình biên dịch tự động thêm vào trong quá trình biên dịch.

Ví dụ: Khai báo mảng kí tự và chuỗi kí tự.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | #include <stdio.h>  #include <string.h>  #include <conio.h>      **void** main( **void** )  {  **char** s1[] = {'v', 'n', 'c', 'o', 'd', 'i', 'n', 'g'};  **char** s2[] = "vncoding";  **int** sz1 = **sizeof**(s1);  **int** sz2 = **sizeof**(s2);  **int** len = **strlen**(s2);  **printf**("\nsize of s1 = %d", sz1);  **printf**("\nsize of s2 = %d", sz2);  **printf**("\nlenth of s2 = %d", len);        getch();  } |

Kết quả:

Sự khác nhau giữa mảng và chuỗi

* s1 là mảng kí tự gồm các kí tự (v, n, c, o, d, i, n, g)
* s2 là chuỗi kí tự gồm các kí tự (v, n, c, o, d, i, n, g) và kí tự NULL ẩn (chúng ta ko nhìn thấy) sẽ được trình biên dịch thêm vào trong quá trình biên dịch.
* Toán tử sizeof lấy kích thước (theo byte) của s1 và s2. s1 chứa 8 hằng kí tự tương ứng với kích thước 8 (byte). s2 chứa 8 hằng kí tự + 1 kí tự NULL tương ứng với kích thước 9 (byte).
* Hàm strlen() lấy độ dài của chuỗi kí tự (ko tính kí tự NULL). Do vậy, độ dài là 8 (kí tự). Ngôn ngữ C có support nhiều built-in function thao tác và xử lí chuỗi kí tự:

|  |
| --- |
| * [**Hàm strcpy copy chuỗi nguồn sang chuỗi đích**](http://vncoding.net/2015/11/15/ham-strcpy-copy-noi-dung-cua-chuoi-nguon-sang-chuoi-dich/) * [**Hàm strncpy copy n byte từ chuỗi nguồn sang chuỗi đích**](http://vncoding.net/2015/11/15/ham-strncpy-copy-mot-so-byte-tu-chuoi-nguon-sang-chuoi-dich/) * [**Hàm strcat nối 2 chuỗi**](http://vncoding.net/2015/11/15/ham-strcat-noi-2-chuoi/) * [**Hàm strncat nối n byte chuỗi nguồn vào chuỗi đích**](http://vncoding.net/2015/11/15/ham-strncat-noi-n-byte-chuoi-nguon-vao-chuoi-dich/) * [**Hàm strcmp so sánh 2 chuỗi**](http://vncoding.net/2015/11/15/ham-strcmp-so-sanh-2-chuoi/) * [**Hàm strncmp so sánh 2 chuỗi con**](http://vncoding.net/2015/11/15/ham-strncmp-so-sanh-2-chuoi-con-cua-2-chuoi/) * [**Hàm strchr tìm kiếm kí tự trong chuỗi**](http://vncoding.net/2015/11/15/ham-strchr-tim-kiem-ki-tu-trong-chuoi/) * [**Hàm strrchr tìm kiếm kí tự c trong chuỗi**](http://vncoding.net/2015/11/15/ham-strrchr-tim-kiem-ki-tu-c-trong-chuoi/) * [**Hàm strspn tìm hiếm bộ kí tự trong chuỗi**](http://vncoding.net/2015/11/15/ham-strspn-tim-kiem-bo-ki-tu-trong-chuoi/) * [**Hàm strpbrk trả về con trỏ tới vị trí xuất hiện đầu tiên của 1 trong các kí tự**](http://vncoding.net/2015/11/15/ham-strpbrk-tra-ve-con-tro-toi-vi-tri-xuat-hien-dau-tien-cua-1-trong-cac-ki-tu/) * [**Hàm strstr tìm kiếm chuỗi con**](http://vncoding.net/2015/11/15/ham-strstr-tim-kiem-chuoi-con/) * [**Hàm strlen tính độ dài của chuỗi**](http://vncoding.net/2015/11/15/ham-strlen-tinh-do-dai-chuoi-ki-tu/) * [**Hàm strtok tìm chuỗi con dựa vào bộ kí tự**](http://vncoding.net/2015/11/15/ham-strtok-tim-chuoi-con-dua-vao-bo-ki-tu/) |

# Con trỏ void

Giả sử khi ta cần sử dụng 3 biến con trỏ kiểu: int, float, char, thì ta cần phải khai báo 3 con trỏ kiểu int, float, char. Nhưng trong ngôn ngữ lập trình C, có 1 dạng con trỏ đặc biệt (gọi là void pointer).  
*Khai báo:*

void \*p;

*Đặc điểm của con trỏ void:*  
– Nó có thể lưu trữ địa chỉ của mọi kiểu biến dữ liệu

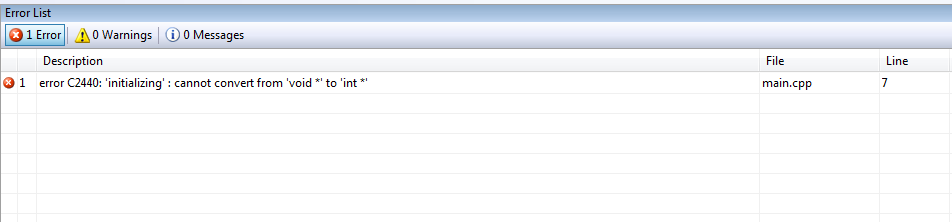
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **int** a;  **float** b;  **char** c;  **void** \*p;  p = &a; // address of a  p = &b; // address of b  p = &c; // address of c |

*Hạn chế:*  
– Ta không thể sử dụng trực tiếp dữ liệu mà con trỏ void trỏ tới bằng toán tử (\*), mà cần biến đổi con trỏ (void\*) sang thành kiểu dữ liệu tương ứng.

*Ví dụ 1:*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #include<stdio.h>  #include<conio.h>  **int** main()  {  **int** a = 10;  **void** \*p = &a;  **int** \*ptr = p; // error  **printf**("%u\n",\*ptr);      getch();  } |

*Kết quả:*Trong đoạn code trên khai báo con trỏ p kiểu void trỏ đến địa chỉ biến a (câu lệnh này OK). Câu lệnh tiếp theo, trình biên dịch sẽ không thể biết biến p có kiểu dữ liệu là gì -> Complier Error.

Con trỏ void

*Ví dụ 2:*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #include <stdio.h>  #include <conio.h>  **int** main()  {  **int** a = 10;  **void** \*p = &a;  **int** \*ptr = (**int**\*)p; // OK  **printf**("%u\n", \*ptr); // 10      getch();  } |

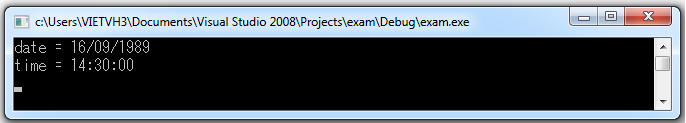
Ở ví dụ này, ta đã fix lỗi ở ví dụ 1 bằng cách ép kiểu con trỏ (void\*) thành (int\*). Lúc này complier sẽ hiểu là con trỏ p trỏ tới kiểu int.

Ví dụ 3: minh họa con trỏ void có thể trỏ tới các kiểu dữ liệu khác như struct, class

|  |  |
| --- | --- |
| 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  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | #include <conio.h>  #include <stdio.h>    **typedef** **struct**  {  **int** date;  **int** month;  **int** year;  }DATE;    **typedef** **struct**  {  **int** hour;  **int** minute;  **int** second;  }TIME;    **void** showTime(**void** \*p, **char** type);    **void** main()  {      DATE d = {16, 9, 1989};      TIME t = {14, 30, 0};      showTime((**void**\*)&d, 'd');      showTime((**void**\*)&t, 't');      getch();  }    **void** showTime(**void** \*p, **char** type)  {      DATE \*pDate;      TIME \*pTime;  **switch**(type)      {  **case** 'd':          pDate = (DATE\*)p;  **printf**("date = %.02d/%.02d/%.04d\n", pDate->date, pDate->month, pDate->year);  **break**;  **case** 't':          pTime = (TIME\*)p;  **printf**("time = %.02d:%.02d:%.02d\n", pTime->hour, pTime->minute, pTime->second);  **break**;  **default**:  **printf**("Default case");      }  } |

Giải thích:  
Đoạn code trên hiển thị thông tin của struct DATE và TIME. Thông thường, chúng ta sẽ phải viết 2 hàm hiển thị cho 2 struct. Tuy nhiên, chúng ta có thể sử dụng tham số kiểu void\* để ép kiểu khi truyền đối số là kiểu DATE và TIME.  
Khi vào hàm showTime(), dựa vào biến type để ép từ kiểu void\* về kiểu dữ liệu tương ứng. Điều này làm giảm thiểu việc viết nhiều function cùng chức năng nhưng khác kiểu dữ liệu đầu vào.

Kết quả:

Kiểu void\* trỏ tới struct

# Sự khác nhau giữa hàm malloc() và calloc()?

**Điểm giống nhau giữa malloc() và calloc()**

Cả 2 hàm malloc() và calloc() đều được sử dụng để cấp phát vùng nhớ động cho chương trình. Nếu cấp phát vùng nhớ thành công, hàm trả về con trỏ trỏ tới vùng nhớ được cấp phát. Hàm trả về NULL nếu không đủ vùng nhớ.

Hàm malloc() và calloc() trả về NULL trong các trường hợp sau:

* Kích thước vùng nhớ cần cấp phát vượt quá kích thước vật lý của hệ thống
* Tại thời điểm gọi hàm malloc() và calloc(), tạm thời không đủ vùng nhớ để cấp phát. Nhưng application có thể gọi lại nhiều lần malloc() và calloc() để cấp phát vùng nhớ thành công.

**Điểm khác nhau giữa hàm malloc() và calloc()**

|  |  |
| --- | --- |
| malloc() | calloc() |
| malloc viết tắt của memory allocation | calloc viết tắt của contiguous allocation |
| malloc nhận 1 tham số truyền vào là số byte của vùng nhớ cần cấp phát | calloc nhận 2 tham số truyền vào là số block và kích thước mỗi block (byte) |
| void \*malloc(size\_t n);  Hàm trả về con trỏ trỏ tới vùng nhớ nếu cấp phát thành công, trả về NULL nếu cấp phát fail | void \*calloc(size\_t n, size\_t size);  Hàm trả về con trỏ trỏ tới vùng nhớ được cấp phát và vùng nhớ được khởi tạo bằng giá trị 0. Trả về NULL nếu cấp phát fail |
| Hàm malloc() nhanh hơn hàm calloc() | Hàm calloc() tốn thêm thời gian khởi tạo vùng nhớ. Tuy nhiên, sự khác biệt này không đáng kể. |

Ví dụ: Sử dụng malloc và calloc để cấp phát vùng nhớ

|  |  |
| --- | --- |
| 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 <stdlib.h>  #include <conio.h>      **void** main( **void** )  {  **int** \*p1, \*p2;      p1 = (**int**\*)**malloc**(10\***sizeof**(**int**));  **if**(p1)  **printf**("\nmalloc() allocates memory successfully");  **else**  **printf**("\nFail in allocate memory");        p2 = (**int**\*)**calloc**(10, **sizeof**(**int**));  **if**(p2)  **printf**("\ncalloc() allocates memory successfully");  **else**  **printf**("\nFail in allocate memory");        getch();  } |

# Sự khác nhau giữa #include “filename” and #include < filename >?

Lệnh #include <filename> và #include “filename” là tương đương nhau, để include header file của thư viện ngôn ngữ C hoặc header file do lập trình viên tự định nghĩa.

Sự khác nhau giữa #include <filename> and #include “filename” nằm ở khâu tìm kiếm file header của tiền xử lý trước quá trình biên dịch.

* #include <filename>: tiền xử lý (pre-processor) sẽ chỉ tìm kiếm file header (.h) trong thư mục chứa file header của thư viện ngôn ngữ C (thường là thư mục trong bộ cài IDE).
* #include “filename”: Trước tiên, tiền xử lý (pre-processor) tìm kiếm file header(.h) trong thư mục đặt project C/C++. Nếu không tìm thấy, tiền xử lý tìm kiếm file header (.h) trong thư mục chứa file header của thư viện ngôn ngữ C (thường là thư mục trong bộ cài IDE).

Tóm lại,

* Khi cần sử dụng thư viện file header (.h), chúng ta nên sử dụng #include <filename>. Tất nhiên, sử dụng #include “filename” không sai, nhưng tiền xử lí sẽ mất thời gian tìm kiếm trong thư mục project trước khi tìm kiếm trong thư mục header của IDE.
* Khi cần gọi file header (.h) tự định nghĩa, chúng ta nên sử dụng #include “filename”. Chú ý: nếu file header (.h) được đặt ở đường dẫn khác thư mục project, chúng ta cần phải chỉ định đường dẫn tương đối cho file header: #include “..\…\filename.h”

Ví dụ: Khai báo file header của thư viện và file header tự định nghĩa

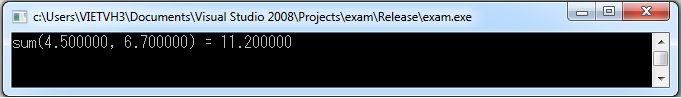
File sum.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | #ifndef \_SUM\_H\_  #define \_SUM\_H\_    **float** sum(**float**, **float**);    #endif |

File sum.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #include <stdio.h>  #include <conio.h>  #include "sum.h"      **void** main( **void** )  {  **float** a = 4.5, b = 6.7;  **printf**("sum(%f, %f) = %f", a, b, sum(a, b));      getch();  }    **float** sum(**float** a, **float** b)  {  **return** (a + b);  } |

Kết quả:

Tổng 2 số

* #include <stdio.h> là thư viện cho hàm printf()
* #include <conio.h> là thư viện cho hàm getch()
* #include “sum.h” được định nghĩa khai bao hàm sum()

# Macro và Hàm trong ngôn ngữ C

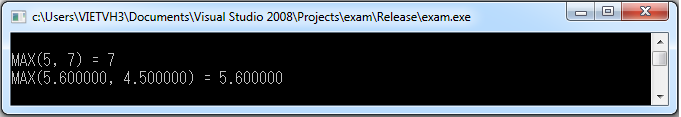
**Macro là gì?**

* Marco là 1 tên bất kì (do lập trình viên đặt tên) trỏ tới 1 khối lệnh thực hiện một chức năng nào đó.
* Trong quá trình tiền xử lí (pre-processor), các macro được sử dụng trong chương trình được thay thế bởi các khối câu lệnh tương ứng.
* Định nghĩa macro bằng lệnh #define

Ví dụ: định nghĩa macro tính giá trị lớn nhất của 2 số.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #include <stdio.h>  #include <conio.h>    #define MAX(A, B) ((A) > (B) ? (A) : (B))    **void** main( **void** )  {  **int** a = 5, b = 7;  **float** c = 5.6, d = 4.5;  **printf**("\nMAX(%d, %d) = %d", a, b, MAX(a, b));  **printf**("\nMAX(%f, %f) = %f", c, d, MAX(c, d));      getch();  } |

Kết quả:

Macro tìm giá trị lớn nhất của 2 số nguyên

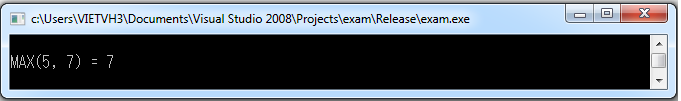
**Hàm là gì?**

Function là 1 khối lệnh thực hiện một chức năng nào đó.

Ví dụ: viết hàm tìm giá trị lớn nhất 2 số nguyên

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #include <stdio.h>  #include <conio.h>    **int** MAX(**int**, **int**);    **void** main( **void** )  {  **int** a = 5, b = 7;  **printf**("\nMAX(%d, %d) = %d", a, b, MAX(a, b));      getch();  }    **int** MAX(**int** a, **int** b)  {  **return** (a > b ? a : b);  } |

Kết quả:

Hàm tìm giá trị lớn nhất của 2 số nguyên

**Ưu nhược điểm của Macro và Hàm**

|  |  |
| --- | --- |
| Macro | Hàm |
| Việc định nghĩa macro khó hơn định nghĩa hàm.  Nếu không chú ý, chúng ta dễ bị side effect. (Mình sẽ phân tích ở phía dưới) | Việc định nghĩa đơn giản hơn |
| không thể debug tìm lỗi của macro trong thời gian thực thi. | Debug đơn giản, dễ bắt lỗi |
| Macro không cần quan tâm kiểu dữ liệu của tham số và kiểu trả về. Như ví dụ trên, chúng ta có thể truyền kiểu int, float. | Phải chỉ rõ kiểu dữ liệu của tham số và giá trị trả về |
| Macro tạo ra các inline code, thời gian xử lí inline code ngắn hơn thời gian gọi hàm | Chương trình mất time dịch từ vùng nhớ hàm được lưu trữ sang vùng nhớ goi hàm. |
| Giả sử macro được gọi 20 lần trong chương trình, 20 dòng code sẽ được chèn vào chương trình trong quá trình tiền xử lí. Điều này làm cho kích thước của chương trình (.EXE, .DLL, .LIB,…) phình to ra. | Giả sử 1 hàm được gọi 20 lần, sẽ chỉ có 1 bản copy của hàm trong chương trình. Kích thước chương trình nhỏ hơn sử dụng macro. |

Tùy thuộc vào tiêu chí thời gian thực thi hay kích thước chương trình, mà lập trình viên quyết định chọn macro hay hàm trong chương trình của mình. Đối với khối chức năng đơn giản ít dòng code, nên sử dụng macro.

**Chú ý: khi sử dụng macro để tránh side effect**

Ví dụ: tạo macro tính bình phương số nguyên

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #include <stdio.h>  #include <conio.h>    #define SQUARE(X) (X\*X)    **void** main( **void** )  {  **printf**("SQUARE(%d) = %d", 3+5, SQUARE(3+5));      getch();  } |

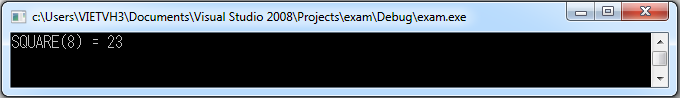
Giải thích:

Kết quả mong muốn sẽ là 8\*8 = 64. Nhưng  thực tế, kết quả như sau: 3+5\*3+5 = 3 + 15 + 5 = 23.

Khi bạn định nghĩa macro, phải chú ý dấu ngoặc. Macro SQUARE được update lại như sau:

#define SQUARE(X) ((X)\*(X))

Kết quả:

Hàm tính bình phương số nguyên

# Sự khác nhau giữa struct và union

Struct và Union là 2 cấu trúc dữ liệu do lập trình viên định nghĩa bao gồm các biến với kiểu dữ liệu khác nhau.

**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:

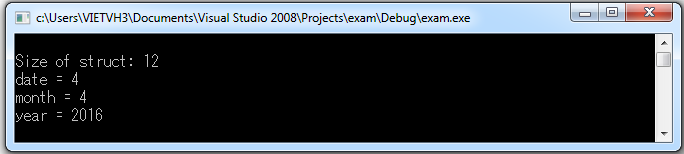
|  |  |
| --- | --- |
| **struct** | **union** |
| Size của struct ít nhất bằng tổng size của các thành phần của struct. Mình sử dụng từ “ít nhất” là vì size struct còn phụ thuộc vào alignment struct.  sizeof(A) = 4 + 10 + 4 = 18 (trong trường hợp struct alignment member = 1 byte)   |  |  | | --- | --- | | 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();    } |

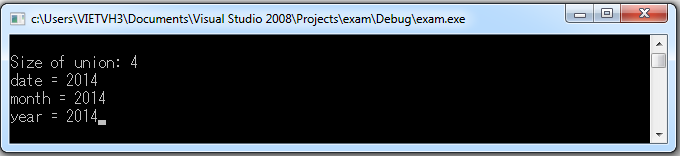
Kết quả:

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.

# Sự khác nhau giữa #define và const?

Như các bạn đã biết #define được sử dụng để định nghĩa hằng số, macro.

Trong bài viết này, chúng ta chỉ phân tích sự khác nhau #define và const trong việc khai báo hằng số.

1. Sử dụng #define để định nghĩa hằng số như: kích thước mảng, không thể sử dụng const để định nghĩa hằng số như: kích thước mảng.

# Sự khác nhau giữa bộ nhớ stack và heap?

Bộ nhớ stack và heap đều được lưu trên RAM của PC.  Dưới đây là một số điểm khác nhau của stack và heap.

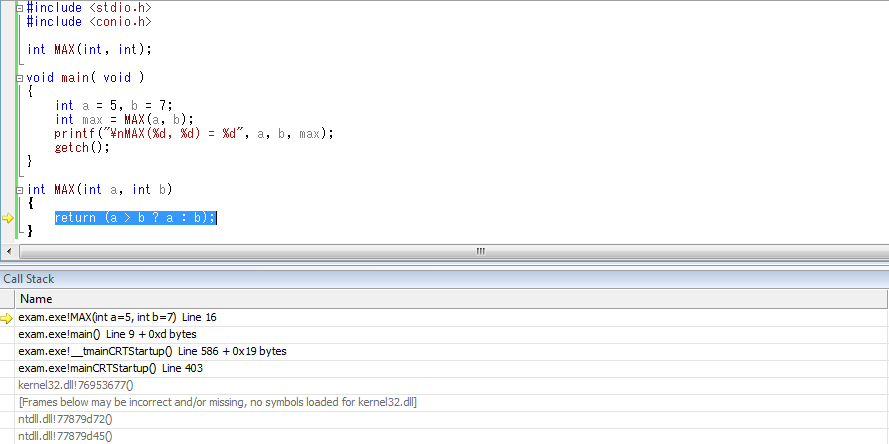
|  |  |
| --- | --- |
| **Stack** | **Heap** |
| * Vùng nhớ được cấp phát khi chương trình được biên dịch. | * Vùng nhớ được cấp phát khi chạy chương trình (run-time). |
| * Vùng nhớ stack được sử dụng cho việc thực thi thread. Khi gọi hàm, các biến cục bộ của hàm được lưu trữ vào block của stack (theo kiểu LIFO). Cho đến khi hàm trả về giá trị, block này sẽ được xóa tự động. Hay nói cách khác, các biến cục bộ được lưu trữ ở vùng nhớ stack và tự động được giải phóng khi kết thúc hàm. | * Vùng nhớ heap được dùng cho cấp phát bộ nhớ động (malloc( ), new( )). Vùng nhớ được cấp phát tồn tại đến khi lập trình viên giải phóng vùng nhớ bằng lệnh free( ) hoặc delete. |
| * Kích thước vùng nhớ stack được fix cố định. Chúng ta không thể tăng hoặc giảm kích thước vùng nhớ stack. Nếu không đủ vùng nhớ stack, gây ra stack overflow. Hiện tượng này xảy ra khi nhiều hàm lồng nhau hoặc đệ quy nhiều lần dẫn đến không đủ vùng nhớ. | * Khi kích thước vùng nhớ heap không đủ cho yêu cầu malloc( ), new. Hệ điều hành sẽ có cơ chế tăng kích thước vùng nhớ heap. |

Ví dụ: Minh họa stack được sử dụng khi gọi hàm

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | #include <stdio.h>  #include <conio.h>    **int** MAX(**int**, **int**);    **void** main( **void** )  {  **int** a = 5, b = 7;  **int** max = MAX(a, b);  **printf**("\nMAX(%d, %d) = %d", a, b, max);      getch();  }    **int** MAX(**int** a, **int** b)  {  **return** (a > b ? a : b);  } |

Giải thích:

Trên đây là chương trình tìm giá trị lớn nhất của 2 số nguyên. Khi chạy chương trình ở chế độ debug và xem call stack trên visual studio.

Call Stack

Các bạn nhìn Call Stack sẽ thấy hàm main( ) gọi hàm MAX( ). Hàm main( ) và MAX( ) được lưu theo quy tắc (LIFO: last in – first out). Khi kết thúc hàm MAX( ), các thông tin lưu trữ hàm MAX( ) bị xóa, stack chỉ lưu trữ thông tin của hàm main( ). Tương tự như vậy, khi kết thúc hàm main( ), stack được xóa hoàn toàn.

Call stack

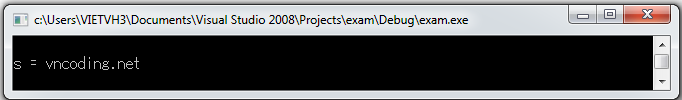
# Sự khác nhau giữa hàm strcpy() và strdup()?

* Hàm strcpy(char\* s1, char\* s2) copy nội dung vùng nhớ s2 và vùng nhớ s1. Vùng nhớ s1 phải được cấp phát tĩnh hoặc cấp phát động trước đó (malloc( ), new( )). Kích thước vùng nhớ s1 phải đủ để chứa chuỗi s2. Nếu không đủ vùng nhớ, gây ra buffer overrun.

Ví dụ:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | #include <stdio.h>  #include <conio.h>  #include <string.h>    **void** main( **void** )  {  **char** s[20];  **char** p[] = "vncoding.net";  **strcpy**(s, p);  **printf**("\ns = %s", s);      getch();  } |

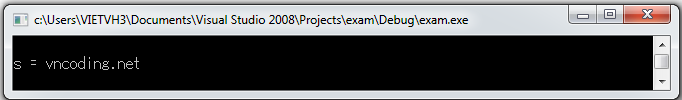
Kết quả:

Hàm strcpy( )

* Hàm strdup(const char\* s) tạo ra vùng nhớ mới (gọi hàm malloc( ) để cấp phát vùng nhớ) để lưu chuỗi s. Sau đó, trả về con trỏ trỏ tới vùng nhớ được cấp phát. Nếu cấp phát fail, hàm trả về NULL. Chú ý: phải free vùng nhớ trả về bởi hàm strdup.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #include <stdio.h>  #include <conio.h>  #include <string.h>    **void** main( **void** )  {  **char** \*s = strdup("vncoding.net");  **if**(s)      {  **printf**("\ns = %s", s);  **free**(s);      }      getch();  } |

Kết quả:

Hàm strdup( )

# Sự khác nhau giữa mảng và danh sách liên kết?

Mảng và danh sách liên kết đều là cấu trúc dự liệu phổ biến trong ngôn ngữ lập trình. Dưới đây là các điểm khác nhau giữa mảng và [**danh sách liên kết**](http://vncoding.net/danh-sach-lien-ket/)

|  |  |
| --- | --- |
| **Mảng** | **Danh sách liên kết** |
| Vùng nhớ của các phần tử trong mảng được sắp xếp liên tục nhau | Vùng nhớ của các phần tử trong danh sách liên kết được sắp xếp tùy ý (do hệ điều hành). Các phần tử lưu 1 con trỏ trỏ tới phần tử tiếp theo. |
| Truy cập tới phần tử trong mảng là truy cập trực tiếp dựa vào chỉ số (ví dụ: a[0], a[1], a[2],…, a[n]) | Cần phải duyệt tuần tự khi muốn truy cập tới phần tử trong danh sách liên kết. |
| * Kích thước của mảng là hằng số, không thay đổi khi chạy chương trình * Sử dụng mảng không tối ưu được bộ nhớ. Có thể thừa hoặc thiếu bộ nhớ khi xóa hoặc chèn phần tử vào mảng | * Kích thước của danh sách liên kết có thể thay đổi khi chạy chương trình. * Sử dụng danh sách liên kết tối ưu được bộ nhớ. Vùng nhớ được cấp phát thêm khi cần chèn thêm phần tử mới, vùng nhớ được free khi xóa phần tử. |

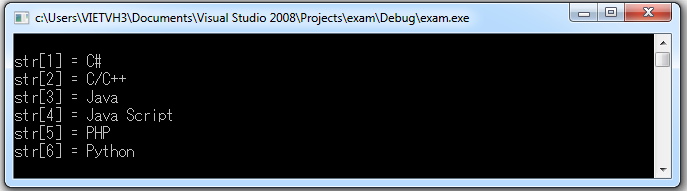
# Mảng con trỏ trong lập trình C

Trước khi tìm hiểu mảng con trỏ, chúng ta xét vài ví dụ để hiểu được vì sao mảng con trỏ là cần thiết trong ngôn ngữ lập trình C.

Ví dụ 1: sắp xếp chuỗi theo thứ tự alpha, beta,..

|  |  |
| --- | --- |
| 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  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | #include <stdio.h>  #include <string.h>  #include <conio.h>    #define LENGTH 30    **void** strsort(**char** str[][LENGTH], **int** num);  **void** strshow(**char** str[][LENGTH], **int** num);    **void** main( **void** )  {  **char** strarr[][LENGTH] = {"C/C++", "Java", "C#", "Python", "PHP", "Java Script"};      strsort(strarr, 6);      strshow(strarr, 6);      getch();  }    **void** strsort(**char** str[][LENGTH], **int** num)  {  **int** i, j;  **char** temp[LENGTH];  **for** (i = 0; i < num; i++)      {  **for** (j = num-1; j > i; j--)          {  **if**(**strcmp**(str[j], str[j-1]) < 0)              {  **strcpy**(temp, str[j]);  **strcpy**(str[j], str[j-1]);  **strcpy**(str[j-1], temp);              }            }      }  }    **void** strshow(**char** str[][LENGTH], **int** num)  {  **int** i;  **for** (i = 0; i < num; i++)      {  **printf**("\nstr[%d] = %s", i+1, str[i]);      }  } |

Kết quả:

Sắp xếp chuỗi theo thứ tự alpha, beta

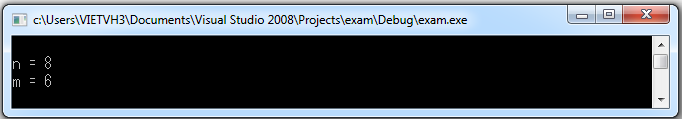
Qua ví dụ ở trên, chúng ta thấy để sắp xếp 1 mảng các chuỗi yêu cầu swap các chuỗi với nhau ( sử dụng hàm strcpy( ) hoán đổi nội dung giữa các chuỗi với nhau). Khi gặp các bài toán với kích thước dữ liệu lớn (chuỗi, struct,…), việc copy dữ liệu làm giảm performance của chương trình.

→ Dùng con trỏ trỏ tới vùng nhớ của dữ liệu (chuỗi, struct,…), sau đó swap con trỏ.

Ví dụ 2:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **void** main( **void** )  {  **int** n = 6, m = 8;  **int** \*p1, \*p2, \*p;      p1 = &n;      p2 = &m;      //swap p1 and p2      p = p1;      p1 = p2;      p2 = p;    **printf**("\nn = %d", \*p1);  **printf**("\nm = %d", \*p2);        getch();  } |

Kết quả:

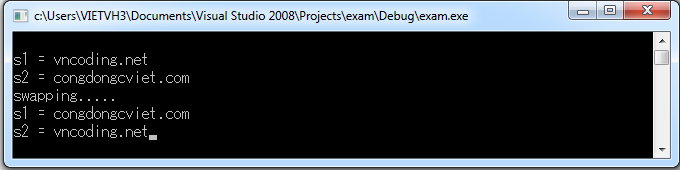
hoán vị 2 số nguyên

Ở đây, chúng ta sử dụng 2 con trỏ p1 và p2 trỏ tới địa chỉ biến n và m. Thay vì hoán vị trực tiếp 2 biến n và m, chúng ta có thể hoán vị con trỏ p1 và p2. Từ ý tưởng này, chúng ta có thể sử dụng con trỏ để hoán vị 2 chuỗi cho nhau. Các bạn xem tiếp ví dụ sau:

Ví dụ 3: hoán vị 2 chuỗi

|  |  |
| --- | --- |
| 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>    **void** main( **void** )  {  **char** s1[] = "vncoding.net";  **char** s2[] = "congdongcviet.com";    **char** \*p1, \*p2, \*p;      p1 = s1;      p2 = s2;    **printf**("\ns1 = %s", p1);  **printf**("\ns2 = %s", p2);    **printf**("\nswapping.....");        //swap 2 strings      p = p1;      p1 = p2;      p2 = p;    **printf**("\ns1 = %s", p1);  **printf**("\ns2 = %s", p2);        getch();  } |

Kết quả:

hoán vị 2 chuỗi

Con trỏ p1, p2 trỏ tới chuỗi s1, s2 tương ứng. Để đổi chỗ 2 chuỗi, chúng ta hoán vị 2 con trỏ.

Tổng kết lại, chúng ta có thể sử dụng mảng con trỏ để trỏ tới các phần tử trong mảng dữ liệu (string, struct,…) tương ứng. Việc sắp xếp các phần tử trong mảng dữ liệu (string, struct,…) dựa vào việc hoán vị các con trỏ. Mảng con trỏ sẽ lưu kết quả sắp xếp mảng (các phần tử mảng dữ liệu thực tế không được hoán vị).

Việc hoán vị con trỏ tăng performance của chương trình so với cách truyền thống như ở ví dụ 1

Chúng ta xét ví dụ tiếp theo về cách khai báo và sử dụng mảng con trỏ để sắp xếp mảng chuỗi kí tự.

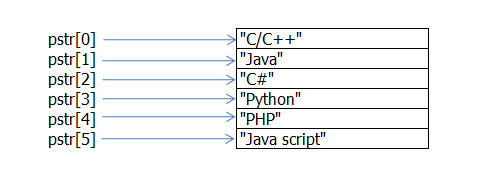
Ví dụ 4:

|  |  |
| --- | --- |
| 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  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | #include <stdio.h>  #include <string.h>  #include <conio.h>    #define LENGTH 30  #define SIZE 10    **void** strsort(**char**\*\* str, **int** num);  **void** strshow(**char**\*\* str, **int** num);  **void** main( **void** )  {  **char** strarr[][LENGTH] = {"C/C++", "Java", "C#", "Python", "PHP", "Java Script"};  **char**\* pstr[SIZE];  **int** i;  **for** (i = 0; i < 6; i++)      {          pstr[i] = strarr[i];      }        strsort(pstr, 6);      strshow(pstr, 6);      getch();  }    **void** strsort(**char**\*\* str, **int** num)  {  **int** i, j;  **char**\* temp;  **for** (i = 0; i < num; i++)      {  **for** (j = num-1; j > i; j--)          {  **if**(**strcmp**(str[j], str[j-1]) < 0)              {                  temp = str[j];                  str[j] = str[j-1];                  str[j-1] = temp;              }          }      }  }    **void** strshow(**char**\*\* str, **int** num)  {  **int** i;  **for** (i = 0; i < num; i++)      {  **printf**("\nstr[%d] = %s", i+1, str[i]);      }  } |

Giải thích:

|  |  |
| --- | --- |
| 1  2  3  4 | **for** (i = 0; i < 6; i++)  {      pstr[i] = strarr[i];  } |

Gán phần tử mảng con trỏ trỏ tới mỗi chuỗi trong mảng chuỗi.

Gán con trỏ cho chuỗi kí tự

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **if**(**strcmp**(str[j], str[j-1]) < 0)  {      temp = str[j];      str[j] = str[j-1];      str[j-1] = temp;  } |

Hoán vị con trỏ. Sau khi thực hiện xong hàm strsort(). Giá trị mảng con trỏ như sau:

Mảng con trỏ sau khi sắp xếp

Kết quả mảng con trỏ lưu kết quả sắp xếp. Do vậy, mọi thao tác: hiển thị thông tin mảng được sắp xếp ra màn hình, ghi ra file đều được xử lí trên mảng con trỏ.

Sắp xếp chuỗi dùng mảng con trỏ

# Con trỏ hàm và ứng dụng trong C/C++?

## Con trỏ hàm là gì?

Như các bạn đã biết, con trỏ lưu địa chỉ của biến. Tương tự, con trỏ hàm lưu địa chỉ của hàm.

## Ứng dụng con trỏ hàm

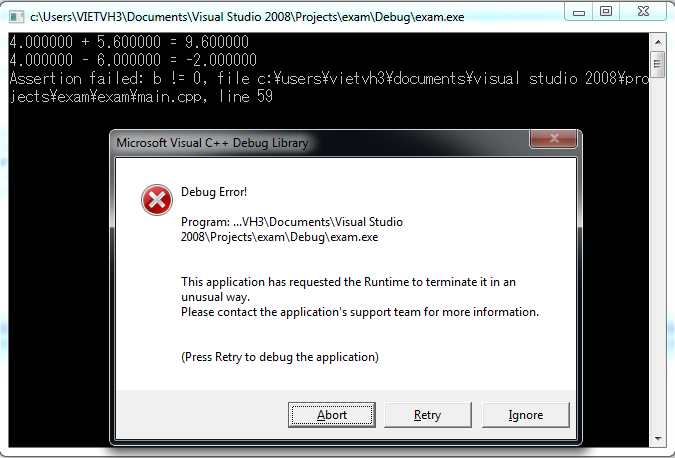
### Thay thế cho lệnh switch case

Xét ví dụ viết các hàm thực hiện phép toán cộng, trừ, nhân, chia. Tại mỗi thời điểm người dùng sẽ chọn 1 trong số các phép toán để thực hiện tính toán.

|  |  |
| --- | --- |
| 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  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61 | #include <stdio.h>  #include <conio.h>  #include <assert.h>    **float** Plus(**float** a, **float** b);  **float** Minus(**float** a, **float** b);  **float** Multiply(**float** a, **float** b);  **float** Divide(**float** a, **float** b);  **void** Switch(**float** a, **float** b, **char** opCode);    **void** main()  {      Switch(4, 5.6, '+');      Switch(4, 6, '-');      Switch(4, 0, '/');      getch();  }    **void** Switch(**float** a, **float** b, **char** opCode)  {  **float** result;        // execute operation  **switch**(opCode)      {  **case** '+' :          result = Plus(a, b);  **break**;  **case** '-' :          result = Minus(a, b);  **break**;  **case** '\*' :          result = Multiply(a, b);  **break**;  **case** '/' :          result = Divide(a, b);  **break**;      }  **printf**("%f %c %f = %f\n", a, opCode, b, result);  }    **float** Plus(**float** a, **float** b)  {  **return** a+b;  }    **float** Minus(**float** a, **float** b)  {  **return** a-b;  }    **float** Multiply(**float** a, **float** b)  {  **return** a\*b;  }    **float** Divide(**float** a, **float** b)  {  **assert**(b != 0);  **return** a/b;  } |

Giải thích: Hàm[**assert()**](http://vncoding.net/2015/11/11/ham-assert-danh-gia-bieu-thuc-dau-vao/) đánh giá biến ‘b’ nếu b = 0, exception message xuất hiện.

Kết quả:

Phép toán cộng, trừ, nhân, chia

Để thực thay  thế việc dùng lệnh switch case, chúng ta có thể sử dụng con trỏ hàm để giải quyết bài toán này.

|  |  |
| --- | --- |
| 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  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | #include <stdio.h>  #include <conio.h>  #include <assert.h>    **float** Plus(**float** a, **float** b);  **float** Minus(**float** a, **float** b);  **float** Multiply(**float** a, **float** b);  **float** Divide(**float** a, **float** b);  **void** Switch(**float** a, **float** b, **float** (\*p2Func)(**float**, **float**));    **void** main()  {      Switch(4, 5.6, &Plus);      Switch(4, 6, &Minus);      Switch(4, 0, &Divide);      getch();  }    **void** Switch(**float** a, **float** b, **float** (\*p2Func)(**float**, **float**))  {  **float** result = p2Func(a, b);  **printf**("result = %f\n", result);  }    **float** Plus(**float** a, **float** b)  {  **return** a+b;  }    **float** Minus(**float** a, **float** b)  {  **return** a-b;  }    **float** Multiply(**float** a, **float** b)  {  **return** a\*b;  }    **float** Divide(**float** a, **float** b)  {  **assert**(b != 0);  **return** a/b;  } |

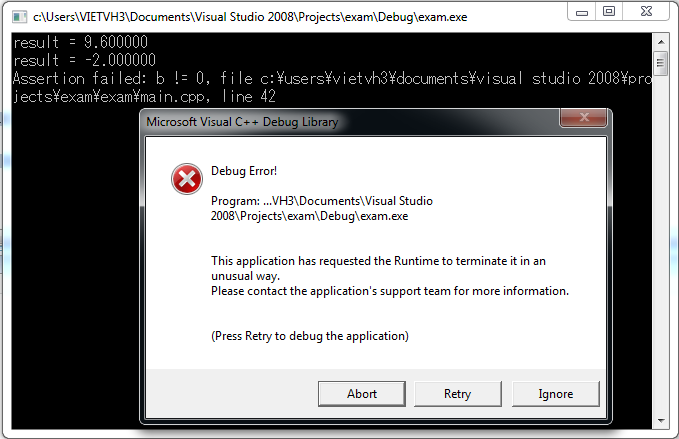
Giải thích:

Hàm Switch() đã được modify bằng cách thay tham số cuối bằng con trỏ hàm. Và đối số truyền vào là địa chỉ hàm: &Plus, &Minus,…

Chú ý:

* Mỗi con trỏ hàm được khai báo với tham số (số lượng + kiểu dữ liệu) và giá trị trả về
* Khi bạn muốn sử dụng 1 con trỏ hàm trỏ tới nhiều hàm (như ví dụ trên), thì tham số (số lượng + kiểu dữ liệu) và giá trị trả về của con trỏ hàm và các hàm được trỏ tới phải giống nhau.

Kết quả:

Phép toán cộng, trừ, nhân, chia

# Double pointer (pointer to pointer) và ứng dụng?

**Double pointer là gì?**

Chúng ta đã biết con trỏ lưu trữ địa chỉ của biến. Khi con trỏ lưu trữ địa chỉ của 1 con trỏ thì được gọi là double pointer hoặc pointer to pointer.

Khai báo:

|  |  |
| --- | --- |
| 1 | <kiểu dữ liệu>\*\* <tên con trỏ>; |

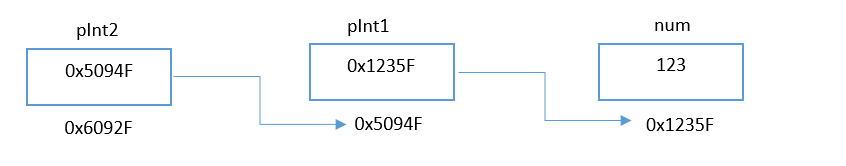
Ví dụ 1: khai báo double pointer

|  |  |
| --- | --- |
| 1  2  3 | **int**\*\* pInt;  **char**\*\* pStr;  **float**\*\* pFt; |

Để hiểu double pointer, chúng ta cùng xem ví dụ 2 và hình minh họa dưới đây  
Ví dụ 2: khai báo double pointer và phép gán.

|  |  |
| --- | --- |
| 1  2  3 | **int** num = 123;  **int**\* pInt1 = &num;  **int**\*\* pInt2 = &pInt1; |

Giải thích:  
– Biến int num có giá trị 123 được lưu tại địa chỉ &num = 0x1235F  
– Con trỏ pInt1 trỏ tới biến num, giá trị pInt1 = &num = 0x1235F, \*pInt1 = num = 123  
– Con trỏ double pInt2 trỏ tới con trỏ pInt1, giá trị pInt2 = &pInt1 = 0x5094F, \*pInt2 = pInt1 = &num = 0x1235F, \*\*pInt2 = \*pInt1 = num = 123



**Ứng dụng double pointer?**

Một trong những ứng dụng phổ biến nhất double pointer là thay đổi giá trị của con trỏ được truyền vào như 1 đối số của hàm. Xét ví dụ sau  
Ví dụ 3: Viết hàm cấp phát bộ nhớ động sử dụng tham số double pointer

|  |  |
| --- | --- |
| 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  28  29  30  31  32  33  34  35  36  37  38 | #include <stdio.h>  #include <stdlib.h>  #include <memory.h>    **bool** memalloc(**int**\*\*, **int**);  **void** memfree(**int**\*\*);    **void** main()  {  **int**\* pInt;  **bool** bRes = memalloc(&pInt, 100);  **if** (bRes)      {  **printf**("\nMemmory allocation is successful");      }      memfree(&pInt);  }    **bool** memalloc(**int**\*\* p, **int** sz)  {      \*p = (**int**\*)**malloc**(sz \* **sizeof**(**int**));  **if** (\*p == NULL)      {          \*p = NULL;  **return** **false**;      }  **memset**(\*p, 0x00, sz \* **sizeof**(**int**));  **return** **true**;  }    **void** memfree(**int**\*\* p)  {  **if** (\*p)      {  **free**(\*p);          \*p = NULL;      }  } |

Giải thích:  
Sau khi thực hiện hàm memalloc(), con trỏ pInt trỏ tới vùng nhớ 100 byte. Sau khi thực hiện hàm memfree(), giải phóng vùng nhớ trỏ bởi pInt và pInt được gán bằng NULL.

Output:

Memmory allocation is successful

# Alignment và padding struct

**Alignment data và padding data**

Data alignment là cách mà trình biên dịch sắp xếp dữ liệu trong memory. Việc sắp xếp dữ liệu liên quan tới 2 khái niệm: data alignment và data padding.

Đối với những PC hiện nay, việc đọc/ghi dữ liệu trong bộ nhớ được với kích thước của  WORD (OS 32 bit: kích thước WORD: 4 byte) hoặc lớn hơn. Data alignment là cách sắp xếp dữ liệu sao cho kích thước bằng bội số của kích thước 1 word = 4k (byte). (k = 0, 1, 2,…). Để alignment data, đôi khi chúng ta cần phải add thêm các byte giả (dummy) vào vị trí thích hợp để đảm bảo data alignment, được gọi là padding data.

Data Alignment  làm tăng performance do việc đọc/ghi thao tác trên block data có kích thước bằng bội số của WORD.

**Data Alignment trên Visual Studio C++**

Khi chúng ta định nghĩa struct trong project Visual Studio C++, trình biên dịch (complier) sẽ cố gắng allocate dữ liệu theo alignment data.

Trình biên dịch sẽ lấy kiểu dữ liệu có kích thước lớn nhất trong struct làm kích thước đường biên cho việc alignment data.

Ví dụ 1:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #include <stdio.h>  #include <conio.h>    **struct** A  {  **char** a;     // 1 byte  **int** b;      // 4 bytes  **short** c;    // 2 bytes  **char** d;     // 1 byte  };    **void** main()  {  **printf**("size = %d", **sizeof**(A));      getch();  } |

Kết quả:

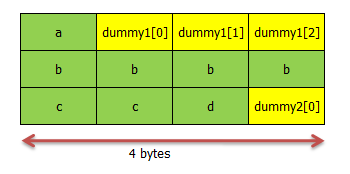
Khi mới học lập trình C/C++, mình nghĩ kích thước của struct A bằng: 1+4+2+1 = 8 (byte). Wrong!!!

Kích thước của struct A chỉ bằng tổng các thành phần cộng lại khi bạn sử dụng packing. Mình sẽ trình bày ở mục bên dưới. Thực tế kích thước của struct A bằng 12. Lý do, trình biên dịch đã padding data (add thêm một số byte giả trong lúc biên dịch để đảm bảo data alignment) như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **struct** A  {  **char** a;     // 1 byte  **char** dummy1[3]; // padding 3 bytes  **int** b;      // 4 bytes  **short** c;    // 2 bytes  **char** d;     // 1 byte  **char** dummy2[1]; // padding 1 byte  }; |

Trình biên dịch chọn kích thước kiểu int (4 bytes) để align data.

* 3 byte dummy1[3] được add thêm vào để: sizeof(a) + sizeof(dummy1) = 4 bytes
* biến int b có kích thước 4 byte nên không cần padding.
* 1 byte dummy2[1] được add thêm vào để: sizeof(c) + sizeof(d) + sizeof(dummy2) = 4 bytes.

Data Alignment

**Struct packing và Alignment**

Trình biên dịch tự động alignment data làm tăng performance của hệ thống. Tuy nhiên, kích thước struct lớn hơn dự định ban đầu. (do padding thêm một số byte) → làm tăng kích thước chương trình.

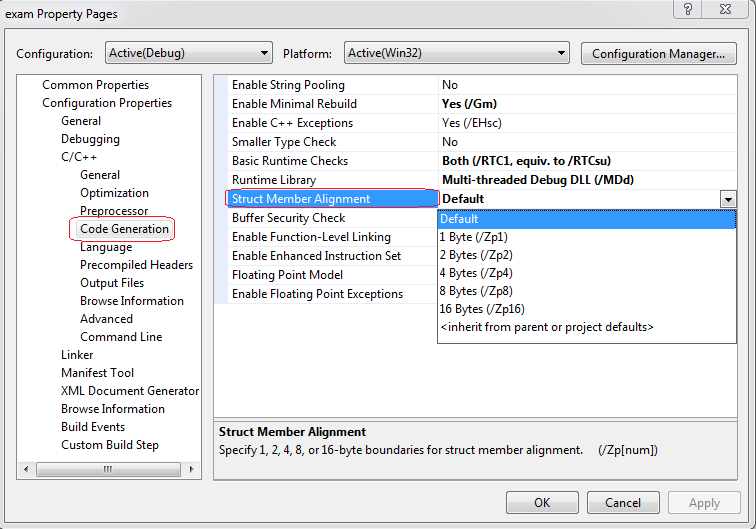
Để kiểm soát kích thước của struct mà vẫn đảm bảo performace, một khái niệm mới struct packing ra đời.

Mối liên hệ giữa alignment và packing data

* Nếu packsize >= kích thước đường biên, giá trị packsize bị bỏ qua và trình biên dịch lấy kích thước đường biên để align struct
* Nếu packsize < kích thước đường biên, trình biên dịch align struct dựa vào packsize.

Packsize là kích thước đóng gói struct. Được setting bằng 2 cách:

Cách 1: Chọn properties → Code Generation → Struct Member Alignment

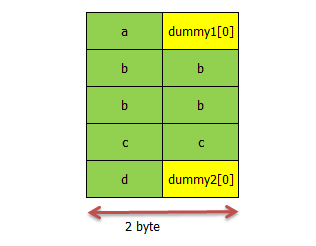
Setting packsize

packsize có các giá trị 1, 2, 4, 8, 16 bytes. (Defaut: 8 bytes)

Như ví dụ 1, packsize là default (8 bytes), nên trình biên dịch lấy sizeof(int) = 4 bytes làm đường biên. Dưới đây, mình đưa thêm một ví dụ packsize = 2.

 Ví dụ 2: source code giống ví dụ 1 + packsize = 2, kết quả size = 10. Trình biên dịch lấy 2 bytes làm đường biên để align data như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **struct** A  {  **char** a;     // 1 byte  **char** dummy1[1]; // padding 1 byte  **int** b;      // 4 bytes  **short** c;    // 2 bytes  **char** d;     // 1 byte  **char** dummy2[1]; // padding 1 byte  }; |

Padding data

Cách 2: Dùng tiền xử lí #pragma pack([n]) (n = 1, 2, 4, 8, 16)

Ví dụ 3:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | #include <stdio.h>  #include <conio.h>    # pragma pack (1)  **struct** A  {  **char** a;     // 1 byte  **int** b;      // 4 bytes  **short** c;    // 2 bytes  **char** d;     // 1 byte  };  # pragma pack ()    **void** main()  {  **printf**("size = %d", **sizeof**(A));      getch();  } |

Kết quả: size = 8

# Con trỏ

## Con trỏ là gì?

Ví dụ 1: Nhập tên và in ra màn hình tên bạn vừa nhập

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | #include "stdio.h"  #include "conio.h"    **void** main()  {  **char** s[30];  **printf**("\nNhap ten: ");  **gets**(s);  **printf**("\nTen cua ban: %s", s);      getch();  } |

Ở đây, chương trình này có một số hạn chế như sau:

* Mảng kí tự s chứa tối đa được 29 kí tự
* Nếu bạn nhập tên có độ dài nhỏ hơn 29 gây lãng phí bộ nhớ
* Nếu bạn nhập tên có độ dài lớn hơn 29 gây lỗi chương trình

Để khắc phục những hạn chế trên đây, ngôn ngữ C đưa ra khái niệm biến con trỏ và cấp phát động

Vậy biến con trỏ là gì? Cấp phát động là gì?

Khái niệm: Biến con trỏ (Pointer variable) là biến chứa địa chỉ của biến hoặc chứa địa chỉ của ô nhớ.

Khai báo

<Kiểu dữ liệu> \* <Tên biến con trỏ>

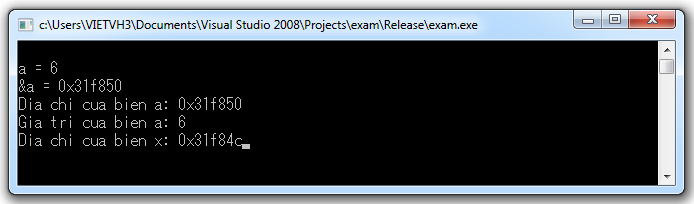
Kiểu dữ liệu: int, char, float, long, struct, … Kiểu dữ liệu nào thì sẽ lưu địa chỉ của biến có kiểu dữ liệu tương ứng

Tên biến con trỏ: đặt tên theo quy tắc đặt tên biến

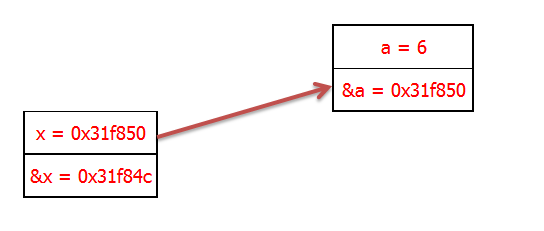
Ví dụ 2:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #include "stdio.h"  #include "conio.h"    **void** main()  {  **int** \*x;  **int** a = 6;      x = &a;  **printf**("\na = %d", a);  **printf**("\n&a = 0x%x", &a);  **printf**("\nDia chi cua bien a: 0x%x", x);  **printf**("\nGia tri cua bien a: %d", \*x);  **printf**("\nDia chi cua bien x: 0x%x", &x);      getch();  } |

Kết quả:

Con trỏ trong ngôn ngữ C

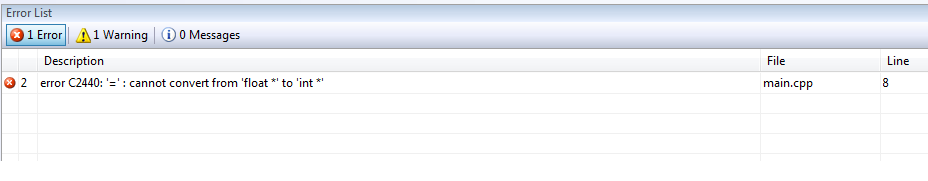
Đoạn chương trình trên khai báo biến con trỏ x kiểu nguyên. Lệnh x = &a, con trỏ x chứa địa chỉ của biến a và \*x = a = 6. Minh họa như hình sau:

Con trỏ ngôn ngữ C

Ví dụ 3:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #include "stdio.h"  #include "conio.h"    **void** main()  {  **int** \*x;  **float** a = 1.6;      x = &a;      getch();  } |

Kết quả:

Con trỏ ngôn ngữ lập trình C

Chương trình biên dịch báo lỗi vì con trỏ kiểu số nguyên không thể trỏ tới biến kiểu số thực.

Chú ý: Con trỏ chỉ có thể chứa địa chỉ của biến có cùng kiểu dữ liệu với con trỏ.

## Phép toán thao tác con trỏ

Có 4 nhóm phép toán liên quan đến con trỏ và địa chỉ là: phép gán, phép tăng giảm địa chỉ, phép truy cập bộ nhớ và so sánh.

### Phép gán

Chỉ nên thực hiện phép gán giữa các con trỏ cùng kiểu dữ liệu. Nếu muốn thực hiện phép gán giữa các con trỏ có kiểu dữ liệu khác nhau cần phải ép kiểu. Tuy nhiên, để thực hiện ép kiểu bạn phải hiểu rất rõ cách lưu địa chỉ của con trỏ.

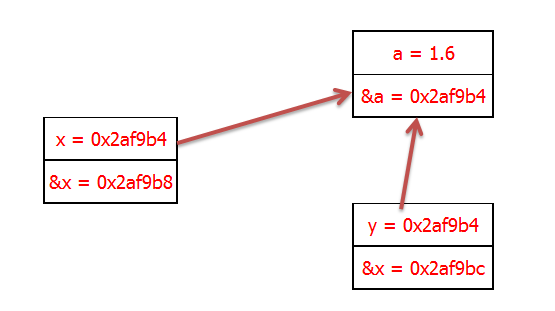
Ví dụ 4:

|  |  |
| --- | --- |
| 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"    **void** main()  {  **float** \*x, \*y;  **float** a = 1.6;      x = &a;      y = x;  **printf**("\na = %f", a);  **printf**("\n\*x = %f", \*x);  **printf**("\n\*y = %f", \*y);    **printf**("\n&a = 0x%x", &a);  **printf**("\nx = 0x%x", x);  **printf**("\ny = 0x%x", y);    **printf**("\n&x = 0x%x", &x);  **printf**("\n&y = 0x%x", &y);        getch();  } |

Kết quả:

Phép gán con trỏ ngôn ngữ lập trình C

Câu lệnh y = x tương đương y = x = &a. Lúc này, biến con trỏ y cũng chứa địa chỉ của biến a hay ta nói con trỏ y trỏ tới biến a. \*x và \*y chính là giá trị của biến a (\*x = \*y = a = 1.6). Hình sau mô tả địa chỉ và giá trị của biến a, biến con trỏ x, y.

Phép gán con trỏ ngôn ngữ lập trình C

Chú ý: Giá trị địa chỉ của các biến là số nguyên (kích thước 4 byte (Windows 32bit, Visual Studio 2008 Professional SP1)).

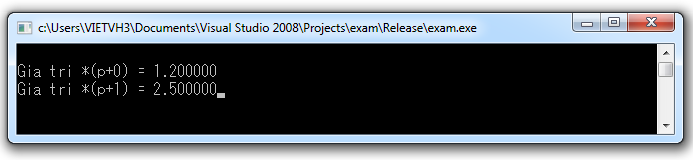
### Phép tăng giảm địa chỉ

Dùng toán tử ++, —

Ví dụ 5:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #include "stdio.h"  #include "conio.h"    **void** main()  {  **float** A[]= {1.2, 2.5, 3.6, 4.8};  **float** \*p;      p = A;  **printf**("\nGia tri \*(p+0) = %f", \*p);      p++;  **printf**("\nGia tri \*(p+1) = %f", \*p);      getch();  } |

Kết quả:

Phép tăng giảm con trỏ ngôn ngữ lập trình C

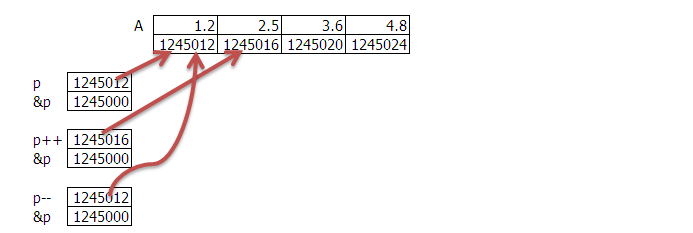
Khi khai báo 1 mảng thì tên mảng A chính là địa chỉ của phần tử đầu tiên của mảng; các phần tử của mảng int sẽ được lưu tại các ô nhớ liền kề nhau có **kích thước 4 byte**.

Câu lệnh gán p = A, con trỏ p trỏ tới phần tử đầu tiên của mảng và \*p = A[0] = 1.2

Câu lệnh p++ tương đương với câu lệnh p = p+1 di chuyển con trỏ p tới **1×4(byte)**(phần tử thứ 2 của mảng A) và \*p = A[1] = 2.5.

***Chú ý:*** Nếu p = p + k (k > 3), chương trình sẽ gây lỗi vì con trỏ p truy cập tới phần tử không xác định.

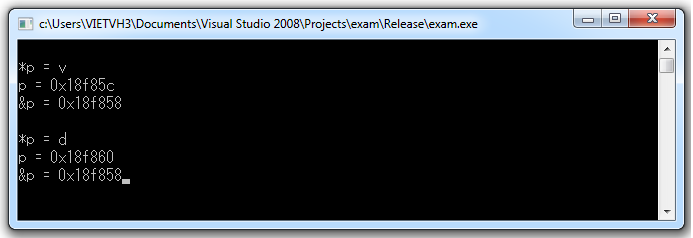
Hình sau minh họa hoạt động tăng giảm của biến con trỏ p. Việc thay đổi giá trị p chỉ làm thay đổi địa chỉ vùng nhớ mà con trỏ p trỏ tới mà không thay đổi địa chỉ của biến con trỏ p.

Mảng và con trỏ

Ví dụ 6:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include "stdio.h"  #include "conio.h"    **void** main()  {  **char** s[]= "vncoding";  **char** \*p;      p = s;  **printf**("\n\*p = %c", \*p);  **printf**("\np = 0x%x", p);  **printf**("\n&p = 0x%x", &p);        p = p + 4;  **printf**("\n\n\*p = %c", \*p);  **printf**("\np = 0x%x", p);  **printf**("\n&p = 0x%x", &p);        getch();  } |

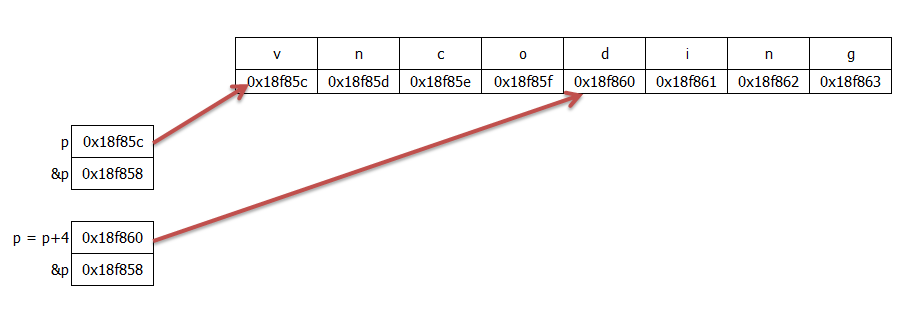
Kết quả:

Phép tăng giảm con trỏ

Như bạn đã biết, khi khai báo 1 mảng thì tên mảng s chính là địa chỉ của phần tử đầu tiên của mảng; các phần tử của mảng char sẽ được lưu tại các ô nhớ liền kề nhau có **kích thước 1 byte**.

Câu lệnh gán p = s, con trỏ p trỏ tới phần tử đầu tiên của mảng và \*p = s[0] = ‘v’

Câu lệnh p = p+4, di chuyển con trỏ p tới **4×1(byte)** tiếp theo (phần tử thứ 5 tính từ phần tử ‘v’) và \*p = s[4] = ‘d’

Phép tăng giảm con trỏ ngôn ngữ C

#### Nguyên tắc truy cập vùng nhớ

Con trỏ kiểu float truy cập 4 byte, kiểu int truy cập 4 byte, kiểu char truy cập 1 byte,..

|  |  |
| --- | --- |
| 1  2  3 | **float** \*a;  **int** \*b;  **char** \*c; |

Giả sử cả 3 con trỏ a,b,c đang trỏ đến vùng nhớ 1500 thì:

\*a sẽ biểu thị vùng nhớ 4 byte liên tiếp từ 1500 – 1503

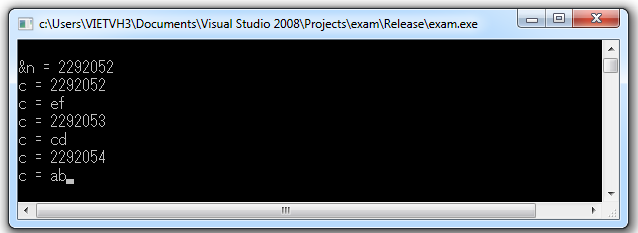
\*b sẽ biểu thị vùng nhớ 4 byte liên tiếp từ 1500 – 1503

\*c sẽ biểu thì vùng nhớ 1 byte từ 1500 – 1501

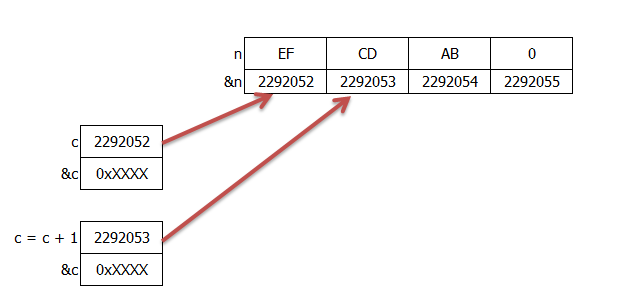
Ví dụ 8:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #include "stdio.h"  #include "conio.h"    **void** main()  {      unsigned **int** n = 0xABCDEF;// hexa      unsigned **char** \*c;      c = (unsigned **char**\*)(&n);  **printf**("\n&n = %d", &n);  **printf**("\nc = %d", c);  **printf**("\nc = %x", \*c);      c++;  **printf**("\nc = %d", c);  **printf**("\nc = %x", \*c);      c++;  **printf**("\nc = %d", c);  **printf**("\nc = %x", \*c);        getch();  } |

Kết quả:

Nguyên tắc truy cập bộ nhớ dùng con trỏ ngôn ngữ C

Giá trị n được lưu trên vùng nhớ như sau:

Nguyên tắc truy cập vùng nhớ dùng con trỏ C

Kiểu số nguyên không dấu có kích thước 4 byte (windows 32bit). Vì con trỏ c là kiểu unsigned char, nên c truy cập 1 byte đầu tiên của n nên \*c = 0xEF và \*(c++) = 0xCD.

### Phép so sánh 2 con trỏ

Cho phép so sánh 2 con trỏ p1,p2 cùng kiểu dữ liệu

p1 < p2 nếu địa chỉ p1 trỏ tới thấp hơn địa chỉ p2 trỏ tới

p1 = p2 nếu địa chỉ p1 trỏ tới bằng địa chỉ p2 trỏ tới

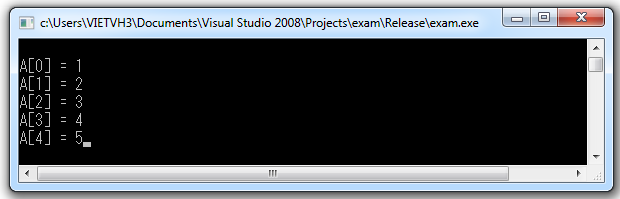
p1 > p2 nếu địa chỉ p1 trỏ tới cao hơn địa chỉ p2 trỏ tới

## Mối liên hệ giữa con trỏ và mảng 1 chiều

Ví dụ 9:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | #include "stdio.h"  #include "conio.h"    **void** main()  {  **int** A[] = {1, 2, 3, 4, 5};  **int** i; //duyet mang  **for**(i = 0; i < 5; i++)  **printf**("\nA[%d] = %d", i, \*(A+i));      getch();  } |

Kết quả:

Mối liên hệ giữa mảng 1 chiều và con trỏ

**Ghi nhớ:** A là địa chỉ của phần tử đầu tiên của mảng A, A+1 : địa chỉ của phần tử thứ 2,…

Giá trị của các phần tử của mảng:

\*(A+0) = A[0] = 1

\*(A+1) = A[1] = 2

…

\*(A+4) = A[4] = 5

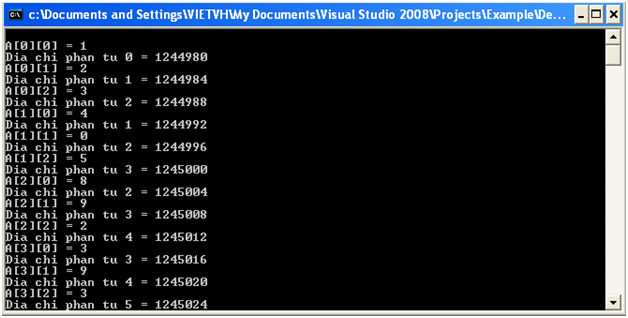
Vậy : Muốn lấy giá trị của phần tử trong mảng 1 chiều ta có thể dùng: **\*(A+i)** , i là chỉ số mảng.

## Mối liên hệ giữa con trỏ và mảng 2 chiều

Ví dụ 10:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #include "stdio.h"  #include "conio.h"  #define N 4  #define M 3    **void** main()  {  **int** A[N][M] = {1, 2, 3,                     4, 0, 5,                     8, 9, 2,                     3, 9, 3};  **int** i, j;  **for**(i = 0; i < N; i++)  **for**(j = 0; j < M; j++)          {  **printf**("\nA[%d][%d] = %d", i, j, \*(\*(A+i)+j));  **printf**("\nDia chi phan tu %d = %d", i+j, \*(A+i)+j);          }      getch();  } |

Kết quả:

Mối liên hệ giữa con trỏ và mảng 2 chiều

Để lấy giá trị của các phần tử mảng 2 chiều ta dùng biểu thức sau: \*(\*(A+i)+j) và để lấy địa chỉ của phần tử mảng 2 chiều ta dùng biểu thức sau: \*(A+i)+j. (trong đó i,j là chỉ số hàng và cột)

# Sư khác nhau giữa ++x và x++ trong C/C++?

Cả 2 toán tử ++x và x++ dùng để tăng giá trị của x lên 1 đơn vị. Tuy nhiên, chúng có 1 điểm khác nhau cơ bản:

* Toán tử tăng trước ++x: tăng giá trị x trước khi thực hiện các phép toán khác trong cùng 1 câu lệnh

Ví dụ 1:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | #include <stdio.h>  #include <conio.h>    **void** main()  {  **int** x = 5;  **int** y = 0;      y += ++x + 5;  **printf**("\n x = %d, y = %d", x, y);      getch();  } |

Kết quả: x = 6, y = 11

* Toán tử tăng sau x++: tăng giá trị x sau khi thực hiện các phép toán khác trong cùng 1 câu lệnh

Ví dụ 2:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | #include <stdio.h>  #include <conio.h>    **void** main()  {  **int** x = 5;  **int** y = 0;      y += x++ + 5;  **printf**("\n x = %d, y = %d", x, y);      getch();  } |

Kết quả: x = 6, y = 10

# Vai trò của từ khóa default và break trong switch case?

Cấu trúc câu lệnh switch case

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **switch**(expression)  {  **case** N1:      // Code  **break**;  **case** N2:      // Code  **break**;  **case** N3:      // Code  **break**;  ...  **default**:      // Code  **break**;  } |

Câu lệnh switch case đánh giá biểu thức expression. Chú ý: Biểu thức expression là biểu thức trả về số nguyên. Nếu expression = N1 (N2, N3,…) câu lệnh nằm trong case N1 (N2, N3,…) được thực hiện,… Nếu expression không bằng bất kì giá trị nào, câu lệnh nằm trong từ khóa default được thực hiện.

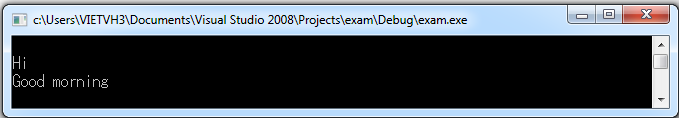
Vậy, từ khóa default đóng vai trò xử lý những case ngoại lệ khi biểu thức điều kiện không thỏa mãn bất cứ case nào.

Khi câu lệnh trong mỗi case được thực hiên xong, lệnh break giúp thoát switch case. Điều này giúp các câu lệnh của các case bên dưới không được thực hiện.

Ví du 1: Không có lệnh break trong switch case

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | #include <stdio.h>  #include <conio.h>    **void** main()  {  **int** sel = 1;  **switch**(sel)      {  **case** 0:  **printf**("\nHello");  **case** 1:  **printf**("\nHi");  **case** 2:  **printf**("\nGood morning");      }      getch();  } |

Kết quả:

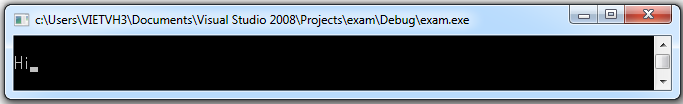
No break in switch case

Khi sel = 1, “Hi” được in ra màn hình. Do không có lệnh break nên “Good morning” được thực hiện.

Ví dụ 2: Lệnh break trong switch case

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | #include <stdio.h>  #include <conio.h>      **void** main()  {  **int** sel = 1;  **switch**(sel)      {  **case** 0:  **printf**("\nHello");  **break**;  **case** 1:  **printf**("\nHi");  **break**;  **case** 2:  **printf**("\nGood morning");  **break**;      }      getch();  } |

Kết quả:

Lệnh break in switch case

Sau khi “Hi” được in ra màn hình. Lệnh break thoát khỏi lệnh switch case, đảm bảo các lệnh của các case bên dưới không được thực hiện. Ở case 2, lệnh break không cần thiết vì sau case 2 không còn case nào nữa. Để đảm bảo tính đồng nhất so với các case khác nên break được add vào tất cả các case.

# Lệnh switch case và lệnh if else?

Như mình đã giới  thiệu tới các lệnh [**switch case**](http://vncoding.net/2015/10/25/lenh-switch-case/) và [**if else**](http://vncoding.net/2015/10/24/lenh-ifelse/) trong bài viết [**ngôn ngữ lập trình C**](http://vncoding.net/ngon-ngu-lap-trinh-cc/).

Lệnh **switch case** và **if else** được dùng khi cần so sánh biểu thức điều kiện và đưa ra quyết định. Tuy nhiên, **switch case** và **if else** lại có đặc điểm riêng được dùng trong từng bài toán khác nhau. Dưới đây là một số ví dụ minh họa giúp các bạn lựa chọn sử dụng **switch case** hoặc **if else** một cách phù hợp.

**Lệnh switch case**

* Sử dụng lệnh switch case dễ nhìn, dễ maintain.
* Chỉ cần so sánh biểu thức điều kiện 1 lần.
* Khi biểu nhiều case thực hiện lệnh như nhau. Chúng ta có thể viết case liên tục và không sử dụng break.

Ví dụ 1: Nhập vào ngày trong tuần, in ra màn hình là ngày làm việc ( thứ 2 – 6) hay ngày nghỉ (thứ 7, chủ nhật)

Cách 1: Viết bằng lệnh switch case

|  |  |
| --- | --- |
| 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  28  29  30  31  32  33  34  35  36 | #include <stdio.h>  #include <conio.h>    **enum** day  {      MONDAY=2,      TUESDAY,      WEDNESDAY,      THURSDAY,      FRIDAY,      SATURDAY,      SUNDAY  };    **void** main()  {  **int** dayOfWeek = MONDAY;  **switch** (dayOfWeek)      {  **case** MONDAY:  **case** TUESDAY:  **case** WEDNESDAY:  **case** THURSDAY:  **case** FRIDAY:  **printf**("\nWeekday");  **break**;  **case** SATURDAY:  **case** SUNDAY:  **printf**("\nWeekend");  **break**;  **default**:  **printf**("\nInvalid");  **break**;      }      getch();  } |

Cách 2: Viết bằng lệnh if else

|  |  |
| --- | --- |
| 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  28  29  30  31  32 | #include <stdio.h>  #include <conio.h>    **enum** day  {      MONDAY=2,      TUESDAY,      WEDNESDAY,      THURSDAY,      FRIDAY,      SATURDAY,      SUNDAY  };    **void** main()  {  **int** dayOfWeek = MONDAY;    **if**(dayOfWeek == MONDAY ||          dayOfWeek == TUESDAY ||          dayOfWeek == WEDNESDAY ||          dayOfWeek == THURSDAY ||          dayOfWeek == FRIDAY)  **printf**("\nWeekday");  **else** **if** (dayOfWeek == SATURDAY ||          dayOfWeek == SUNDAY)  **printf**("\nWeekend");  **else**  **printf**("\nInvalid");        getch();  } |

Qua đây, các bạn có thể thấy rằng sử dụng lệnh switch case dễ nhìn và có performance cao hơn lệnh **if else**. Trong lệnh switch case, chỉ cần so sánh biểu thức **dayOfWeek**1 lần và goto tới **case** tương ứng. Trong lệnh if else, biểu thức **dayOfWeek** được so sánh với với MONDAY, TUESDAY,…, FRIDAY trong lệnh **if**, nếu không thỏa mãn lại tiếp tục được so sánh với SATURDAY, SUNDAY trong lệnh **else if.**

**Lệnh If else**

* Lệnh **switch case** có hạn chế là biểu thức điều kiện phải là số nguyên. Đối với lệnh **if else**, biểu thức điều kiện là true hoặc false. Do vậy, một số trường hợp, bắt buộc phải sử dụng lệnh **if else** thay vì **switch case**.

Ví dụ 2: Quy đổi thang điểm 10 sang thang điểm ABCD

Score < 4: điểm F

4  <= Score < 6: điểm D

6 <= Score < 7: điểm C

7 <= Score <= 8: điểm B

Score >= 8.5: điểm A

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #include <stdio.h>  #include <conio.h>    **void** main()  {  **float** score = 7.5;    **if**(score < 4 && score >= 0)  **printf**("\nDiem F");  **else** **if**(score >= 4 && score < 6) **printf**("\nDiem D"); **else** **if**(score >= 6 && score < 7) **printf**("\nDiem C"); **else** **if**(score >= 7 && score <= 8) **printf**("\nDiem B"); **else** **if**(score >= 8.5 && score <= 10)  **printf**("\nDiem A");  **else**  **printf**("\nInvalid");      getch();  } |

Kết luận:

* Nên sử dụng switch case trong bài toán mul-ti choice, biểu thức điều kiện tính toán phức tạp nhưng phải có giá trị nguyên.
* Nên sử dụng if else trong bài toán ít trường hợp, các biểu thức điều kiện đơn giản

# Từ khóa “auto”

Từ khóa ***auto***xuất phát từ tổ tiên của C++, đó là ngôn ngữ lập trình C. Từ ***auto***vẫn giữ nguyên ý nghĩa ban đầu của nó, nhưng C++ đã thêm một số thứ hay ho vào nó. Hãy bắt đầu với các đặc tính ban đầu cũng như hiệu ứng của nó trong chương trình.

Tất cả các biến trong code đều thuộc một trong hai loại:

* các biến tự động (automatic variables), tạo và hủy, đôi khi lặp đi lặp lại, và được thực hiện một cách tự động khi chạy chương trình
* các biến tĩnh (static variables), tồn tại liên tục trong quá trình thực hiện toàn bộ chương trình

Các ngôn ngữ lập trình C và C++ giả định rằng tất cả các biến được mặc định là tự động trừ khi chúng được khai báo là biến tĩnh một cách tường minh. Điều thú vị là từ khóa ***auto***vẫn thường được sử dụng trong các phiên bản cũ của ngôn ngữ C để đánh dấu các biến tự động một cách tường minh.

Từ khoá được bảo toàn để đảm bảo khả năng tương thích ngược và vẫn được công nhận bởi các trình biên dịch C++, mặc dù ngày nay nó rất hiếm khi được sử dụng.

Hãy xem chương trình sau →

C++



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #include <iostream>    using namespace std;    void fun(void) {      auto int var = 99;      cout << "var = " << ++var << endl;  }    int main(void) {      for(int i = 0; i < 5; i++) {          fun();      }        return 0;  } |

Biến *var* được khai báo bên trong hàm *fun* và là biến tự động. Chúng tôi đã sử dụng từ khóa ***auto***, nhưng hành vi của chương trình sẽ vẫn như cũ nếu bạn xóa từ khoá ***auto***đi (bạn nên code thử và kiểm chứng). Biến được tạo ra mỗi khi hàm *fun* được gọi và biến được hủy mỗi lần hàm kết thúc.

Điều này có nghĩa là, bất kể số lượng lời gọi, hàm *fun*sẽ luôn luôn cho ra kết quả giống nhau như sau →

C++



|  |  |
| --- | --- |
| 1  2  3  4  5 | var = 100  var = 100  var = 100  var = 100  var = 100 |

Sự khác nhau giữa kích thước của biến con trỏ int\* và biến con trỏ char\*

Kích thước của một biến con trỏ phụ thuộc vào kiến trúc và hệ điều hành của máy tính mà bạn đang sử dụng. Tuy nhiên, trong hầu hết các hệ thống, kích thước của một con trỏ sẽ là cố định, không phụ thuộc vào kiểu dữ liệu mà con trỏ đang trỏ tới.

Ví dụ, trong một hệ thống 32-bit, kích thước của một con trỏ sẽ là 4 byte, bất kể con trỏ đang trỏ tới kiểu dữ liệu nào (int, char, hay bất kỳ kiểu dữ liệu nào khác). Trong hệ thống 64-bit, kích thước của một con trỏ thường là 8 byte.

Do đó, không có sự khác biệt về kích thước giữa biến con trỏ int\* và biến con trỏ char\*. Cả hai đều có kích thước tương tự và phụ thuộc vào kiến trúc và hệ điều hành của máy tính.

Con trỏ gần(near pointer) là gì?

Con trỏ gần (near pointer) là một loại con trỏ được sử dụng trong một số kiến trúc vi xử lý cổ điển, chẳng hạn như kiến trúc 16-bit của các hệ thống như DOS. Trong các hệ thống này, địa chỉ bộ nhớ được biểu diễn bằng 16-bit, nghĩa là chỉ có thể truy cập tối đa 64KB bộ nhớ.

Con trỏ gần chỉ có thể trỏ đến các vùng nhớ trong cùng đoạn 64KB đó. Nó không thể trỏ đến các vùng nhớ ở các đoạn khác. Điều này giới hạn khả năng truy cập và quản lý bộ nhớ của các hệ thống sử dụng con trỏ gần.

Trong các kiến trúc hiện đại, như kiến trúc 32-bit và 64-bit, con trỏ gần không còn được sử dụng, vì địa chỉ bộ nhớ được biểu diễn bằng 32-bit hoặc 64-bit, cho phép truy cập và quản lý bộ nhớ lớn hơn nhiều so với 64KB. Thay vào đó, các loại con trỏ như con trỏ chính (far pointer) hoặc con trỏ không gian (void pointer) được sử dụng để truy cập và quản lý bộ nhớ trong các kiến trúc hiện đại.

Con trỏ xa (far pointer) là gì?

Con trỏ xa (far pointer) là một loại con trỏ được sử dụng trong các kiến trúc vi xử lý cổ điển, như kiến trúc 16-bit của các hệ thống như DOS. Trong các hệ thống này, địa chỉ bộ nhớ được biểu diễn bằng 16-bit, và do đó chỉ có thể truy cập tối đa 64KB bộ nhớ.

Tuy nhiên, để vượt qua giới hạn này và truy cập các vùng nhớ ở các địa chỉ bộ nhớ khác, con trỏ xa được sử dụng. Con trỏ xa bao gồm một cặp giá trị, gồm một giá trị offset và một giá trị segment. Offset đại diện cho vị trí của dữ liệu trong đoạn 64KB, trong khi segment đại diện cho đoạn 64KB nơi dữ liệu được lưu trữ.

Khi sử dụng con trỏ xa, ta cần kết hợp giá trị offset và segment để xác định địa chỉ thực sự của dữ liệu. Điều này cho phép truy cập và quản lý bộ nhớ nằm ngoài 64KB giới hạn của con trỏ gần.

Tuy nhiên, trong các kiến trúc hiện đại như kiến trúc 32-bit và 64-bit, con trỏ xa không còn được sử dụng nữa, vì địa chỉ bộ nhớ được biểu diễn bằng 32-bit hoặc 64-bit, cho phép truy cập và quản lý bộ nhớ lớn hơn nhiều so với 64KB. Thay vào đó, các loại con trỏ như con trỏ chính (near pointer) hoặc con trỏ không gian (void pointer) được sử dụng để truy cập và quản lý bộ nhớ trong các kiến trúc hiện đại

Sử dụng hàm nào an toàn hơn trong 2 hàm fgets() và gets()?

Trong số hai hàm fgets() và gets(), hàm fgets() được coi là an toàn hơn. Đây là một số lý do:

1. fgets() cho phép bạn xác định số lượng ký tự tối đa mà hàm sẽ đọc từ đầu vào. Điều này giúp tránh việc đọc quá giới hạn bộ nhớ được cấp phát cho biến đệm, như trường hợp sử dụng gets() có thể gây ra.

2. Hàm fgets() cũng cho phép bạn chỉ định luồng đầu vào mà hàm sẽ đọc từ đó. Điều này giúp tránh việc đọc từ luồng đầu vào không mong muốn, như trường hợp sử dụng gets() có thể gây ra.

3. Hàm fgets() cũng bao gồm ký tự newline (\n) trong chuỗi kết quả, trong khi gets() không bao gồm. Điều này có thể hữu ích khi bạn muốn đọc dữ liệu theo dòng từ đầu vào.

4. Hàm gets() không kiểm tra độ dài của chuỗi đầu vào và không thể xác định được kích thước tối đa của chuỗi. Điều này có thể dẫn đến tràn bộ nhớ và gây ra lỗi không mong muốn. Trong khi đó, hàm fgets() cho phép bạn xác định số lượng ký tự tối đa để tránh tràn bộ nhớ.

Do những lý do trên, nên sử dụng hàm fgets() thay vì gets() để đọc dữ liệu từ đầu vào. Nó giúp tránh các vấn đề bảo mật liên quan đến tràn bộ nhớ và cung cấp kiểm soát hơn về việc đọc dữ liệu từ nguồn.

Những ưu điểm khi sử dụng enum thay vì sử dụng #define

Khi sử dụng enum thay vì sử dụng #define trong ngôn ngữ lập trình C/C++, có một số ưu điểm sau:

1. Tính rõ ràng và dễ đọc: Enum cho phép bạn định nghĩa các hằng số với các tên có ý nghĩa, giúp mã nguồn trở nên dễ đọc và dễ hiểu hơn so với #define.

2. Kiểm tra kiểu tĩnh: Enum cung cấp kiểm tra kiểu tĩnh tự động. Điều này giúp tránh nhầm lẫn và lỗi không mong muốn khi sử dụng các hằng số không đúng kiểu dữ liệu.

3. Tính tương thích và mở rộng: Enum có thể dễ dàng mở rộng bằng cách thêm các giá trị mới vào danh sách enum. Điều này giúp giữ cho mã nguồn linh hoạt và dễ dàng bảo trì khi có các yêu cầu thay đổi trong tương lai.

4. Debugging dễ dàng: Khi sử dụng enum, các giá trị được hiển thị trong bộ gỡ lỗi (debugger) thay vì các giá trị mã hóa được định nghĩa bằng #define. Điều này giúp dễ dàng theo dõi và debug các giá trị trong mã nguồn.

5. Tính chặt chẽ và an toàn: Enum đảm bảo rằng các giá trị được định nghĩa trong enum là duy nhất, và không thể gán một giá trị không hợp lệ cho biến enum. Điều này giúp tránh các lỗi logic và tăng tính an toàn của mã nguồn.

Tóm lại, việc sử dụng enum thay vì #define trong C/C++ mang lại nhiều lợi ích về tính rõ ràng, kiểm tra kiểu tĩnh, tính mở rộng, debugging dễ dàng và tính an toàn.

Từ khóaTừ khóa `#pragma` là một từ khóa đặc biệt trong ngôn ngữ lập trình C/C++ được sử dụng để điều chỉnh hoặc chỉ định các hành vi hoặc thuộc tính cụ thể của trình biên dịch.

`#pragma` có thể được sử dụng để thực hiện các hành động như:

1. Điều chỉnh cấu hình trình biên dịch: `#pragma` có thể được sử dụng để sửa đổi cấu hình hoặc chế độ biên dịch của trình biên dịch. Ví dụ, `#pragma warning` trong C/C++ cho phép bạn điều chỉnh cách trình biên dịch xử lý cảnh báo.

2. Điều chỉnh việc xử lý mã nguồn: `#pragma` có thể được sử dụng để điều chỉnh cách trình biên dịch xử lý mã nguồn. Ví dụ, `#pragma inline` trong C/C++ cho phép bạn chỉ định việc biên dịch hàm như một hàm nội tuyến (inline function).

3. Điều chỉnh việc tạo các mã máy tùy chỉnh: `#pragma` có thể được sử dụng để tùy chỉnh việc tạo mã máy hoặc tối ưu hóa mã. Ví dụ, `#pragma omp` trong C/C++ cho phép bạn điều chỉnh việc tạo mã song song sử dụng OpenMP.

Tuy nhiên, giá trị cụ thể và cách sử dụng của `#pragma` có thể khác nhau tùy thuộc vào trình biên dịch và tiêu chuẩn ngôn ngữ C/C++ mà bạn đang sử dụng. Do đó, nên kiểm tra tài liệu của trình biên dịch hoặc tiêu chuẩn ngôn ngữ để biết cách sử dụng `#pragma` trong ngữ cảnh cụ thể.

#prama là gì?

Dưới đây là một số ví dụ về cách sử dụng từ khóa `#pragma` trong C/C++:

1. Sử dụng `#pragma once` để ngăn chặn việc đệm mã nguồn:

c

#**pragma** once

// Mã nguồn của tệp header

Đoạn mã trên sẽ đảm bảo rằng tệp header chỉ được đệm một lần trong quá trình biên dịch.

2. Sử dụng `#pragma warning` để điều chỉnh cách trình biên dịch xử lý cảnh báo:

c

#pragma warning(disable: 4996)

// Mã nguồn có cảnh báo 4996 được tắt

Đoạn mã trên sẽ tắt cảnh báo 4996 của trình biên dịch.

3. Sử dụng `#pragma omp` để điều chỉnh việc tạo mã song song sử dụng OpenMP:

c

#pragma omp parallel for

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

// Các phần tử của vòng lặp được thực hiện song song

}

Đoạn mã trên sẽ sử dụng OpenMP để thực hiện vòng lặp for song song trên nhiều luồng.

4. Sử dụng `#pragma message` để hiển thị thông điệp tùy chỉnh trong quá trình biên dịch:

c

#**pragma** message("Xin chào! Đây là thông điệp tùy chỉnh.")

// Mã nguồn tiếp theo

Đoạn mã trên sẽ hiển thị "Xin chào! Đây là thông điệp tùy chỉnh." trong quá trình biên dịch.

Lưu ý rằng cú pháp và chức năng cụ thể của `#pragma` có thể khác nhau tùy thuộc vào trình biên dịch và tiêu chuẩn ngôn ngữ C/C++ mà bạn đang sử dụng.

Viết con trỏ tương ứng với giá trị mảng a[i][j][k][m]

Để viết con trỏ tương ứng với giá trị mảng a[i][j][k][m], bạn có thể sử dụng con trỏ có kiểu tương ứng với kiểu dữ liệu của mảng. Dưới đây là một ví dụ về cách viết con trỏ tương ứng:

c

**int** main() {

**int** a[10][20][30][40]; // Mảng a có kích thước 10x20x30x40

// Khai báo con **tr**ỏ p tương ứng với mảng a

**int** (\*p)[20][30][40] = a;

// Sử dụng con **tr**ỏ p để truy cập các phần tử của mảng a

**for** (**int** i = 0; i < 10; i++) {

**for** (**int** j = 0; j < 20; j++) {

**for** (**int** k = 0; k < 30; k++) {

**for** (**int** m = 0; m < 40; m++) {

// Gán giá **tr**ị cho phần tử a[i][j][k][m] thông qua con **tr**ỏ p

(\*p)[j][k][m] = i + j + k + m;

}

}

}

p++; // Di chuyển con **tr**ỏ p sang phần tử tiếp theo của mảng a

}

// In giá **tr**ị của mảng a

**for** (**int** i = 0; i < 10; i++) {

**for** (**int** j = 0; j < 20; j++) {

**for** (**int** k = 0; k < 30; k++) {

**for** (**int** m = 0; m < 40; m++) {

// In giá **tr**ị của phần tử a[i][j][k][m]

**printf**("%d ", a[i][j][k][m]);

}

}

}

}

**return** 0;

}

Trong ví dụ trên, con trỏ `p` được khai báo với kiểu `int (\*p)[20][30][40]`, tương ứng với mảng `a`. Bạn có thể sử dụng con trỏ `p` để truy cập và gán giá trị cho các phần tử của mảng `a`.

Sự khác nhau giữa const và volatite

`const` và `volatile` là hai từ khóa quan trọng trong ngôn ngữ lập trình C/C++ để chỉ định thuộc tính của biến.

1. `const`: Từ khóa `const` được sử dụng để khai báo một biến không thể thay đổi giá trị sau khi được gán một lần. Điều này có nghĩa là bạn không thể thay đổi giá trị của biến `const` trong quá trình thực thi chương trình. Ví dụ:

c

const int x = 5;

x = 10; // Lỗi! Không thể thay đổi giá trị của biến const

2. `volatile`: Từ khóa `volatile` được sử dụng để khai báo một biến có thể thay đổi giá trị bất ngờ trong quá trình thực thi chương trình. Điều này đảm bảo rằng trình biên dịch không tối ưu hóa các hoạt động liên quan đến biến `volatile`. Ví dụ:

c

**volatile** int x = 5;

x = 10; // Có thể thay đổi giá trị của biến volatile bất kỳ lúc nào

Sự khác nhau giữa `const` và `volatile`:

- `const` chỉ định rằng giá trị của biến không thể thay đổi sau khi được gán, trong khi `volatile` chỉ định rằng giá trị của biến có thể thay đổi bất ngờ trong quá trình thực thi chương trình.

- `const` đảm bảo tính không thay đổi của giá trị, trong khi `volatile` đảm bảo tính nhạy bén của giá trị.

- `const` được sử dụng để bảo vệ giá trị khỏi việc thay đổi không mong muốn, trong khi `volatile` được sử dụng để đảm bảo sự tương thích với các sự kiện ngoại vi hoặc các hoạt động đồng thời.

Tóm lại, `const` và `volatile` là hai từ khóa có vai trò quan trọng trong việc chỉ định thuộc tính của biến trong ngôn ngữ lập trình C/C++. `const` chỉ định tính không thay đổi của giá trị, trong khi `volatile` chỉ định tính nhạy bén của giá trị và sự tương thích với các sự kiện ngoại vi.

Tại sao nên gán NULL cho con trỏ sau khi giải phóng vùng nhớ?

Khi giải phóng vùng nhớ mà một con trỏ đang trỏ tới, nên gán giá trị NULL cho con trỏ đó để tránh xảy ra lỗi truy cập con trỏ không hợp lệ sau này. Dưới đây là một số lý do tại sao nên gán NULL cho con trỏ sau khi giải phóng vùng nhớ:

1. Tránh lỗi truy cập con trỏ không hợp lệ: Nếu không gán NULL cho con trỏ sau khi giải phóng vùng nhớ, con trỏ sẽ vẫn giữ địa chỉ của vùng nhớ đã được giải phóng. Nếu bạn cố gắng truy cập vào con trỏ sau khi vùng nhớ đã được giải phóng, sẽ xảy ra lỗi truy cập con trỏ không hợp lệ. Bằng cách gán NULL cho con trỏ, bạn có thể dễ dàng kiểm tra xem con trỏ có trỏ tới vùng nhớ hợp lệ hay không trước khi sử dụng nó.

2. Đảm bảo an toàn trong việc kiểm tra con trỏ: Khi kiểm tra giá trị của con trỏ, nếu con trỏ chưa được gán NULL sau khi giải phóng vùng nhớ, giá trị của con trỏ có thể là một giá trị ngẫu nhiên hoặc giá trị của vùng nhớ khác. Điều này có thể dẫn đến việc kiểm tra con trỏ không chính xác và gây ra lỗi trong chương trình của bạn. Bằng cách gán NULL cho con trỏ sau khi giải phóng vùng nhớ, bạn đảm bảo rằng con trỏ không chứa giá trị ngẫu nhiên và có thể kiểm tra một cách an toàn.

3. Giúp phát hiện lỗi gọi lại con trỏ: Nếu bạn cố gắng sử dụng con trỏ sau khi giải phóng vùng nhớ mà không gán NULL cho nó, chương trình của bạn có thể tiếp tục thực hiện các thao tác trên vùng nhớ đã được giải phóng. Điều này có thể gây ra lỗi nghiêm trọng và khó xác định. Bằng cách gán NULL cho con trỏ sau khi giải phóng vùng nhớ, nếu bạn cố gắng sử dụng con trỏ đã được giải phóng, bạn sẽ nhận được một cảnh báo hoặc lỗi ngay lập tức, giúp bạn phát hiện và sửa lỗi gọi lại con trỏ.

Tóm lại, việc gán NULL cho con trỏ sau khi giải phóng vùng nhớ là một thói quen tốt để tránh lỗi truy cập con trỏ không hợp lệ và đảm bảo an toàn trong việc kiểm tra con trỏ.

**Tổng quan về RTOS**

Series về RTOS giúp các bạn mới tiếp cận có thể hiểu và lập trình được.

Link đăng ký khóa học: <https://deviot.vn/chi-tiet-san-pham/lap-trinh-stm32-va-rtos.31078516>

**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:**

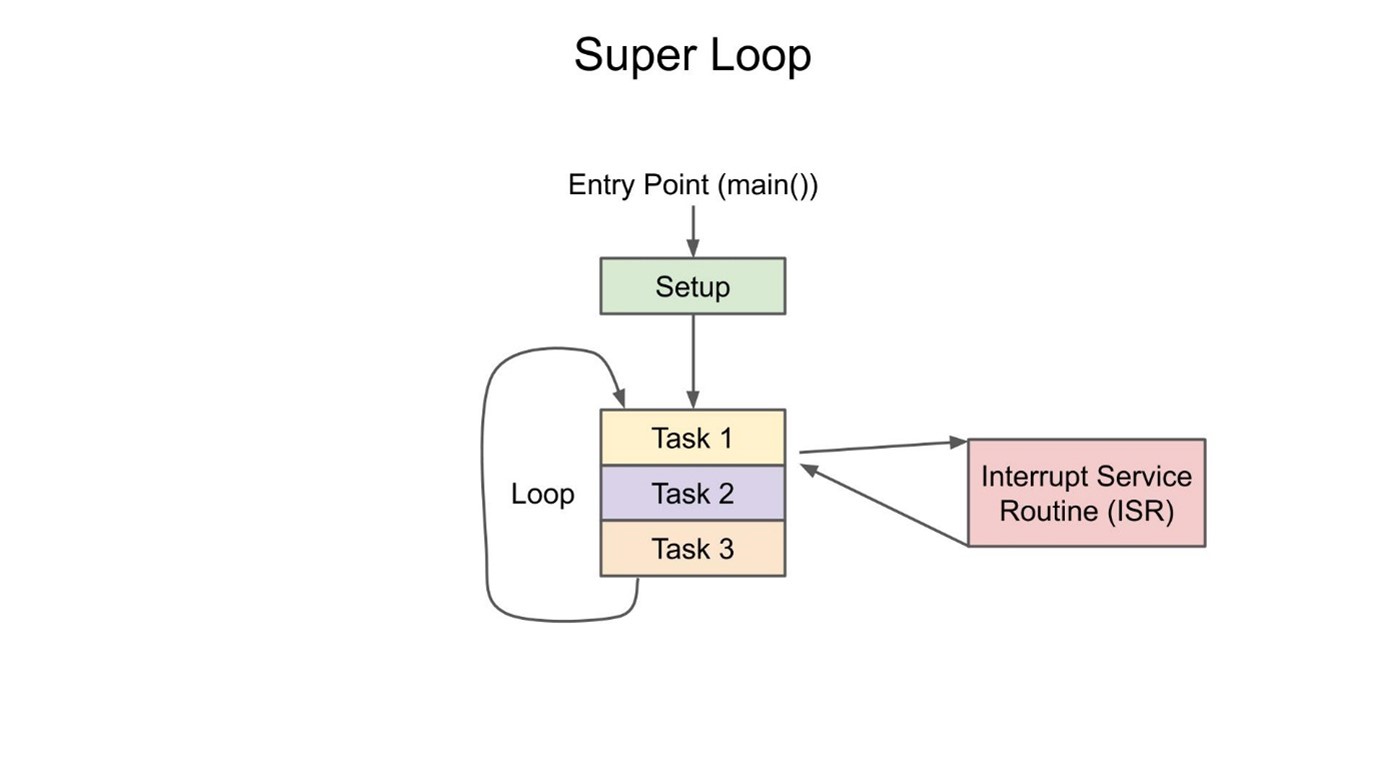
·       Giúp chương trình của bạn dễ quản lý và phát triển vì nó giúp phân “chia” 1 vấn đề phức tạp thành các phần nhỏ hơn “để trị”.

·       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:**

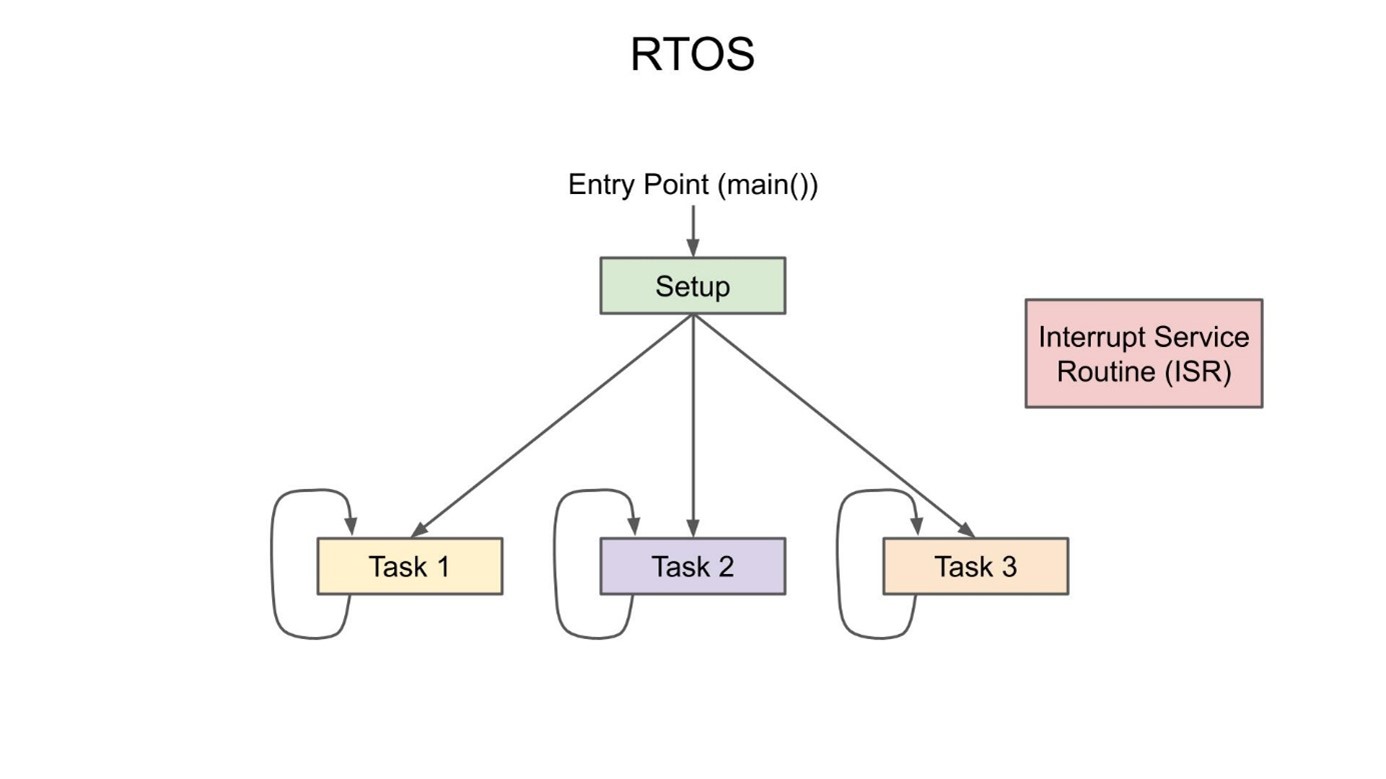
Nếu các bạn đã từng viết ứng dụng về nhúng thì chắc hẳn đã quá quen thuộc với kiến trúc của “siêu vòng lặp”. Ở đây, chúng không sử dụng hệ điều hành và cấu trúc khá đơn giản. Sau khi khai báo biến, thư viện, … thì trong hàm main() sau khi đã khởi tạo xong thì thì chúng sẽ thực hiện nhiều tác vụ định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

**RTOS Phần 2: TASK & SCHEDULE**

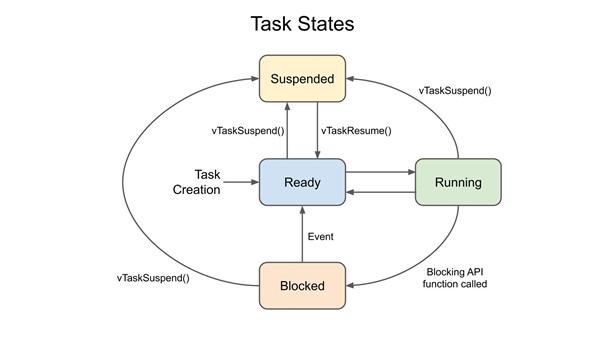
Ở phần 1 ta đã có nhắc đến khái niệm về task, ở phần này ta sẽ đi sâu hơn về task cũng như các phương pháp lập lịch trong RTOS.

Link đăng ký khóa học: <https://deviot.vn/chi-tiet-san-pham/lap-trinh-stm32-va-rtos.31078516>

TỔNG QUAN:

Vì thời gian chuyển trạng thái giữa các task không đáng kể nên người dùng cảm thấy các task như chạy cùng lúc, nhưng thực tế thì tại 1 thời điểm chỉ có duy nhất 1 task running (đối với single core). và các task còn lại thì sẽ ở trạng thái khác, cùng tìm hiểu xem các trạng thái đó là gì nhé..

TASK STATE:



Trong Rtos thì task có 4 trạng thái:

·       **Ready**: Task đã sẵn sàng để có thể thực thi nhưng chưa được thực thi do có các task khác với độ ưu tiên ngang bằng hoặc hơn đang chạy (tương tự như đối với ngắt).

·       **Running**: Task đang thực thi.

·       **Blocked**: Task đang chờ 1 sự kiện nào đó xảy ra, sự kiện này có thể là khoảng thời gian hoặc 1 sự kiện nào đó từ task khác.

·       **Suspended**: Task ở trạng thái treo, về cơ bản thì trạng thái này cũng tương tự như Blocked. Nhưng điểm khác nhau là “cách” chuyển từ trạng thái hiện tại sang Ready State. Chỉ khi gọi hàm vTaskResume() thì task bị treo mới được chuyển sang trạng thái Ready để có thể thực thi.

TASK STRUCTURE:

Void MyTask1(**void \*param**)

{

           //các lệnh khởi tạo.

           While(**1**)

           {

                       //các lệnh thực thi.

           }

}

Khi xây dựng 1 task ta cần chú ý, mỗi task nên là 1 “siêu vòng lặp” vì nếu không chip của bạn sẽ bị reset khi task chạy xong các lệnh của nó. Để tránh điều này ta cần Delete task đó trước khi nó kết thúc bằng hàm: vTaskDelete ();

TASK PROPERTIES:

Trong FreeRtos mỗi Task sẽ được gán:

·       1 mức ưu tiên (priority): tùy vào thư viện mà rank của mức độ ưu tiên có thể được định nghĩa khác nhau (càng thấp càng được ưu tiên hoặc càng cao càng được ưu tiên).

·       1 “cái tên”: chủ yếu được sử dụng để debug.

·       Size: Kích thước ngăn xếp của task.

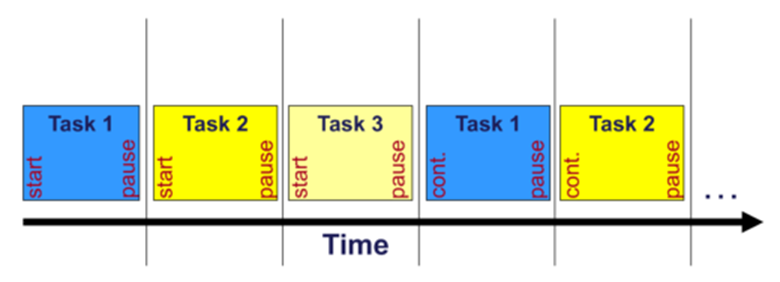
·       Tham số truyền vào.

·       1 con trỏ đại diện cho task (xTaskHandle): thường được sử dụng để điều khiển task từ 1 task khác.

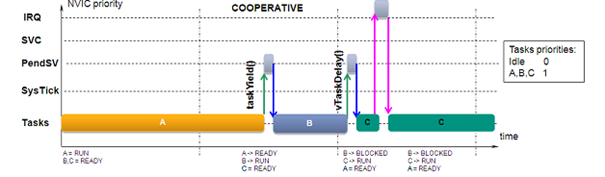
TASK SCHEDULING:

Để lập lịch cho Task, có 3 kỹ thuật được áp dụng:

·       **Round Robin Scheduling:** Mỗi task được chia cho một khe thời gian cố định, nếu trong khoảng thời gian được chia đó mà task chưa thực hiện xong thì sẽ bị tạm dừng, chờ đến lượt tiếp theo để thực hiện tiếp công việc sau khi hệ thống xử lý hết một lượt các task.



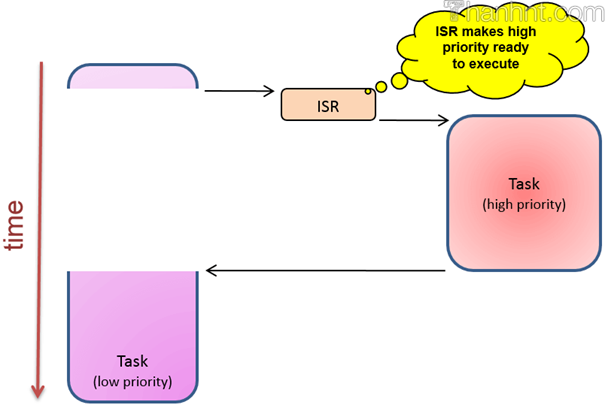
·       **Co-operative scheduling:** Mỗi task được thực thi đến khi kết thúc quá trình thì task tiếp theo mới được thực thi.



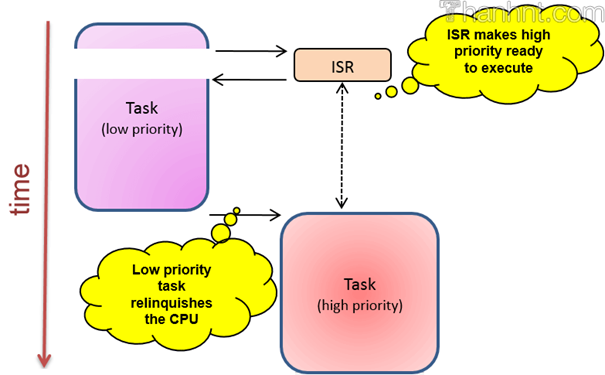
·       **Preemptive Scheduling:** Phương pháp này ưu tiên phân bổ thời gian cho các task có mức ưu tiên cao hơn. Mỗi task được gán 1 mức ưu tiên duy nhất. Có thể có 256 mưc ưu tiên trong hệ thống, và có thể có nhiều task có cùng mức ưu tiên.



o  **Preemptive:** Các task có mức ưu tiên cao nhất luôn được kiểm soát bởi CPU, khi phát sinh ISR thì hệ thống sẽ tạm dừng task đang thực thi, hoàn thành ISR sau đó hệ thống thực thi task có mức ưu tiên cao nhất tại thời điểm đó. Sau đó hệ thống mới tiến hành nối lại các task đang bị gián đoạn. Ở chế độ preemptive, hệ thống có thể đáp ứng các công việc khẩn cấp một cách nhanh chóng. Đa số các hệ thống thực tế đang chạy ở chế độ này.



o  **Non-preemptive:** Ở chế độ non-preemptive thì các task được chạy cho đến khi nó hoàn tất. Khi phát sinh ISR thì hệ thống sẽ tạm dừng task đang thực thi và hoàn thành ISR , sau khi hoàn thành ISR thì hệ thống sẽ quay lại thực thi nốt phần việc còn lại của task bị gián đoạn. Task có mức ưu tiên cao hơn sẽ giành quyền kiểm soát CPU sau khi task bị gián đoạn thực thi xong.



Mọi người thấy bài viết hữu ích thì ủng hộ team để mọi người có động lực viết những topic "sang xịn mịn" hơn nhé!!!

**RTOS Phần 3: Interrupt**

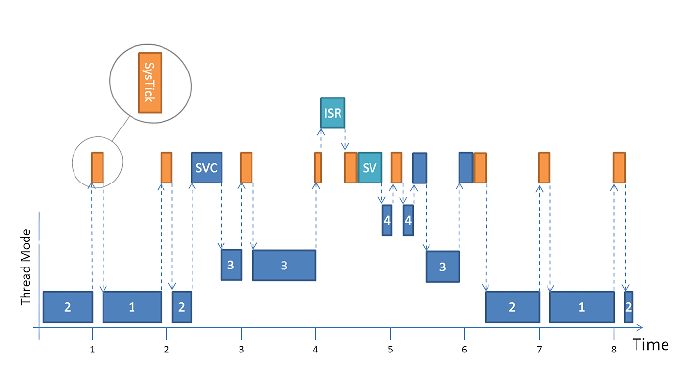
Liệu ngắt trong hệ điều hành RTOS có khác với ngắt thông thường không nhỉ ?

Link đăng ký khóa học: <https://deviot.vn/chi-tiet-san-pham/lap-trinh-stm32-va-rtos.31078516>

Tổng quan:

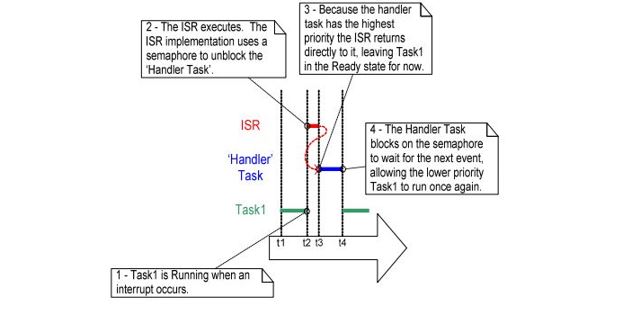
Nhân (Kernel) của RTOS hoạt động dựa vào 1 ngắt timer của chip, ngắt này tạo clock của RTOS gọi là Tick. Mỗi lần Tick thì RTOS sẽ thực hiện các hoạt động phân quyền chạy Task vụ. Tức là cho phép Task nào chạy, Task nào tạm dừng, Task nào chạy trước, Task nào chạy sau.

Khi ngắt Tick xảy ra thì trong đấy sẽ gọi 1 hàm để hệ thống không thể xảy ra ngắt nữa gọi là hàm enter\_critical\_section(). Tại sao vậy? Bởi vì khi Nhân RTOS đang hoạt động nó không muốn có bất kỳ ngắt nào xảy ra làm ảnh hưởng đến hoạt động của nó. Cho đến khi nhân hoạt động xong thì nó mới gọi hàm exit\_critical\_section() để cho phép các ngắt khác có thể xảy ra.



Ngắt thông thường và Ngắt trong RTOS:

Khi ngắt thông thường xảy ra nó sẽ nhảy vào hàm ngắt và các công việc cần phải làm đều đặt trong hàm ngắt. Còn ở RTOS thì khi ngắt xảy ra nó sẽ nhảy vào hàm thủ tục ngắt, hàm thủ tục ngắt lại phát đi 1 "tín hiệu" để hàm thực hiện chức năng của ngắt Unblocking và hàm này có Priority cao hơn tất cả các task khác nên nó sẽ thực hiện ngay lập tức.



Những lưu ý:

·       Không phải API nào của RTOS cũng được phép gọi trong interrupt, chỉ những API có đuôi FromISR mới được phép gọi trong interrupt. Bởi vì đây là các hàm an toàn. Như là:

o   xTaskResumeFromISR()

o   xQueueSendFromISR()

·       Tick RTOS không phải cái quan trọng nhất trong hệ thống, có 1 số cái quan trọng hơn, như USB, truyền thông, nhiều cái chỉ chậm 1 chút thôi sẽ bị hụt mất dữ liệu. Nên Tick RTOS không cần phải có mức ưu tiên cao nhất trừ khi ứng dụng của bạn không có truyền thông và cần đảm bảo độ real time.

**RTOS Phần 4: Queue**

Ở phần 2 ta đã tìm hiểu rõ hơn về task cũng như các phương pháp lập lịch, phần này ta sẽ tìm hiểu cách mà các task trao đổi dữ liệu với nhau

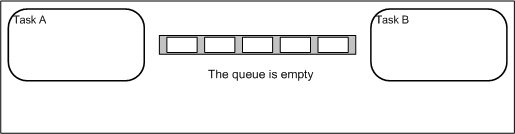
Link đăng ký khóa học: <https://deviot.vn/chi-tiet-san-pham/lap-trinh-stm32-va-rtos.31078516>

Trong quá trình chạy multi-task sẽ có những lúc các task cần sử dụng chung tài nguyên cũng như trao đổi dữ liệu với nhau, thường thì những bạn mới tiếp cận có thể sẽ nghĩ ngay đến biến toàn cục,… Việc này tuy khả thi nhưng có rất nhiều bất cập như không kiểm soát đúng được luồng dữ liệu. Và Queue sẽ giúp các bạn giải quyết vấn đề này 1 cách ổn thỏa nhất.

Queue

Khái niệm

Queue là một loại buffer theo kiểu FIFO (first in first out) nơi dữ liệu sẽ được ghi vào cuối của queue. Tuy nhiên ta vẫn có thể ghi dữ liệu vào đầu queue bằng cách sử dụng hàm của thư viện.



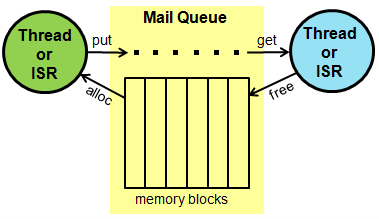
1 queue có thể chữa dữ liệu hoặc con trỏ trỏ đến dữ liệu. Mỗi mục dữ liệu sẽ được ghi trực tiếp vào queue. Kích thước (size) của ô dữ liệu và số lượng ô dữ liệu tối đa (length) được cố đinh khi tạo queue. Và tât nhiên, các ô dữ liệu của 1 queue sẽ có cùng 1 size.

Các thao tác đơn giản với queue



Phụ lục: Mail Queue

Thực ra Mail Queue là 1 dạng của Queue. Ở trong thư viện FreeRTOS thì không hỗ trợ API của phần này nhưng trong thư viện CMSIS (thư viện RTOS trong STM32-cubeMx) thì có. Với Mail Queue thì data được gửi đi dưới dạng block, các memory block này cần được cấp phát (allocate) trước khi đưa data vào và giải phóng(free) sau khi cho data ra.



Các thao tác đơn giản với Mail Queue





*Các bạn thấy bài viết hay thì ủng hộ team DevIot bọn mình nhé !!!*

**RTOS Phần 5: Semaphore**

Ở phần 4 ta đã tìm hiểu về Queue cũng như cách sử dụng nó, ở phần này ta tiếp tục khám phá về cách đồng bộ giữa các task với nhau và với ngắt thông qua Semaphore

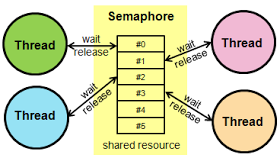
Link đăng ký khóa học: <https://deviot.vn/chi-tiet-san-pham/lap-trinh-stm32-va-rtos.31078516>

Khi thực hiện multi-task trong 1 project. Nếu ta cứ tạo ra các task và để cho chúng chạy độc lập với nhau thì sẽ rất khó khăn trong việc phân chia tài nguyên cho các task cũng như kiểm soát thuật toán của bản thân. Vì thế nên Semarphore được xây dựng để giúp chúng ta giải quyết vấn đề này 1 cách ổn áp nhất.

Tổng quan về Semaphores

Khái niệm:

Semaphore giống như 1 cái hàng đợi (queue), dùng để quản lý và bảo vệ tài nguyên dùng chung(share resource). Các thread/task khác nhau khi có yêu cầu sử dụng tài nguyên dùng chung sẽ bị tống vào hàng đợi này. Khi nhận được semaphore token thì thread/task nào được tống vào queue trước thì sử dụng tài nguyên trước. Sau đó nó lại release ra cho thread/task khác dùng.



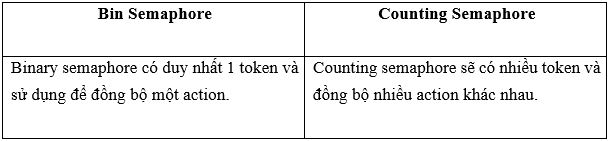
Nguyên lý hoạt động:

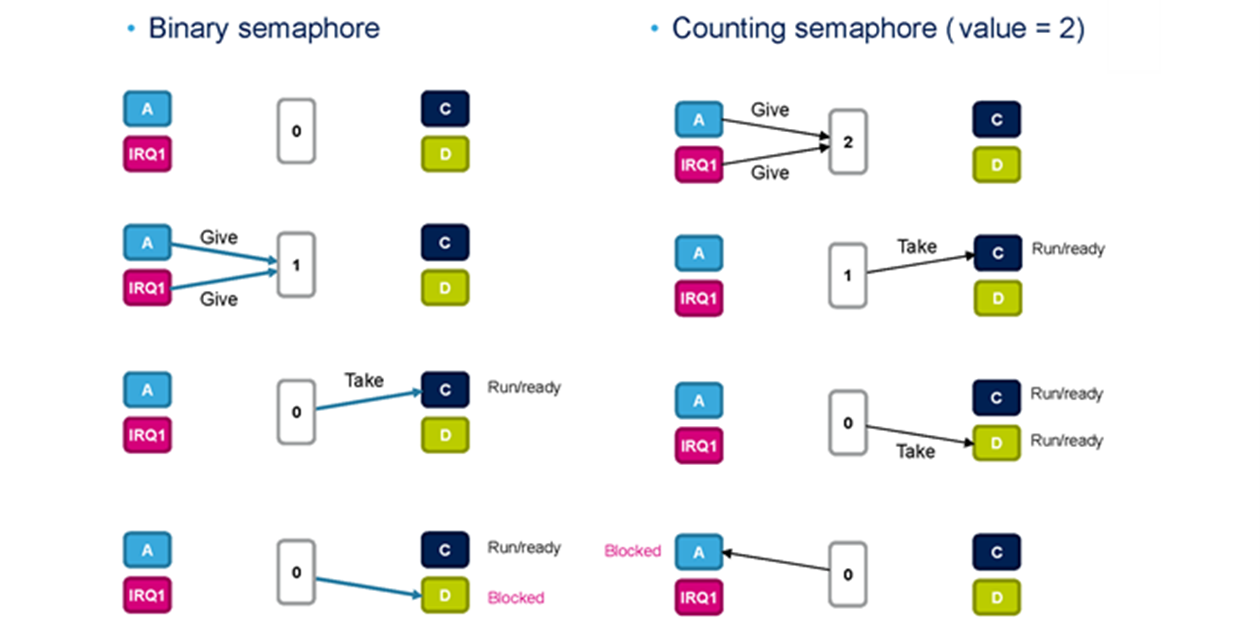
Các bạn hãy tưởng tượng, có 1 căn phòng (tài nguyên của CPU), và để có thể vào được căn phòng đó để thực hiện nhiệm vụ (các câu lệnh) thì các task cần phải có chìa khóa (Semaphore). Và số lượng chìa khóa có thể đánh ra được nhiều cái tùy người dùng config. Trước khi vào phòng, các task sẽ “take” semaphore có sẵn, nếu không thì task sẽ “wait” cho tới khi có semaphore, sau khi đã có được semarphore task sẽ thực thi nhiệm vụ của mình và “Give” semaphore về lại “chỗ cũ”. Ngoài các task ra thì các hàm xử lý ngắt cũng có thể “Give” semaphore.



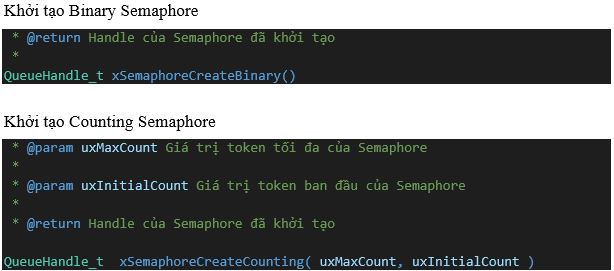
Phân loại

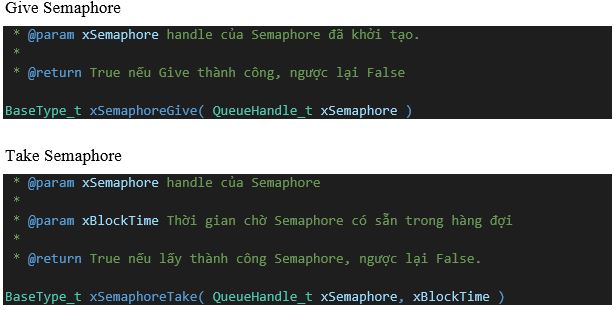
Semaphore được phân làm 2 dạng chính:





Các thao tác cơ bản với Sermaphore





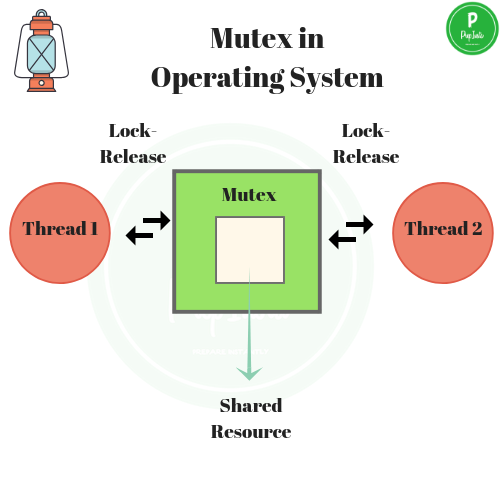
**RTOS Phần 6: MUTEX**

Một người anh em cùng cha khác mẹ với Semaphore, mọi người cùng tìm hiểu về nó nhé ^^

Link đăng ký khóa học: <https://deviot.vn/chi-tiet-san-pham/lap-trinh-stm32-va-rtos.31078516>

Tổng quan

Về cơ bản thì MUTEX tương tự như bin Semaphore nhưng có tích thêm cơ chế “kế thừa mức ưu tiên” và được dùng cho mục đích loại trừ (hạn chế quyền truy cập vào resource của các task khác) chứ không phải là để đồng bộ như Semaphore.



***Kế thừa mức ưu tiên****: Khi task có mức ưu tiên cao hơn muốn take MUTEX trong khi task có mức ưu tiên thấp hơn đang giữ nó thì, mức ưu tiên của 2 task sẽ được tráo cho nhau, cho tới khi task kia give MUTEX thì mức ưu tiên sẽ hoàn trả lại như cũ. Cơ chế này được thiết kế nhằm chắc chắn rằng các task có mức ưu tiên cao chỉ ở trạng thái block trong thời gian ngắn nhất có thể nhằm tránh tình trạng deadlock của hệ thống.*

Nguyên lý hoạt động.

Khi 1 task muốn truy cập vào tài nguyên để thực thi nhiệm vụ thì sẽ take Mutex. Trong lúc đó, bất kì task nào muốn take Mutex đều bị block cho tới khi task đang giữ Mutex “give” về chỗ cũ



Có thể hiểu đơn giản Mutex giống như khóa tủ gửi đồ trong siêu thị, muốn mở được tủ khóa để cất đồ thì cần có chìa khóa, và sau khi dùng xong phải để chìa khóa về chỗ cũ cho người khác dùng.

Một số điểm khác so với Semaphore.

Task đang sở hữu mutex sẽ không bao h phải chờ task khác thực thi, còn task đang sở hữu semaphore thì có thể vẫn phải chờ task khác thực thi xong.

Chỉ có thằng cầm mutex mới được give mutex, bất kì th nào cũng có thể give semaphore. Chính vì vậy trình phục vụ ngắt có thể give Semaphore nhưng đ.v Mutex thì không.

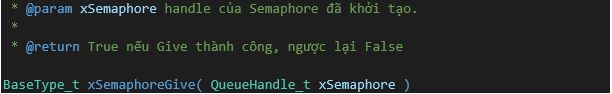
Có thể có nhiều Semaphore, nhưng chỉ có duy nhất 1 mutex.

Các thao tác cơ bản với Mutex.

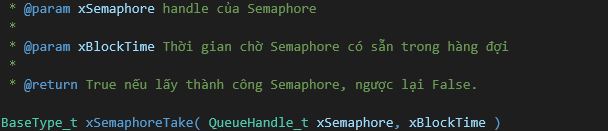
Khởi tạo Mutex



Give Mutex



Take Mutex



**RTOS Phần 7: Software Timer**

Timer "mềm", liệu có được xịn sò như Hareware Timer

Link đăng ký khóa học: <https://deviot.vn/chi-tiet-san-pham/lap-trinh-stm32-va-rtos.31078516>

Tổng quan

     Khái niệm

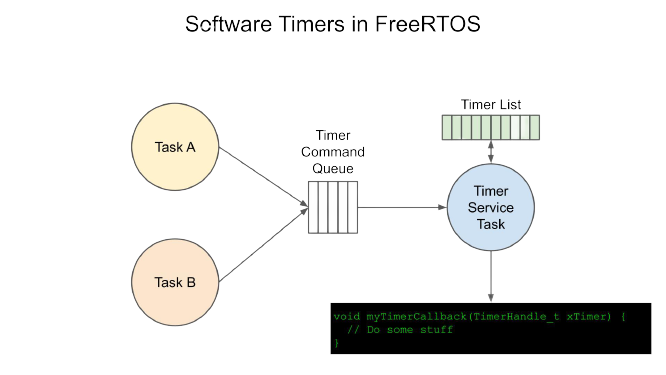
Có thể hiểu đơn giản, trong RTOS, software timer giúp chúng ta thực hiện 1 nhiệm vụ nào đó với 1 chu kỳ nhất định.

     Phân loại

One-shot timer: khi tới thời gian định kỳ được cài đặt từ trước thì hàm callback (vai trò như trình phục vụ ngắt) chỉ được thực thi 1 lần duy nhất. Tuy nhiên vẫn có thể khởi động lại theo cách thủ công chứ không tự khởi động lại.

Auto-reload timer: Khác với One-shot timer là nó sẽ tự khởi động lại sau khi thực thi hàm callback.

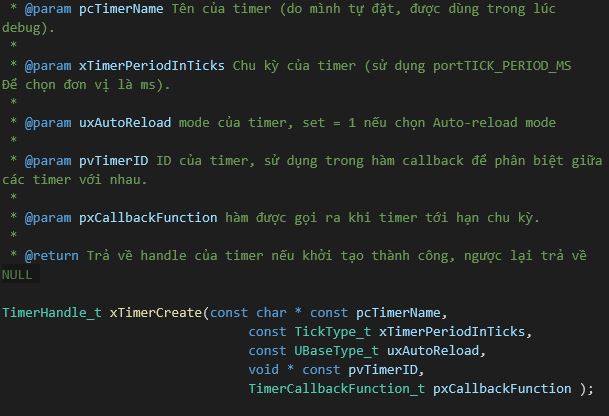
Nguyên lý hoạt động



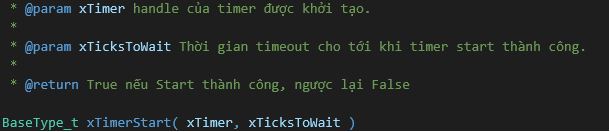
Có 1 task được gọi là Tmr Svc (Timer Serivce) sẽ duy trì “list software timer” được đặt ra. Task này sẽ không chạy liên tục vì nó biết thời gian mà mỗi timer trong timer list “wake up”. Khi timer đến kỳ hạn của nó, thì Tmr Svc sẽ gọi hàm callback của timer đó.Ngoài ra, Tmr Svc còn sử dụng 1 hàng đời Queue, để các task vụ khác có thể giao tiếp và điều khiển timer trong này (start hoặc stop timer).

Các thao tác cơ bản với Software Timer

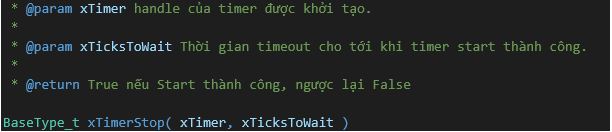
**Tạo Software Timer**



**Start Software Timer**



**Stop Software Timer**



**Reset Timer:**



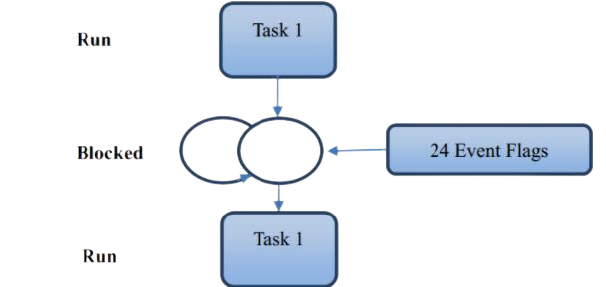
**RTOS Phần 8: Event Groups**

Một cách khác để đồng bộ các task mà không phải dùng Semaphore nhưng tiết kiệm bộ nhớ hơn

Link đăng ký khóa học: <https://deviot.vn/chi-tiet-san-pham/lap-trinh-stm32-va-rtos.31078516>

Tổng quan:

Cũng như Semaphore hay mutex, Event group có thể kiến 1 task rơi vào trạng thái block cũng như unblock task đó. Tuy nhiên, khác ở chỗ, task bị block có thể phải chờ cho nhiều sự kiện. và cùng một lúc có thể unblock nhiều task bằng 1 hoặc nhiều sự kiện.

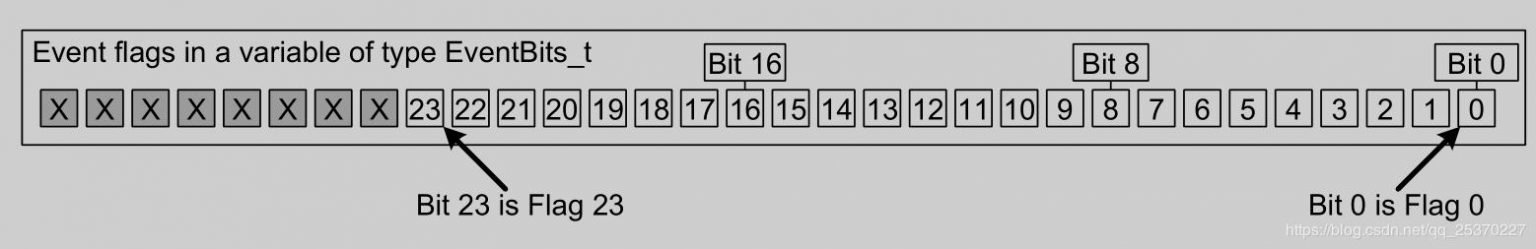


Tại sao nên dùng event group:

Như các bạn đã biết thì các ứng dụng về lập trình nhúng đòi hỏi người lập trình phải biết tối ưu về bộ nhớ. Bằng cách sử dụng event group,       chúng ta có thể giảm thiểu việc sử dụng RAM. Bởi thay vì sử dung nhiều Bin-Sema, chúng ta chỉ cần thay thế chúng bằng một event group duy nhất trong nhiều trường hợp.

Nguyên lý hoạt động:

Event group về có bản là một tập hợp các event flag mang giá trị 0 hoặc 1. Những giá trị này thể hiện sự có hay không của 1 sự kiện nào đó.



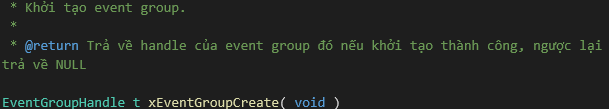
Nhờ đó thì ta có thể lưu trạng thái của 1 event flag chỉ bằng 1 bit duy nhất, và các bit đó được lưu giữ trong một biến gọi là event group. Có tối đa 24 event flag tương ứng với 24 event trong 1 event group. Được cấu hình qua configUSE\_16\_BIT\_TICKS

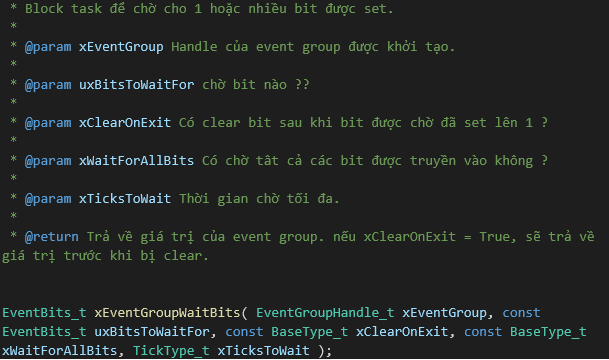
·       If configUSE\_16\_BIT\_TICKS = 1: 8 flags.

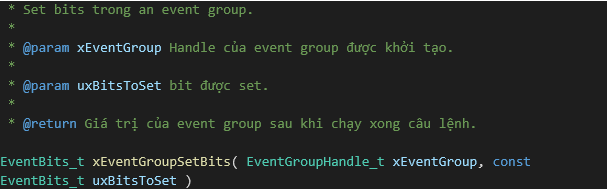
·       If configUSE\_16\_BIT\_TICKS = 0: 24 flags.

Kiểu dữ liệu **EventBits\_t** được dùng để chứa các event bits trong 1 event group.

Các thao tác với event group.







**RTOS Ngoại Truyện: Delay vs vTaskDelay vs vTaskDelayUntil**

Một bài viết chiều theo yêu cầu của member :v

Link đăng ký khóa học: <https://deviot.vn/chi-tiet-san-pham/lap-trinh-stm32-va-rtos.31078516>

**Tổng quan:**

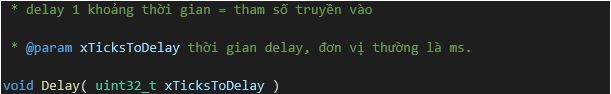
**Delay**: là loại Delay được sử dụng phổ biến, không thuộc hệ điều hành nào cả. Tùy vào thư viện thì nó sẽ có API khác nhau ( HAL\_Delay() trong STM32 thư viện HAL, hay delay() trong Arduino). Khi dùng hàm này trong RTOS sẽ không đưa task vào trạng thái block

**vTaskDelay**: Hàm delay thuộc hệ điều hành RTOS, khi thực hiện hàm này sẽ đưa task vào trạng thái block để có thể thực hiện task các.

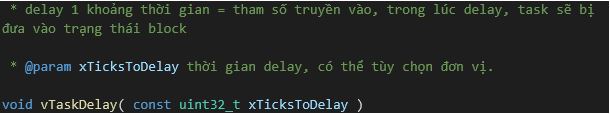
**vTaskDelayUntil**: Giống như vTaskDelay. Nhưng hoạt động dựa trên timeline Tick của hệ điều hành. (mỗi Tick = 1ms).

**Cấu trúc câu lệnh:**

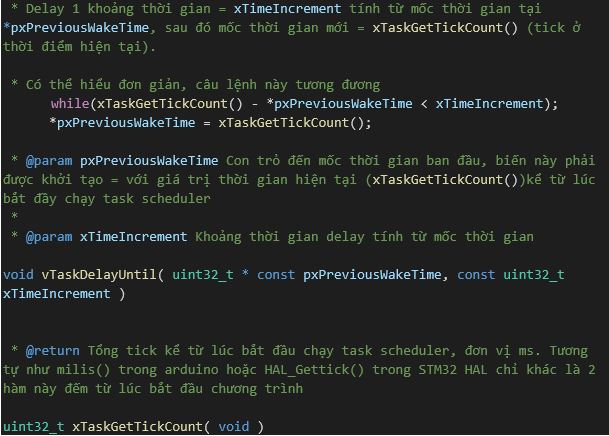
**Delay**



**vTaskDelay**



**vTaskDelayUntil**



**Ngoại Truyện 2: Memory Management**

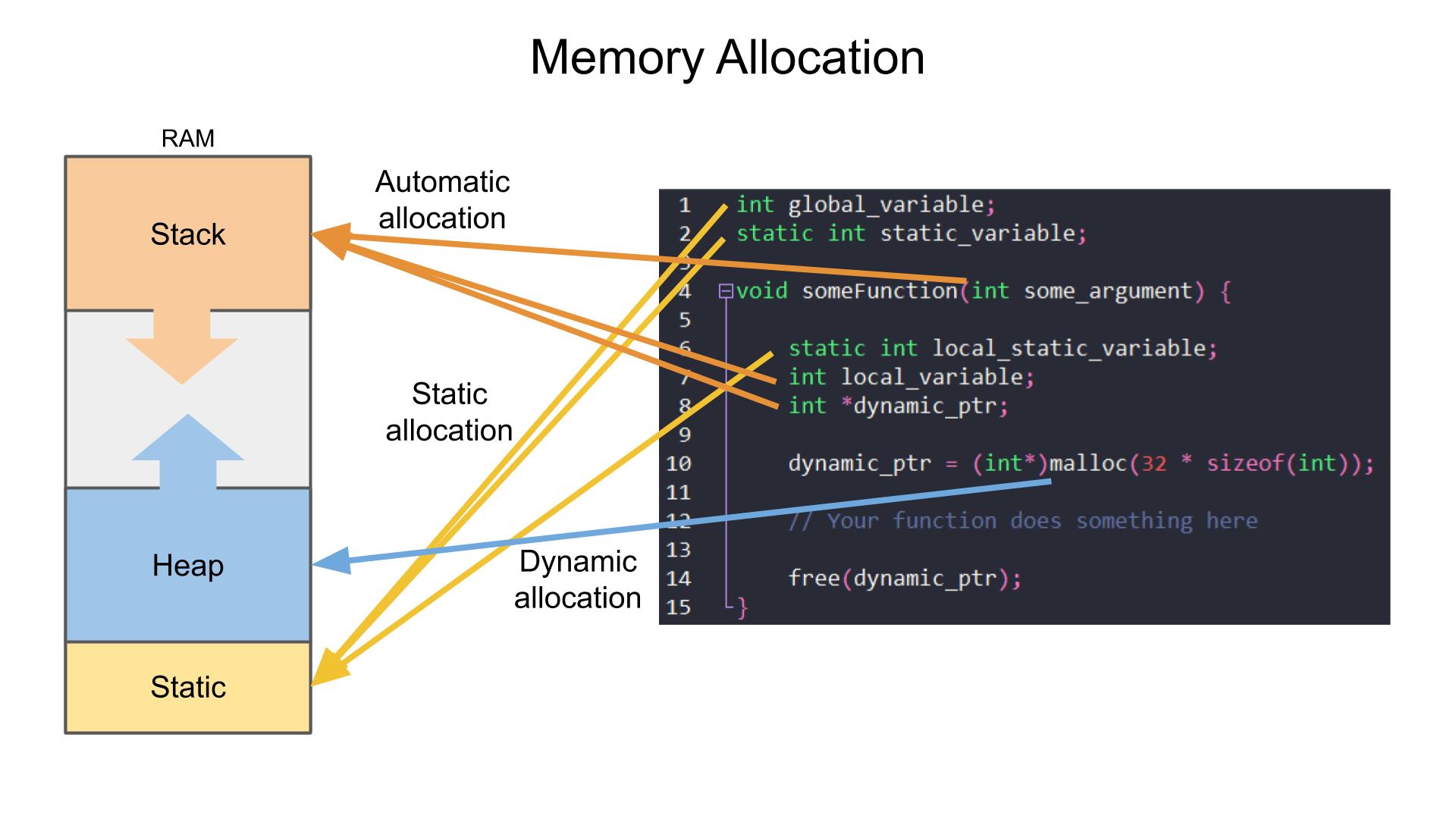
Phần kiến thức quan trọng trước khi bắt đầu làm quen với hệ điều hành RTOS

Link đăng ký khóa học: <https://deviot.vn/chi-tiet-san-pham/lap-trinh-stm32-va-rtos.31078516>

Khi làm việc với hệ điều hành RTOS, ta cần cung cấp 1 vùng nhớ cố định khi khởi tạo 1 task, chính vì thế việc hiểu cách phân vùng bộ nhớ trong RTOS khá là quan trọng, không thể cứ cấp phát bừa bãi được, như thế khá là tốn tài nguyên khi làm dự án lớn.

NON-RTOS

Volatile memory ( như RAM). Trong phần lớn hệ thống vi điều khiển được chia làm 3 vùng: static, stack, và heap.



Static memory được dùng để lưu trữ các biến toàn cục và các biến được khai báo “static”.

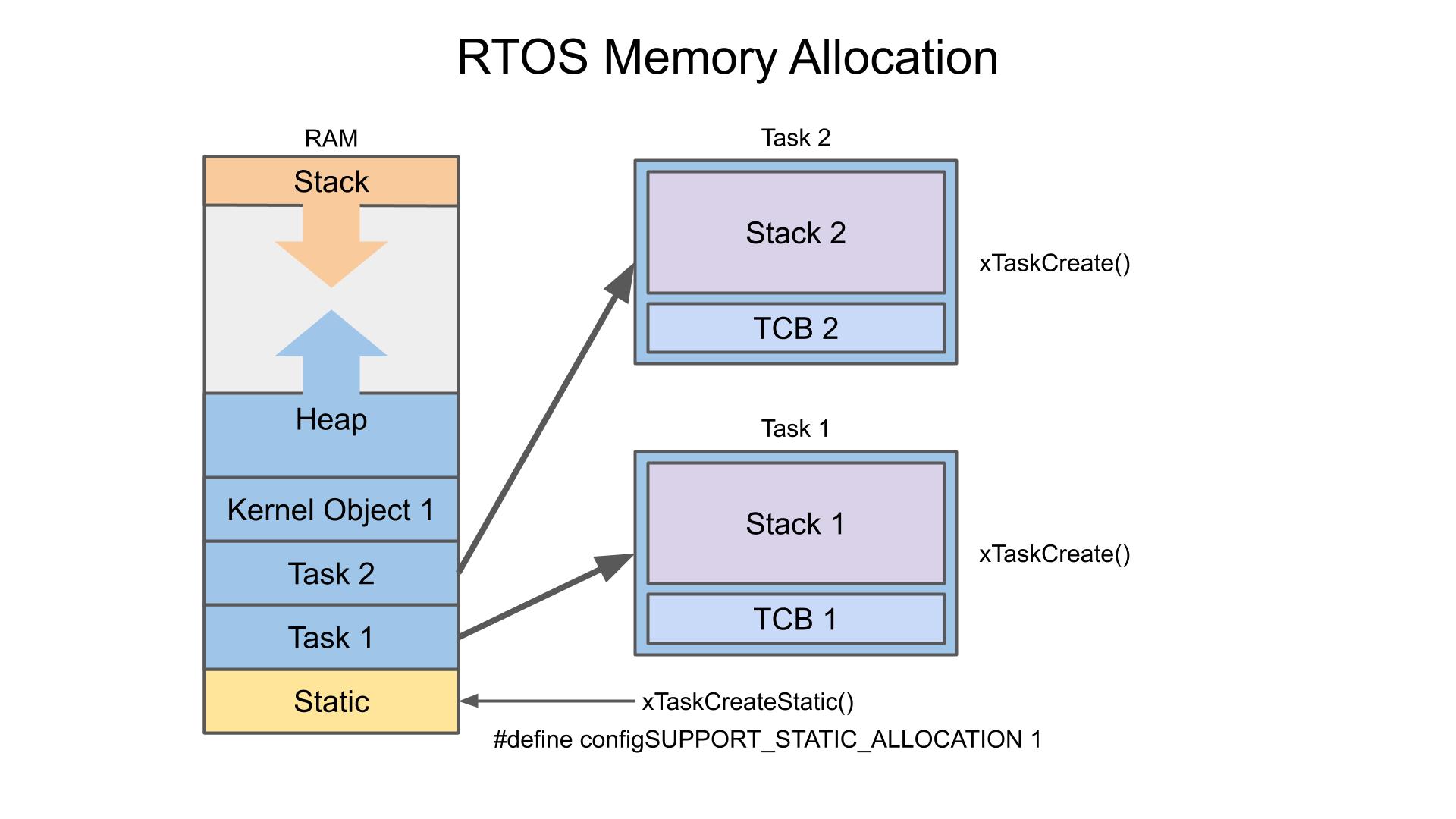
Stack memory là nơi lưu trữ các biến cục bộ, được tổ chức dưới dạng (LIFO) last-in-first-out để các biến của 1 hàm có thể được đẩy vào khi hàm đó được gọi.

Heap memory thì phải được phân bổ rõ ràng bởi lâp trình viên. Thông thường thì bạn hay sử dụng hàm MALLOC() để phẩn bổ heap cho các biến, vv.v. Hay còn được hiểu là cấp phát động. Và lưu ý, khi biến đó không còn được sử dụng ta nên giải phóng vùng nhớ đã cấp phát cho nó.

Một lưu ý nữa, vì Stack và Heap không có “ranh giới rạch ròi” nên dữ liệu ở 2 vùng nhớ này có thể ghi đè lên nhau, như các bạn thấy ở trên hình, thằng heap thì “đi từ dưới lên”, thằng stack thì lại “đi từ trên xuống” nên nếu không có sự kiểm soát thì sớm hay muộn cả 2 cũng sẽ “gặp nhau” và “đè lên nhau”.

RTOS

Khi bạn khởi tạo 1 task vụ trong RTOS (ví dụ xTaskCreate() hoặc osThreadCreate() ), thì hệ điều hành sẽ phân bổ 1 phần bộ nhớ heap cho task đó.



Một phần trong đó được gọi là Task control block (TCB). Là nơi lưu trữ các thông tin của task như mức ưu tiên, con trỏ đại diện,…

Phần còn lại là local stack, hoạt động như global stack (nhưng quy mô nhỏ hơn). Các biến cục bộ được khởi tạo trung suốt quá trình gọi hàm trong task sẽ được lưu trữ ở đây. Do đó ta cần tính toán được kích thước stack và truyền vào tham số stack size khi khởi tạo task.

Trong RTOS, không nên sử dụng malloc() và free() mà nên thay thế bằng pvPortMalloc() và vPortFree() để thay thế, vì nó sẽ an toàn hơn trong hệ điều hành. Khi sử dụng chúng, thì bộ nhớ cấp phát sẽ nằm trong vùng heap của toàn hệ thống thay vì vùng heap được cấp phát riếng cho task.

Đặc biệt, ở trong các phiên bản gần đây của thư viện RTOS, thì nó cho phép khởi tạo các task static (xTaskCreateStatic() ). Điều này giúp cho chúng ta rất nhiều khi mà ta không thể hoặc khổng muốn sử dụng bộ nhớ heap nữa để tránh tràn heap.