# BẮT ĐẦU VỚI NGÔN NGỮ C

## PHẦN 1.1 HELLO WORLD

Để tạo một chương trình C đơn giản in dòng chữ "Hello, World " lên màn hình, sử dụng trình soạn thảo văn bản để tạo file mới (ví dụ:

hello.c — phần mở rộng file phải là .c) chứa mã nguồn sau:

**hello.c**

|  |
| --- |
| # include<stdio.h>  int main(void)  {      puts("Hello, World");      return 0;  } |

**Hãy xem từng dòng trong chương trình đơn giản này**

|  |
| --- |
| # include<stdio.h> |

Dòng mã này yêu cầu trình biên dịch cần thêm nội dung của tệp tiêu đề thư viện chuẩn **stdio.h** vào chương trình.

Tệp tiêu đề thường chứa các khai báo của các hàm, macros và kiểu dữ liệu trong thư viện chuẩn.

và bạn cần phải thêm vào tệp tiêu đề trước khi sử dụng chúng. Dòng này thêm vào thư viện **stdio.h** để nó có thể gọi hàm **puts().**

Tìm hiểu thêm về tệp tiêu đề

|  |
| --- |
| int main(void) |

Dòng mã này bắt đầu định nghĩa của một hàm. Nó cho biết tên của hàm (main), kiểu dữ liệu và số lượng các đối số mà nó mong đợi (void, nghĩa là không có) và loại giá trị mà hàm này trả về (int). Theo nguyên tắc, **main** là một tên đặc biệt, nó chỉ dùng để đặt cho function chính của chương trình, và lúc nào chương trình cũng sẽ bắt đầu từ **function main**.

Việc thực thi chương trình bắt đầu trong hàm main().

|  |
| --- |
| {      ...  } |

Dấu ngoặc nhọn (curly braces) được sử dụng thành cặp để chỉ định vị trí bắt đầu và kết thúc một khối mã(khối lệnh). Chúng có thể được sử dụng theo nhiều cách khác nhau, nhưng trong trường hợp này, chúng chỉ định vị trí bắt đầu và kết thúc của một hàm.

|  |
| --- |
| puts("Hello, World"); |

Dòng này gọi hàm puts() để đưa ra văn bản ra đầu ra tiêu chuẩn (mặc định là màn hình), tiếp theo là một dòng mới.

Chuỗi cần đưa ra được bao gồm trong cặp dấu ngoặc đơn().

"Hello, World" là chuỗi sẽ được ghi ra màn hình. Trong ngôn ngữ C, mọi giá trị chuỗi phải nằm trong dấu ngoặc kép "...".

Tìm hiểu thêm về chuỗi.

Trong các chương trình C, mỗi câu lệnh cần được kết thúc bằng dấu chấm phẩy (tức là **;**).

|  |
| --- |
| return 0; |

Khi chúng ta định nghĩa hàm **main(),** chúng ta khai báo nó là một hàm trả về kiểu **int**, có nghĩa là nó cần trả về một giá trị số nguyên. Trong ví dụ này, chúng ta đang trả về giá trị nguyên là **0**, được sử dụng để chỉ rằng chương trình đã kết thúc thành công. Sau câu lệnh **return 0;**, quá trình thực thi sẽ kết thúc.

**Chỉnh sửa chương trình**

Các trình soạn thảo, chỉnh sửa văn bản đơn giản bao gồm **vim** hoặc **gedit** trên **Linux,** hoặc **Notepad** trên **Windows.** Các trình soạn thảo đa nền tảng khác bao gồm **Visual Studio Code** hoặc **Sublime Text.**

Trình chỉnh sửa phải tạo các tệp văn bản thuần túy, không phải RTF hoặc bất kỳ định dạng nào khác.

**Biên dịch và chạy chương trình**

Để chạy chương trình, file nguồn này (hello.c) trước tiên cần được biên dịch thành file thực thi (ví dụ: **hello** trên hệ thống **Unix/Linux** hoặc **hello.exe** trên **Windows**). Điều này được thực hiện bằng cách sử dụng trình biên dịch cho ngôn ngữ C.

**Biên dịch bằng GCC**

GCC **(GNU Compiler Collection)** là một trình biên dịch C được sử dụng rộng rãi. Để sử dụng nó, hãy mở một thiết bị đầu cuối, sử dụng dòng lệnh để điều hướng đến vị trí của tệp nguồn rồi chạy:

|  |
| --- |
| gcc hello.c -o hello |

Nếu không tìm thấy lỗi trong mã nguồn (hello.c), trình biên dịch sẽ tạo một tệp nhị phân, tên của tệp được cung cấp bởi đối số cho tùy chọn dòng lệnh -o (hello). Đây là tập tin thực thi cuối cùng.

Chúng tôi cũng có thể sử dụng các tùy chọn cảnh báo *-Wall -Wextra -Werror*, giúp xác định các sự cố có thể khiến chương trình bị lỗi hoặc tạo ra kết quả không mong muốn. Chúng không cần thiết cho chương trình đơn giản này nhưng đây là cách thêm chúng:

|  |
| --- |
| gcc -Wall -Wextra -Werror -o hello hello.c |

**Sử dụng trình biên dịch clang**

Để biên dịch chương trình bằng **clang**, bạn có thể sử dụng:

|  |
| --- |
| clang -Wall -Wextra -Werror -o hello hello.c |

Theo thiết kế, các tùy chọn dòng lệnh **clang** tương tự như của **GCC.**

**Sử dụng trình biên dịch Microsoft C từ dòng lệnh**

Nếu sử dụng trình biên dịch Microsoft cl.exe trên hệ thống Windows hỗ trợ Visual Studio và nếu tất cả các biến môi trường được đặt, ví dụ C này có thể được biên dịch bằng lệnh sau. Lệnh này sẽ tạo ra tệp hello.exe có thể thực thi được trong thư mục mà lệnh được thực thi trong (Có các tùy chọn cảnh báo như /W3 cho cl, đại khái là tương tự như -Wall vv cho GCC hoặc clang).

|  |
| --- |
| cl hello.c |

**Thực hiện chương trình**

Sau khi được biên dịch, tệp nhị phân sau đó có thể được thực thi bằng cách nhập ./hello trong thiết bị đầu cuối. Khi thực thi, chương trình đã biên dịch sẽ in **Hello, World**, theo sau là một dòng mới, tới dấu nhắc lệnh.

## PHẦN 1.2 BẢN GỐC “HELLO WORLD!” TRONG K&R C

Sau đây là bản gốc "Hello World!" chương trình từ cuốn sách **The C Programming Language** của BrianKernighan và Dennis Ritchie (Ritchie là nhà phát triển ban đầu của ngôn ngữ lập trình C tại Bell Labs), được gọi là "K&R":

|  |
| --- |
| Version = K&R  #include <stdio.h>  main()  {   printf("hello, world\n");  } |

Lưu ý rằng **The C Programming Language** không được chuẩn hóa vào thời điểm viết ấn bản đầu tiên của cuốn sách này (1978), và chương trình này có thể sẽ không được biên dịch trên hầu hết các trình biên dịch hiện đại trừ khi chúng được hướng dẫn chấp nhận mã C90.

Câu ví dụ đầu tiên trong cuốn sách K&R (**The C Programming Language**) hiện được coi là chất lượng kém, một phần là do thiếu một kiểu trả về rõ ràng cho **main()** và một phần là do thiếu câu lệnh **return**. Phiên bản thứ hai của cuốn sách được viết cho tiêu chuẩn C89 cũ. Trong C89, kiểu của main sẽ mặc định là int, nhưng ví dụ K&R không trả về một giá trị xác định cho môi trường. Trong các tiêu chuẩn C99 và sau này, kiểu trả về là bắt buộc, nhưng có thể bỏ qua câu lệnh **return** của main (**và chỉ main**), vì có một trường hợp đặc biệt được giới thiệu với C99 5.1.2.2.3 - nó tương đương với việc trả về 0, biểu thị thành công.

Dạng chính được khuyến nghị và di động nhất cho các hệ thống được lưu trữ là **int main(void)** khi chương trình không sử dụng bất kỳ đối số dòng lệnh nào, hoặc **int main(int argc, char \*\*argv)** khi chương trình sử dụng các đối số ở các dòng lệnh.

**C90 §5.1.2.2.3 Kết thúc chương trình**

Một câu lệnh **return** từ lời gọi ban đầu tới hàm **main** tương đương với việc gọi hàm **exit** với giá trị trả về của hàm **main** là **đối số**. Nếu hàm main thực hiện một câu lệnh **return** mà không xác định giá trị, trạng thái kết thúc trả về cho môi trường chủ không được xác định.

**C90 §6.6.6.4 Câu lệnh return**

Nếu một câu lệnh **return** mà không có biểu thức được thực thi và giá trị của lời gọi hàm được sử dụng bởi người gọi, hành vi đó là không xác định. Đạt đến dấu **}** kết thúc một hàm tương đương với việc thực thi một câu lệnh **return** mà không có biểu thức.

**C99 §5.1.2.2.3 Kết thúc chương trình**

Nếu kiểu trả về của hàm **main** là một kiểu tương thích với **int**, một câu lệnh **return** từ lời gọi ban đầu tới hàm **main** tương đương với việc gọi hàm **exit** với giá trị trả về của hàm main là đối số; đạt đến dấu **}** kết thúc hàm main sẽ trả về giá trị 0. Nếu kiểu trả về không tương thích với int, trạng thái kết thúc trả về cho môi trường chủ không được xác định.

# COMMENTS

Chú thích (comments) được sử dụng để chỉ điều gì đó cho người đọc mã nguồn. Chú thích được coi như một phần trống trên mã nguồn và không thay đổi bất kỳ ý nghĩa thực sự nào của mã. Có hai cú pháp được sử dụng cho chú thích trong ngôn ngữ C, đó là **/\* \*/** (bản gốc) và **//** (bản mới hơn 1 chút). Một số hệ thống tài liệu sử dụng chú thích được định dạng đặc biệt để hỗ trợ tạo tài liệu cho mã nguồn.

## PHẦN 2.1 CHÚ THÍCH BẰNG CÁCH SỬ DỤNG TIỀN XỬ LÝ(PREPROCESSOR)

Một phần lớn mã nguồn cũng có thể bị "chú thích" bằng cách sử dụng các chỉ thị tiền xử lý **#if 0** và **#endif.** Điều này hữu ích khi mã chứa các chú thích nhiều dòng mà không thể lồng nhau nếu không có cơ chế chú thích này.

|  |
| --- |
| #if 0 /\* Bắt đầu "chú thích", bất cứ điều gì từ đây trở đi sẽ bị loại bỏ bởi tiền xử lý \*/  /\* Một lượng mã lớn với các chú thích nhiều dòng \*/  int foo()  {  /\* một số câu lệnh \*/  ...  /\* ... một chú thích mô tả câu lệnh if ... \*/  if (someTest) {   /\* some more comments \*/  return 1;  }  return 0;  }  #endif /\* 0 \*/  /\* Mã từ đây trở đi không bị "chú thích" (được bao gồm trong tệp thực thi biên dịch) \*/ |

## PHẦN 2.2 CHÚ THÍCH ĐƯỢC GIỚI HẠN BẰNG /\* \*/

Một chú thích bắt đầu với một dấu gạch chéo kề ngay sau đó là một dấu hoa thị (/\*) và kết thúc ngay khi gặp một dấu hoa thị kề ngay sau đó là một dấu gạch chéo (\*/). Mọi thứ nằm giữa các cặp ký tự này đều là chú thích và được coi là một phần trống (tức là bị bỏ qua) bởi trình biên dịch.

|  |
| --- |
| /\* Đây là một chú thích \*/ |

Chú thích ở trên là chú thích trên một dòng. Chú thích kiểu /\* này có thể trải dài trên nhiều dòng, như sau:

|  |
| --- |
| /\* Đây là một chú thích  nhiều dòng \*/ |

Mặc dù điều này không bắt buộc, quy ước phong cách thông thường với chú thích trên nhiều dòng là đặt khoảng trắng và dấu hoa thị ở các dòng sau đầu tiên và đặt /\* và \*/ trên các dòng mới, sao cho chúng xếp thành hàng:

|  |
| --- |
| /\* Đây là   \*một chú   \* thích   \*nhiều dòng  \*/ |

Những dấu hoa thị thêm không có tác dụng chức năng nào đối với chú thích vì chúng không có dấu gạch chéo liên quan.

Các chú thích kiểu /\* này có thể được sử dụng trên một dòng riêng biệt, ở cuối một dòng mã hoặc ngay cả trong các dòng mã:

|  |
| --- |
| /\* Chú thích này nằm trên một dòng riêng biệt \*/  if(x && y) { /\* Chú thích này nằm ở cuối dòng mã \*/      if ((complexCondition1) /\* Chú thích này nằm trong một dòng mã \*/          && (complexCondition2)) {      /\* Chú thích này nằm trong một câu lệnh if, trên một dòng riêng biệt \*/      }  } |

Chú thích không thể lồng nhau. Điều này xảy ra vì bất kỳ /\* nào tiếp theo sẽ bị bỏ qua (là một phần của chú thích) và đầu tiên \*/ gặp được sẽ được coi là kết thúc chú thích. Chú thích trong ví dụ dưới đây sẽ không hoạt động:

|  |
| --- |
| /\* Chú thích bên ngoài, có nghĩa là điều này bị bỏ qua => /\* chú thích nội bộ cố gắng \*/ <= kết thúc chú thích,  không phải chú thích này => \*/ |

Để chú thích các khối mã chứa chú thích kiểu này, mà nếu không sẽ bị lồng nhau, bạn có thể tham khảo ví dụ về Chú thích bằng cách sử dụng tiền xử lý.

## PHẦN 2.3 CHÚ THÍCH ĐƯỢC GIỚI HẠN BẰNG DẤU //

Phiên bản ≥ C99

C99 đã giới thiệu việc sử dụng chú thích trên một dòng theo kiểu C++. Loại chú thích này bắt đầu bằng hai dấu gạch chéo kề nhau và kéo dài đến cuối dòng:

|  |
| --- |
| // Mỗi dòng trong số này đều là một chú thích trên một dòng  // Chú ý rằng mỗi dòng phải bắt đầu bằng  // hai dấu gạch chéo kép liên tiếp (//) |

Loại chú thích này có thể được sử dụng trên một dòng riêng biệt hoặc ở cuối một dòng mã. Tuy nhiên, do chú thích này kéo dài đến cuối dòng, nên chúng không thể được sử dụng trong một dòng mã.

|  |
| --- |
| // Chú thích này nằm trên một dòng riêng biệt  if (x && y) { // Chú thích này nằm ở cuối dòng  // Chú thích này nằm trong một câu lệnh if, trên một dòng riêng biệt  } |

## PHẦN 2.4 RỦI RO CÓ THỂ XẢY RA DO TRIGRAPH

Trong quá trình viết chú thích được giới hạn bởi //, có thể xảy ra lỗi chính tả ảnh hưởng đến hoạt động dự kiến của chú thích. Nếu bạn gõ sai như sau:

|  |
| --- |
| int a = 10 //Sao tôi làm điều này??/ |

Dấu gạch chéo / ở cuối là một lỗi chính tả nhưng giờ đây sẽ được hiểu là dấu gạch chéo ngược . Điều này xảy ra vì ??/ tạo thành một trigraph. Trigraph ??/ thực tế là một cách viết dài cho dấu gạch chéo ngược , đây là ký hiệu tiếp tục dòng. Điều này có nghĩa là trình biên dịch sẽ hiểu rằng dòng tiếp theo là tiếp tục của dòng hiện tại, tức là tiếp tục của chú thích, điều này có thể không phải ý định ban đầu.

|  |
| --- |
| int foo = 20; // Bắt đầu từ 20 ??/  int bar = 0;  // Dòng sau sẽ gây lỗi biên dịch (biến 'bar' chưa được khai báo)  // vì 'int bar = 0;' là một phần của chú thích trên dòng trước đó  bar += foo; |

# KIỂU DỮ LIỆU

## PHẦN 3.1 GIẢI THÍCH CÁC KHAI BÁO

Một đặc điểm cú pháp đặc biệt của ngôn ngữ C là các khai báo phản ánh việc sử dụng của đối tượng được khai báo như trong một biểu thức thông thường.

Tập hợp các toán tử sau đây có cùng mức độ ưu tiên và tính kết hợp được tái sử dụng trong các khai báo, bao gồm:

* Toán tử unary \* "dereference" (giải tham chiếu) thể hiện một con trỏ.
* Toán tử binary [] "array subscription" (phần tử mảng) thể hiện một mảng.
* Toán tử (1+n)-ary () "function call" (gọi hàm) thể hiện một hàm.
* Dấu ngoặc () nhóm nhưng ghi đè mức độ ưu tiên và tính kết hợp của các toán tử còn lại được liệt kê.

Ba toán tử trên có độ ưu tiên và tính kết hợp như sau:

|  |  |  |
| --- | --- | --- |
| Toán tử | Mức độ ưu tiên | Tính kết hợp |
| [ ] Phần tử mảng | 1 | Trái sang phải |
| ( ) Gọi hàm | 1 | Trái sang phải |
| \*Giải tham chiếu | 2 | Phải sang trái |

Khi giải thích các khai báo, người đọc cần bắt đầu từ từng định danh và áp dụng các toán tử kề cạnh theo thứ tự đúng như trong bảng trên. Mỗi lần áp dụng một toán tử, ta có thể thay thế nó bằng các từ tiếng Anh sau đây:

|  |  |
| --- | --- |
| Biểu thức | Giải thích |
| Thing[X] | Kích thức của mảng X là… |
| Thing(x1,x2,x3) | 1 hàm nhận x1 x2 x3 và trả về … |
| \*Thing | Một con trỏ trỏ tới .. |

Theo đó phần bắt đầu của việc giải thích bằng tiếng Anh sẽ luôn bắt đầu với định danh và kết thúc với kiểu dữ liệu nằm ở phía bên trái của khai báo.

**Ví dụ**

|  |
| --- |
| char \*names[20]; |

[] được ưu tiên hơn \*, vì vậy cách hiểu là: **names** là một mảng có kích thước 20 của một con trỏ trỏ tới char.

|  |
| --- |
| char (\*place)[10]; |

Trong trường hợp sử dụng dấu ngoặc () để ghi đè mức độ ưu tiên, toán tử \* được áp dụng trước: **place** là một con trỏ tới một mảng có kích thước là 10 của char.

|  |
| --- |
| int fn(long, short); |

Ở đây không có ưu tiên cần quan tâm: fn là một hàm nhận tham số long và short, và trả về kiểu int.

|  |
| --- |
| int \*fn(void); |

Dấu ngoặc () được áp dụng trước: fn là một hàm nhận tham số void và trả về một con trỏ tới kiểu int.

|  |
| --- |
| int (\*fp)(void); |

Ghi đè mức độ ưu tiên của (): fp là một con trỏ tới một hàm nhận tham số void và trả về kiểu int.

|  |
| --- |
| int arr[5][8]; |

Mảng đa chiều không phải là một ngoại lệ trong quy tắc; các toán tử [] được áp dụng theo thứ tự từ trái sang phải theo tính kết hợp được liệt kê trong bảng: arr là một mảng có kích thước 5 của một mảng có kích thước 8 của kiểu int.

|  |
| --- |
| int \*\*ptr; |

Hai toán tử dereference có cùng mức độ ưu tiên, do đó tính kết hợp có hiệu lực. Các toán tử được áp dụng theo thứ tự từ phải sang trái: ptr là một con trỏ tới một con trỏ tới kiểu int.

**Khai báo nhiều biến**

Dấu phẩy có thể được sử dụng làm dấu phân tách (không hoạt động như toán tử dấu phẩy) để phân cách các khai báo đa biến trong cùng một câu lệnh. Câu lệnh sau chứa năm khai báo:

|  |
| --- |
| int fn(void), \*ptr, (\*fp)(int), arr[10][20], num; |

Các đối tượng được khai báo trong ví dụ trên là:

* fn: một hàm không tham số và trả về kiểu int.
* ptr: một con trỏ tới kiểu int.
* fp: một con trỏ hàm nhận tham số kiểu int và trả về kiểu int.
* arr: một mảng có kích thước 10, mỗi phần tử là một mảng có kích thước 20 kiểu int.
* num: kiểu int.

**Giải thích thay thế**

Bởi vì các khai báo phản ánh việc sử dụng, nên một khai báo cũng có thể được giải thích dựa trên các toán tử có thể được áp dụng lên đối tượng và kiểu dữ liệu cuối cùng của biểu thức đó. Kiểu dữ liệu nằm ở phía bên trái là kết quả cuối cùng sau khi áp dụng tất cả các toán tử.

|  |
| --- |
| /\*   \*Sử dụng toán tử [] để lấy phần tử của "arr" và sau đó dùng toán tử dereference để lấy giá trị,   \*ta được một kết quả kiểu "char".   \*Cụ thể: \*arr[5] có kiểu "char".   \*/  char arr[20];  /\*   \*Gọi hàm "fn" trả về một kết quả kiểu "int".   \*Cụ thể: fn('b') có kiểu "int".   \*/  int fn(char);  /\*   \*Sử dụng toán tử dereference để lấy giá trị của "fp", sau đó gọi nó trả về một kết quả kiểu "int".   \*Cụ thể: (\*fp)() có kiểu "int".   \*/  int (fp)(void);  /\*   \*Sử dụng toán tử [] hai lần để lấy phần tử của "strings", sau đó sử dụng toán tử dereference để lấy giá trị,   \*ta được một kết quả kiểu "char".   \*Cụ thể: \*strings[5][15] có kiểu "char".   \*/  char \*strings[10][20]; |

## PHẦN 3.2 KIỂU SỐ NGUYÊN CÓ ĐỘ RỘNG CỐ ĐỊNH ( KỂ TỪ C99)

Phiên bản ≥ C99 Tiêu đề <stdint.h> cung cấp một số định nghĩa kiểu số nguyên có độ rộng cố định. Các kiểu này là tùy chọn và chỉ được cung cấp nếu nền tảng có một kiểu số nguyên có độ rộng tương ứng và nếu kiểu có dấu tương ứng có biểu diễn bù hai của các giá trị âm. Xem phần lưu ý để biết các gợi ý về việc sử dụng các kiểu có độ rộng cố định.

|  |
| --- |
| /\* Các kiểu phổ biến thường được sử dụng bao gồm \*/  *uint32\_t* u32 = 32; / có độ rộng chính xác 32 bit /  uint8\_t u8 = 255; / có độ rộng chính xác 8 bit /  int64\_t i64 = -65; / có độ rộng chính xác 64 bit trong biểu diễn bù hai \*/ |

## PHẦN 3.3 KIỂU SỐ NGUYÊN VÀ HẰNG SỐ

Các số nguyên có dấu có thể thuộc các kiểu sau đây (int sau từ short hoặc long là tùy chọn):

|  |
| --- |
| signed char c = 127; /\* yêu cầu có độ rộng 1 byte, xem phần lưu ý để biết thêm thông tin \*/  signed short int si = 32767; /\* yêu cầu có ít nhất 16 bit \*/  signed int i = 32767; /\* yêu cầu có ít nhất 16 bit \*/  signed long int li = 2147483647; /\* yêu cầu có ít nhất 32 bit \*/  Phiên bản ≥ C99  signed long long int li = 2147483647; /\* yêu cầu có ít nhất 64 bit \*/ |

Mỗi kiểu số nguyên có dấu này đều có một phiên bản không dấu tương ứng.

|  |
| --- |
| unsigned int i = 65535;  unsigned short = 2767;  unsigned char = 255; |

Đối với tất cả các kiểu, ngoại trừ char, phiên bản có dấu được cho là mặc định nếu phần có dấu hoặc không dấu bị bỏ qua. Kiểu char tạo thành một kiểu ký tự thứ ba, khác với signed char và unsigned char, và sự có dấu (hoặc không) phụ thuộc vào nền tảng.

Có các loại hằng số số nguyên khác nhau (được gọi là chữ số trong ngôn ngữ C) có thể được viết theo các cơ số khác nhau và có độ rộng khác nhau, dựa trên tiền tố hoặc hậu tố của chúng.

|  |
| --- |
| /\* Các biến sau được khởi tạo với cùng một giá trị: \*/  int d = 42; /\* hằng số thập phân (cơ số 10) \*/  int o = 052; /\* hằng số bát phân (cơ số 8) \*/  int x = 0xaf; /\* hằng số thập lục phân (cơ số 16) \*/  int X = 0XAf; /\*\*/ (chữ cái 'a' đến 'f' (không phân biệt chữ hoa chữ thường) biểu thị từ 10 đến 15) \*/ |

Hằng số thập phân luôn luôn có dấu**(signed)**. Hằng số thập lục phân bắt đầu bằng 0x hoặc 0X và hằng số bát phân bắt đầu chỉ với chữ số 0. Hai loại hằng số sau có thể có dấu**(signed)** hoặc không dấu**(unsigned)** tùy thuộc vào việc giá trị có phù hợp với kiểu có dấu hay không.

|  |
| --- |
| /\* Tiền tố để chỉ độ rộng và dấu \*/  long int i = 0x32; /\* không có tiền tố đại diện cho int hoặc long int \*/  unsigned int ui = 65535u; /\* u hoặc U đại diện cho unsigned int hoặc unsigned long int \*/  long int li = 65536l; /\* l hoặc L đại diện cho long int \*/ |

Nếu không có tiền tố, hằng số có kiểu dữ liệu là kiểu đầu tiên mà giá trị của nó phù hợp, nghĩa là hằng số thập phân lớn hơn INT\_MAX sẽ có kiểu long nếu có thể, hoặc long long nếu không được.

Tập tin tiêu đề <limits.h> mô tả các giới hạn của số nguyên như sau. Các giá trị xác định bởi người thực hiện (implementation-defined) phải có giá trị tuyệt đối lớn hơn hoặc bằng giá trị được hiển thị dưới đây, với cùng dấu.

|  |  |  |
| --- | --- | --- |
| MARCO | Kiểu dữ liệu | Giá trị |
| CHAR\_BIT | smallest object that is not a bit-field (byte) 8 |  |
| SCHAR\_MIN | signed char | -127 / -(27 - 1) |
| SCHAR\_MAX | signed char | +127 / 27 - 1 |
| UCHAR\_MAX | unsigned char | 255 / 28 - 1 |
| CHAR\_MIN | char | see below |
| CHAR\_MAX | char | see below |
| SHRT\_MIN | short int | -32767 / -(215 - 1) |
| SHRT\_MAX | short int | +32767 / 215 - 1 |
| USHRT\_MAX | unsigned short int | 65535 / 216 - 1 |
| INT\_MIN | int | -32767 / -(215 - 1) |
| INT\_MAX | int | +32767 / 215 - 1 |
| UINT\_MAX | unsigned int | 65535 / 216 - 1 |
| LONG\_MIN | long int | -2147483647 / -(231 - 1) |
| LONG\_MAX | long int | +2147483647 / 231 - 1 |
| ULONG\_MAX | unsigned long int | 4294967295 / 232 - 1 |
| Version ≥ C99 |  |  |
| LLONG\_MIN | long long int | -9223372036854775807 / -(263 - 1) |
| LLONG\_MAX | long long int | +9223372036854775807 / 263 - 1 |
| ULLONG\_MAX | unsigned long long int | 18446744073709551615 / 264 - 1 |

Nếu giá trị của một đối tượng có kiểu char được mở rộng dấu khi sử dụng trong một biểu thức, giá trị của CHAR\_MIN sẽ giống như SCHAR\_MIN và giá trị của CHAR\_MAX sẽ giống như SCHAR\_MAX. Nếu giá trị của một đối tượng có kiểu char không được mở rộng dấu khi sử dụng trong một biểu thức, giá trị của CHAR\_MIN sẽ là 0 và giá trị của CHAR\_MAX sẽ giống như UCHAR\_MAX.

Phiên bản ≥ C99

Tiêu chuẩn C99 đã thêm một tiêu đề mới, <stdint.h>, chứa các định nghĩa cho các số nguyên có chiều rộng cố định. Xem ví dụ số nguyên chiều rộng cố định để được giải thích sâu hơn.

## PHẦN 3.4 HẰNG SỐ THỰC

Ngôn ngữ C có ba kiểu số thực dấu phẩy động bắt buộc, đó là **float**, **double** và **long double**.

|  |
| --- |
| float f = 0.314f; /\* Hậu tố f hoặc F chỉ định kiểu float \*/  double d = 0.314; /\* Không có hậu tố chỉ định kiểu double \*/  long double ld = 0.314l; /\* Hậu tố l hoặc L chỉ định kiểu long double \*/  /\* Các phần khác nhau trong một định nghĩa số thực là tùy chọn \*/  double x = 1.; /\* Hợp lệ, phần thập phân là tùy chọn \*/  double y = .1; /\* Hợp lệ, phần nguyên là tùy chọn \*/  /\* Chúng cũng có thể được định nghĩa trong dạng ký hiệu khoa học \*/  double sd = 1.2e3; /\* Phần thập phân 1.2 được nhân với 10^3, tức là 1200.0 \*/ |

Tiêu đề <float.h> xác định các giới hạn khác nhau cho các hoạt động của dấu phẩy động Số học dấu phẩy động được xác định triển khai. Tuy nhiên, hầu hết các nền tảng hiện đại (arm, x86, x86\_64, MIPS) đều sử dụng các hoạt động của dấu phẩy động IEEE 754. C cũng có ba loại dấu phẩy động phức tạp tùy chọn bắt nguồn từ loại trên.

## PHẦN 3.5 CHUỖI KÍ TỰ

Một chuỗi ký tự trong ngôn ngữ C là một chuỗi các ký tự, kết thúc bằng một ký tự số không (literal zero)

|  |
| --- |
| char\* str = "hello, world"; /\* chuỗi ký tự nguyên mẫu (string literal) \*/  /\* chuỗi ký tự nguyên mẫu có thể được sử dụng để khởi tạo mảng \*/  char a1[] = "abc"; /\* a1 là mảng char[4] chứa {'a','b','c','\0'} \*/  char a2[4] = "abc"; /\* tương tự a1 \*/  char a3[3] = "abc"; /\* a3 là mảng char[3] chứa {'a','b','c'}, thiếu ký tự '\0' \*/ |

Chuỗi ký tự nguyên mẫu trong C **không thể sửa đổi**, và việc cố gắng thay đổi giá trị của chúng sẽ dẫn đến hành vi không xác định. Chuỗi ký tự nguyên mẫu thường được lưu trữ trong bộ nhớ chỉ đọc (read-only memory), chẳng hạn như phần .rodata, và mọi cố gắng sửa đổi giá trị của chúng có thể dẫn đến kết quả không mong muốn.

|  |
| --- |
| char\* s = "foobar";  s[0] = 'F'; /\* hành vi không xác định \*/  /\* Thực hành tốt là đánh dấu chuỗi nguyên mẫu là hằng số bằng cách sử dụng const \*/  char const s1 = "foobar";  s1[0] = 'F'; /\* lỗi biên dịch! \*/ |

Nhiều chuỗi ký tự được nối vào thời gian biên dịch, có nghĩa là bạn có thể viết cấu trúc như thế này.

|  |
| --- |
| Phiên bản trước C99  /\* Chỉ có thể nối hai chuỗi ký tự hẹp (narrow) hoặc hai chuỗi ký tự rộng (wide) \*/  char\* s = "Hello, " "World";  Phiên bản C99 trở lên  /\* Từ phiên bản C99 trở đi, có thể nối nhiều hơn hai chuỗi \*/  /\* Quá trình nối chuỗi phụ thuộc vào triển khai cụ thể \*/  char\* s1 = "Hello" ", " "World";  /\* Một số cách sử dụng phổ biến là nối chuỗi định dạng (format strings) \*/  char\* fmt = "%" PRId16; /\* Macro PRId16 được định nghĩa từ C99 \*/ |

Chuỗi ký tự nguyên mẫu, tương tự như hằng số ký tự, hỗ trợ các bộ ký tự khác nhau.

|  |
| --- |
| /\* Chuỗi ký tự thông thường, kiểu char[] \*/  char\* s1 = "abc";  /\* Chuỗi ký tự rộng (wide character), kiểu wchar\_t[] \*/  *wchar\_t*\* s2 = L"abc";  Phiên bản C11 trở lên  /\* Chuỗi ký tự UTF-8, kiểu char[] \*/  char\* s3 = u8"abc";  /\* Chuỗi ký tự rộng 16-bit, kiểu char16\_t[] \*/  *char16\_t*\* s4 = u"abc";  /\* Chuỗi ký tự rộng 32-bit, kiểu char32\_t[] \*/  *char32\_t*\* s5 = U"abc"; |

# TOÁN TỬ

Toán tử trong ngôn ngữ lập trình là một ký hiệu cho biết trình biên dịch hoặc trình thông dịch thực hiện một phép toán, phép toán quan hệ hoặc logic cụ thể và tạo ra kết quả cuối cùng

Ngôn ngữ C có nhiều toán tử mạnh mẽ. Nhiều toán tử trong C là toán tử nhị phân, có nghĩa là chúng có hai toán hạng. Ví dụ, trong biểu thức a / b, / là một toán tử nhị phân nhận hai toán hạng (a, b). Ngoài ra, còn có một số toán tử một ngôi (unary operators) chỉ yêu cầu một toán hạng (ví dụ: ~, ++), và chỉ có một toán tử 3 ngôi duy nhất ?

## PHẦN 4.1 TOÁN TỬ QUAN HỆ

Các toán tử quan hệ trong ngôn ngữ C kiểm tra xem một mối quan hệ cụ thể giữa hai toán hạng có đúng hay không. Kết quả được đánh giá là 1 (đúng) hoặc 0 (sai). Kết quả này thường được sử dụng để ảnh hưởng đến luồng điều khiển (thông qua câu lệnh if, while, for), nhưng cũng có thể được lưu trữ trong các biến.

### Bằng “==”

Kiểm tra xem các toán hạng được cung cấp có bằng nhau không.

|  |
| --- |
| 1 == 0; /\* đánh giá thành 0.(Sai) \*/  1 == 1; /\* đánh giá thành 1.(Đúng) \*/  int x = 5;  int y = 5;  int \*xptr = &x, yptr = &y;  xptr == yptr; /\* đánh giá thành 0, các toán hạng giữ các địa chỉ vùng nhớ khác nhau.(Sai) \*/  \*xptr == yptr; /\* đánh giá thành 1, các toán hạng trỏ tới các vị trí vùng nhớ chứa cùng một giá trị.(Đúng) \*/ |

**Chú ý:** Không nên nhầm lẫn toán tử này với toán tử gán “=” !!!

### Không bằng (Khác) “!=”

Kiểm tra xem các toán hạng được cung cấp có bằng nhau không

|  |
| --- |
| 1 == 0; /\* đánh giá thành 0.(Sai) \*/  1 == 1; /\* đánh giá thành 1.(Đúng) \*/  int x = 5;  int y = 5;  int \*xptr = &x, yptr = &y;  xptr == yptr; /\* đánh giá thành 0, các toán hạng giữ các địa chỉ vùng nhớ khác nhau.(Sai) \*/  \*xptr == yptr; /\* đánh giá thành 1, các toán hạng trỏ tới các vị trí vùng nhớ chứa cùng một giá trị.(Đúng) \*/ |

Toán tử này trả về kết quả ngược lại với kết quả của toán tử **bằng bằng** (==).

### Not “!”

Kiểm tra xem một đối tượng có bằng 0 hay không.

Toán tử "!" cũng có thể được sử dụng trực tiếp với một biến như sau:

|  |
| --- |
| !someVal |

Điểu này có tác dụng tương tự như

|  |
| --- |
| someVal == 0 |

### So sánh lớn hơn “>”

Kiểm tra xem toán hạng bên trái có giá trị lớn hơn toán hạng bên phải hay không

|  |
| --- |
| 5 > 4 /\* trả về 1.(Đúng) \*/  4 > 5 /\* trả về 0.(Sai) \*/   1. > 4 /\* trả về 0.(Sai) \*/ |

### 4.1.5 So sánh bé hơn “<”

Kiểm tra xem toán hạng bên trái có giá trị bé hơn toán hạng bên phải hay không

|  |
| --- |
| 5 < 4 /\* trả về 0.(Sai) \*/  4 < 5 /\* trả về 1.(Đúng) \*/   1. < 4 /\* trả về 0.(Sai) \*/ |

### 4.1.6 So sánh lớn hơn hoặc bằng “>=”

Kiểm tra xem toán hạng bên trái có giá trị lớn hơn hoặc bằng toán hạng bên phải hay không

|  |
| --- |
| 5 >= 4 /\* trả về 1.(Đúng) \*/  4 >= 5 /\* trả về 0.(Sai) \*/   1. >= 4 /\* trả về 1.(Đúng) \*/ |

### 4.1.7 So sánh bé hơn hoặc bằng “<=”

Kiểm tra xem toán hạng bên trái có giá trị bé hơn hoặc bằng toán hạng bên phải hay không

|  |
| --- |
| 5 <= 4 /\* trả về 0.(Sai) \*/  4 <= 5 /\* trả về 1.(Đúng) \*/  4 <= 4 /\* trả về 1.(Đúng) \*/ |

## PHẦN 4.2 TOÁN TỬ ĐIỀU KIỆN/ TOÁN TỬ 3 NGÔI

Toán tử 3 ngôi "?" trong ngôn ngữ C được sử dụng để thực hiện một phép toán 3 ngôi (ternary operation). Nó đánh giá toán hạng đầu tiên và nếu giá trị kết quả khác 0, nó đánh giá toán hạng thứ hai. Ngược lại, nếu giá trị kết quả bằng 0, nó đánh giá toán hạng thứ ba.

|  |
| --- |
| a = b ? c : d;  /\*Tương đương với\*/  if (b)      a = c;  else      a = d; |

Đoạn mã sau đây mô tả cú pháp của toán tử 3 ngôi: **điều kiện ? kết quả nếu đúng : kết quả nếu sai**. Mỗi giá trị có thể là kết quả của một biểu thức đã được đánh giá.

|  |
| --- |
| int x = 5;      int y = 42;      printf("%i, %i\n", 1 ? x : y, 0 ? x : y); /\* Kết quả "5, 42" \*/ |

Toán tử điều kiện có thể được lồng vào nhau. Ví dụ: đoạn mã sau xác định số lớn hơn trong ba số:

|  |
| --- |
| big= a > b  ? (a > c ? a : c)              : (b > c ? b : c); |

Ví dụ sau ghi số nguyên chẵn vào một tệp và số nguyên lẻ vào tệp khác

|  |
| --- |
| #include<stdio.h>  int main()  {  *FILE* \*even, \*odds;      int n = 10;  *size\_t* k = 0;      even = fopen("even.txt", "w");      odds = fopen("odds.txt", "w");      for(k = 1; k < n + 1; k++)      {          k%2==0 ? fprintf(even, "\t%5d\n", k)                  : fprintf(odds, "\t%5d\n", k);      }      fclose(even);      fclose(odds);      return 0;  } |

Toán tử điều kiện liên kết từ phải sang trái. Hãy xem xét những điều sau đây:

|  |
| --- |
| exp1 ? exp2 : exp3 ? exp4 : exp5 |

Vì liên kết từ phải sang trái, biểu thức trên được đánh giá là:

|  |
| --- |
| exp1 ? exp2 : ( exp3 ? exp4 : exp5 ) |

## PHẦN 4.3 TOÁN TỬ BITWISE

Toán tử bitwise có thể được sử dụng để thực hiện thao tác cấp độ bit trên các biến. Dưới đây là danh sách tất cả sáu toán tử bitwise được hỗ trợ trong C:

|  |  |
| --- | --- |
| Kí hiệu | Toán tử |
| & | Toán tử AND |
| | | Toán tử OR |
| ^ | Toán tử XOR |
| ~ | Toán tử phủ định NOT |
| << | Toán tử dịch trái |
| >> | Toán tử dịch phải |

Chương trình sau đây minh họa việc sử dụng tất cả các toán tử bitwise:

|  |
| --- |
| #include <stdio.h>  int main()  {      unsigned int a = 29; /\* 29 = 0001 1101 \*/      unsigned int b = 48; /\* 48 = 0011 0000 \*/      int c = 0;      c = a & b; /\* 32 = 0001 0000 \*/      printf("%d & %d = %d\n", a, b, c );      c = a | b; /\* 61 = 0011 1101 \*/      printf("%d | %d = %d\n", a, b, c );      c = a ^ b; /\* 45 = 0010 1101 \*/      printf("%d ^ %d = %d\n", a, b, c );      c = ~a; /\* -30 = 1110 0010 \*/      printf("~%d = %d\n", a, c );      c = a << 2; /\* 116 = 0111 0100 \*/      printf("%d << 2 = %d\n", a, c );      c = a >> 2; /\* 7 = 0000 0111 \*/      printf("%d >> 2 = %d\n", a, c );      return 0;  } |

**Kết quả:**

|  |
| --- |
| **29 & 48 = 16**  **29 | 48 = 61**  **29 ^ 48 = 45**  **~29 = -30**  **29 << 2 = 116**  **29 >> 2 = 7** |

Các phép toán bitwise với kiểu dữ liệu có dấu nên được sử dụng cẩn thận vì bit dấu của kiểu dữ liệu đó có ý nghĩa riêng. Có các hạn chế đặc biệt áp dụng cho các toán tử dịch chuyển (shift operators):

* Dịch trái một bit 1 vào bit dấu của kiểu dữ liệu có dấu được coi là sai và dẫn đến hành vi không xác định. Điều này có nghĩa rằng kết quả của phép dịch như vậy không được xác định rõ ràng và có thể khác nhau tùy thuộc vào trình biên dịch và nền tảng.
* Dịch phải một giá trị âm (với bit dấu được đặt là 1) được xác định bởi từng trình biên dịch và do đó không có tính chất di động. Điều này có nghĩa rằng hành vi của phép dịch như vậy có thể khác nhau giữa các phiên bản khác nhau của ngôn ngữ C và không được đảm bảo di động trên các nền tảng hoặc trình biên dịch khác nhau.
* Nếu giá trị của toán hạng phải của toán tử dịch chuyển là số âm hoặc lớn hơn hoặc bằng độ rộng của toán hạng trái được gán, hành vi là không xác định. Điều này có nghĩa là kết quả của phép dịch như vậy không được chỉ định và có thể cho ra kết quả không mong đợi hoặc không nhất quán.

Để đảm bảo tính di động và tránh hành vi không xác định, thường khuyến nghị sử dụng kiểu dữ liệu không dấu khi thực hiện các phép toán bitwise liên quan đến dịch chuyển.

**Masking:**

Masking (hoặc còn gọi là áp dụng mặt nạ) đề cập đến quá trình trích xuất các bit mong muốn từ một biến (hoặc biến đổi các bit mong muốn trong biến) bằng cách sử dụng các phép toán logic bitwise. Toán hạng (một hằng số hoặc biến) được sử dụng để thực hiện việc áp dụng mặt nạ được gọi là mặt nạ (mask).

Trong quá trình masking, mặt nạ là một giá trị có các bit được thiết lập một cách đặc biệt để thực hiện việc trích xuất hoặc biến đổi các bit trong biến gốc. Khi thực hiện phép toán bitwise AND (&) giữa biến gốc và mặt nạ, chỉ có các bit tương ứng mà mặt nạ đặt thành 1 và các bit còn lại trong biến gốc sẽ bị xóa đi (thiết lập thành 0). Điều này cho phép trích xuất hoặc biến đổi các bit cần thiết trong biến gốc và bỏ qua các bit không mong muốn.

Mặt nạ được sử dụng theo nhiều cách khác nhau

* Để quyết định mẫu bit của một biến số nguyên.
* Để sao chép một phần của mẫu bit đã cho sang một biến mới, trong khi phần còn lại của biến mới được điền bằng 0 (sử dụng bitwise AND)
* Để sao chép một phần của mẫu bit đã cho sang một biến mới, trong khi phần còn lại của biến mới được lấp đầy bằng 1 (sử dụng OR theo bit).
* Để sao chép một phần của mẫu bit đã cho sang một biến mới, trong khi phần còn lại của mẫu bit ban đầu được đảo ngược trong biến mới (sử dụng OR loại trừ bit).

Hàm sau sử dụng mặt nạ để hiển thị mẫu bit của một biến:

|  |
| --- |
| #include <limits.h>  void bit\_pattern(int *u*)  {      int i, x, word;      unsigned mask = 1;      word = CHAR\_BIT \* sizeof(int);      mask = mask << (word - 1); /\* shift 1 to the leftmost position \*/      for(i = 1; i <= word; i++)      {          x = (*u* & mask) ? 1 : 0; /\* identify the bit \*/          printf("%d", x); /\* print bit value \*/          mask >>= 1; /\* shift mask to the right by 1 bit \*/      }  } |

## PHẦN 4.4 HOẠT ĐỘNG NGẮN MẠCH CỦA TOÁN TỬ LOGIC

## PHẦN 4.5 TOÁN TỬ DẤU PHẨY

Toán tử dấu phẩy (,) cho phép bạn đánh giá nhiều biểu thức ở bất kỳ nơi nào cho phép một biểu thức duy nhất. Toán tử dấu phẩy đánh giá toán hạng bên trái, sau đó là toán hạng bên phải, rồi trả về kết quả của toán hạng bên phải.

|  |
| --- |
| int x = 42, y = 42;  printf("%i\n", (x \*= 2, y)); /\* Outputs "42". \*/ |

Toán tử dấu phẩy giới thiệu một điểm thứ tự giữa các toán hạng của nó.

**Lưu ý** rằng dấu phẩy được sử dụng trong các hàm gọi các đối số riêng biệt đó **KHÔNG** phải là toán tử dấu phẩy, thay vào đó, nó được gọi là dấu phân cách khác với toán tử dấu phẩy. Do đó, nó không có các thuộc tính của toán tử dấu phẩy.

Lệnh gọi printf() ở trên chứa cả toán tử dấu phẩy và dấu phân cách.

|  |
| --- |
| printf("%i\n", (x \*= 2, y)); /\* Outputs "42". \*/  /\*           ^        ^ Đây là toán tử dấu phẩy \*/  /\*           Đây là dấu phân cách \*/ |

Toán tử dấu phẩy thường được sử dụng trong phần khởi tạo cũng như trong phần cập nhật của vòng lặp for. Ví dụ:

|  |
| --- |
| for(k = 1; k < 10; printf("*\%*d\\n", k), k += 2); /\*outputs the odd numbers below 9/\*  /\* outputs sum to first 9 natural numbers \*/  for(sumk = 1, k = 1; k < 10; k++, sumk += k)      printf("*\%*5d*\%*5d\\n", k, sumk); |

## PHẦN 4.6 TOÁN TỬ SỐ HỌC

### 4.6.1 Số học cơ bản

Trả về một giá trị là kết quả của việc áp dụng toán hạng bên trái cho toán hạng bên phải, sử dụng phép toán liên quan. Các quy tắc giao hoán toán học thông thường được áp dụng (nghĩa là phép cộng và phép nhân có tính chất giao hoán, phép trừ, phép chia và mô đun thì không).

### 4.6.2 Toán tử cộng

Toán tử cộng (+) được sử dụng để cộng hai toán hạng lại với nhau. Ví dụ:

|  |
| --- |
| #include<stdio.h>  int main(){      int a = 5;      int b = 7;      int c = a + b;      printf("%d + %d = %d",a,b,c);      return 0;  } |

**Kết quả**

|  |
| --- |
| **5 + 7 = 12** |

### 4.6.3 Toán tử trừ

Toán tử trừ được sử dụng để trừ toán hạng thứ 2 cho toán hạng thứ nhất. Ví dụ

|  |
| --- |
| #include<stdio.h>  int main(){      int a = 5;      int b = 7;      int c = a - b;      printf("%d - %d = %d",a,b,c);      return 0;  } |

**Kết quả**

|  |
| --- |
| **5 - 7 = -2** |

### 4.6.4 Toán tử nhân

Toán tử nhân (\*) được sử dụng để nhân cả hai toán hạng. Ví dụ:

|  |
| --- |
| #include<stdio.h>  int main(){      int a = 5;      int b = 7;      int c = a \* b;      printf("%d \* %d = %d",a,b,c);      return 0;  } |

**Kết quả**

|  |
| --- |
| **5 \* 7 = 35** |

**Đừng nhầm lẫn với con trỏ (\*)**

### 4.6.5 Toán tử chia

Toán tử chia (/) chia toán hạng thứ nhất cho toán hạng thứ hai. Nếu cả hai toán hạng của phép chia là số nguyên, nó sẽ trả về một giá trị nguyên và loại bỏ phần còn lại (sử dụng toán tử modulo % để tính toán và lấy phần còn lại).

Nếu một trong các toán hạng là một giá trị dấu chấm động, thì kết quả là một phép tính gần đúng của phân số.

**Ví dụ:**

|  |
| --- |
| #include<stdio.h>  int main(){      int a = 19 / 2 ;      int b = 18 / 2 ;      int c = 255 / 2;      int d = 44 / 4 ;      double e = 19 / 2.0 ;      double f = 18.0 / 2 ;      double g = 255 / 2.0;      double h = 45.0 / 4 ;      printf("19 / 2 = %d\n", a);      printf("18 / 2 = %d\n", b);      printf("255 / 2 = %d\n", c);      printf("44 / 4 = %d\n", d);      printf("19 / 2.0 = %g\n", e);      printf("18.0 / 2 = %g\n", f);      printf("255 / 2.0 = %g\n", g);      printf("45.0 / 4 = %g\n", h);  } |

**Kết quả**

|  |
| --- |
| **19 / 2 = 9**  **18 / 2 = 9**  **255 / 2 = 127**  **44 / 4 = 11**  **19 / 2.0 = 9.5**  **18.0 / 2 = 9**  **255 / 2.0 = 127.5**  **45.0 / 4 = 11.25** |

**Chú ý: Ép kiểu**

|  |
| --- |
| #include<stdio.h>  int main(){      int a = 9;      int b = 2;      float c;      //c = (float) a / b;      //c = 9.0 / 2;      c = 9 / 2.0;      printf("%d / %d = %f",a,b,c);  } |

**Kết quả**

|  |
| --- |
| **9 / 2 = 4.500000** |

### 4.6.6 Toán tử modulo (Phép chia lấy dư)

Toán tử modulo (%) chỉ nhận toán hạng số nguyên và được sử dụng để tính phần còn lại sau khi toán hạng thứ nhất được chia cho toán hạng thứ hai. Toán tử mô đun (còn được gọi là toán tử chia lấy phần dư) là một toán tử trả về phần còn lại sau khi thực hiện phép chia số nguyên.

Ví dụ:

|  |
| --- |
| #include <stdio.h>  int main (void) {      int a = 25 % 2; /\* a holds value 1 \*/      int b = 24 % 2; /\* b holds value 0 \*/      int c = 155 % 5; /\* c holds value 0 \*/      int d = 49 % 25; /\* d holds value 24 \*/      printf("25 *%* 2 = %d\n", a); /\* Will output "25 % 2 = 1" \*/      printf("24 *%* 2 = %d\n", b); /\* Will output "24 % 2 = 0" \*/      printf("155 *%* 5 = %d\n", c); /\* Will output "155 % 5 = 0" \*/      printf("49 *%* 25 = %d\n", d); /\* Will output "49 % 25 = 24" \*/      return 0;  } |

### 4.6.7 Toán tử tăng giảm

Các toán tử tăng (a++) và giảm (a--) khác nhau ở chỗ chúng thay đổi giá trị của biến mà bạn áp dụng chúng mà không có toán tử gán. Bạn có thể sử dụng toán tử **tăng** và **giảm** trước hoặc sau biến. Vị trí của toán tử thay đổi thời gian tăng/giảm giá trị thành trước hoặc sau khi gán nó cho biến.

Ví dụ:

|  |
| --- |
| #include <stdio.h>  int main(void)  {      int a = 1;      int b = 4;      int c = 1;      int d = 4;      a++;      printf("a = %d\n",a); /\* Sẽ in ra "a = 2" \*/      b--;      printf("b = %d\n",b); /\* Sẽ in ra "b = 3" \*/      if (++c > 1) { /\* c được tăng thêm 1 trước khi được so sánh trong điều kiện \*/          printf("This will print\n"); /\* Dòng này sẽ được in ra \*/      } else {          printf("This will never print\n"); /\* Dòng này sẽ không được in ra \*/      }      if (d-- < 4) { /\* d được giảm sau khi được so sánh \*/          printf("This will never print\n"); /\* Dòng này sẽ không được in ra \*/      } else {          printf("This will print\n"); /\* Dòng này sẽ được in ra \*/      }  } |

Kết quả

|  |
| --- |
| a = 2  b = 3  This will print  This will print |

Như ví dụ cho c và d cho thấy, cả hai toán tử đều có hai dạng, là ký hiệu tiền tố và ký hiệu hậu tố. Cả hai đều có tác dụng giống nhau trong việc tăng (++) hoặc giảm (--) biến, nhưng khác nhau về giá trị mà chúng trả về: các phép toán tiền tố thực hiện thao tác trước rồi trả về giá trị, trong khi các phép toán hậu tố trước tiên xác định giá trị được trả lại, và sau đó thực hiện thao tác.

Do hành vi có khả năng phản trực giác này, việc sử dụng các toán tử tăng/giảm bên trong các biểu thức đang gây tranh cãi.

## PHẦN 4.7 TOÁN TỬ TRUY CẬP

Các toán tử truy cập thành viên (dấu chấm . và mũi tên ->) được sử dụng để truy cập một thành viên của cấu trúc(struct).

### 4.7.1 Thuộc tính của đối tượng

Được đánh giá thành giá trị, chỉ định đối tượng là thành viên của đối tượng được truy cập.

|  |
| --- |
| struct *MyStruct*  {      int x;      int y;  };  struct *MyStruct* myObject;  myObject.x = 42;  myObject.y = 123;  printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /\* Outputs ".x = 42, .y = 123". \*/ |

### 4.7.2 Thành viên của đối tượng được trỏ tới

Toán tử mũi tên (->) được sử dụng để tiện lợi hóa quá trình giải tham chiếu và truy cập thành viên của một cấu trúc. Thay vì viết (\*x).y để truy cập thành viên y của đối tượng được giải tham chiếu bởi con trỏ x, chúng ta có thể sử dụng x->y. Điều này giúp làm rõ hơn cách thao tác và làm cho mã nguồn dễ đọc hơn, đặc biệt là khi có các con trỏ cấu trúc lồng nhau.

|  |
| --- |
| #include <stdio.h>  int main(){  struct *MyStruct*  {      int x;      int y;  };  struct *MyStruct* myObject;  struct *MyStruct* \*p = &myObject;  p->x = 42;  p->y = 123;  printf(".x = %i, .y = %i\n", p->x, p->y); /\* Kết quả ".x = 42, .y = 123". \*/  printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /\* Kết quả cũng sẽ là ".x = 42, .y = 123". \*/  } |

### 4.7.3 Lấy địa chỉ

Toán tử & (unary address of operator) trong ngôn ngữ C được sử dụng để lấy địa chỉ của một biến. Khi áp dụng toán tử & cho một biến, nó trả về một con trỏ chứa địa chỉ của biến đó.

**Ví dụ:**

|  |
| --- |
| #include <stdio.h>  int main(){  int x = 3;  int \*p = &x;  printf("%p = %p\n", &x, (p)); /\* Xuất ra "A = A",Với A là một giá trị do triển khai cụ thể quy định ở đây là giá trị địa chỉ ngẫu nhiên.\*/  } |

**Kết quả**

|  |
| --- |
| 0061FF18 = 0061FF18 |

### 4.7.4 Dereference

The unary \* operator dereferences a pointer. It evaluates into the lvalue resulting from dereferencing the pointer that results from evaluating the given expression.

|  |
| --- |
| int x = 42;  int \*p = &x;  printf("x = %d, \*p = %d\n", x, \**p*); /\* Xuất ra "x = 42, \*p = 42". \*/  \*p = 123;  printf("x = %d, \*p = %d\n", x, \**p*); /\* Xuất ra "x = 123, \*p = 123". \*/ |

### 4.7.5 Chỉ số mảng

Trong C, cú pháp indexing (chỉ số mảng) thực tế là một cách viết tắt cho việc thực hiện phép cộng con trỏ và sau đó giải tham chiếu. Biểu thức dạng **a[i]** tương đương với **\*(a + i)**, trong đó a là một con trỏ và i là chỉ số hoặc độ lệch.

Trong C, mảng sẽ tự động chuyển thành con trỏ tới phần tử đầu tiên của nó trong hầu hết các ngữ cảnh. Khi bạn sử dụng cú pháp chỉ số **a[i],** trình biên dịch sẽ coi a như một con trỏ tới phần tử đầu tiên của mảng và thực hiện phép cộng con trỏ với độ lệch **i**. Sau đó, con trỏ kết quả được giải tham chiếu sử dụng toán tử **\*** để truy cập vào giá trị tại vị trí đó trong bộ nhớ.

|  |
| --- |
| int arr[] = { 1, 2, 3, 4, 5 };  printf("arr[2] = %i\n", arr[2]); /\* Xuất ra "arr[2] = 3". \*/ |

### 4.7.6 Khả năng hoán đổi của chỉ số mảng

Thêm một con trỏ vào một số nguyên là một phép toán giao hoán (nghĩa là thứ tự của các toán hạng không làm thay đổi kết quả) vì vậy con trỏ + số nguyên == số nguyên + con trỏ.

Hệ quả của điều này là arr[3] và 3[arr] là tương đương.

|  |
| --- |
| printf("3[arr] = %i\n", 3[arr]); /\* Xuất ra "3[arr] = 4". \*/ |

## PHẦN 4.8 TOÁN TỬ SIZEOF

### 4.8.1 With a type as operand

Đánh giá kết quả là kích thước trong byte, của kiểu size\_t, của đối tượng thuộc kiểu đã cho. Yêu cầu có dấu ngoặc đơn xung quanh kiểu dữ liệu.

|  |
| --- |
| printf("%zu\n", sizeof(int)); /\* Hợp lệ, in ra kích thước của một đối tượng int,                                 \*có thể thay đổi theo nền tảng. \*/  printf("%zu\n", sizeof *int*); /\* Không hợp lệ, các kiểu dữ liệu truyền vào cần được                                \*bao bọc bởi dấu ngoặc đơn! \*/ |

### 4.8.2 With an expression as operand

Đánh giá kết quả là kích thước trong byte, kiểu size\_t, của các đối tượng có cùng kiểu dữ liệu với biểu thức được cung cấp. Biểu thức chính nó không được đánh giá. Dấu ngoặc đơn không bắt buộc; tuy nhiên, vì biểu thức được cung cấp phải là toán tử một ngôi, nên luôn tốt nhất là sử dụng chúng.

|  |
| --- |
| char ch = 'a';  printf("%zu\n", sizeof(ch)); /\* Hợp lệ, sẽ in ra kích thước của một đối tượng char,                                \* luôn là 1 cho mọi nền tảng. \*/  printf("%zu\n", sizeof *ch*); /\* Hợp lệ, sẽ in ra kích thước của một đối tượng char,                               \*luôn là 1 cho mọi nền tảng. \*/ |

## PHẦN 4.9 TOÁN TỬ ÉP KIỂU

Thực hiện một chuyển đổi rõ ràng sang kiểu dữ liệu đã chỉ định từ giá trị thu được sau khi đánh giá biểu thức cung cấp.

|  |
| --- |
| int x = 3;  int y = 4;  printf("%f\n", (double)x / y); /\* In ra "0.750000". \*/ |

Ở đây, giá trị của x được chuyển đổi thành kiểu double, phép chia cũng chuyển đổi giá trị của y thành kiểu double và kết quả của phép chia, một giá trị double, được truyền vào hàm printf để in ra màn hình.

## PHẦN 4.10 TOÁN TỬ GỌI HÀM

Toán tử gọi hàm trong ngôn ngữ lập trình yêu cầu toán hạng đầu tiên phải là một con trỏ tới hàm (function pointer) hoặc một biểu thức chỉ định hàm (function designator). Đối tượng này xác định hàm mà ta muốn gọi. Các toán hạng còn lại, nếu có, được gọi là đối số của cuộc gọi hàm. Toán tử này đánh giá thành giá trị trả về từ việc gọi hàm thích hợp với các đối số tương ứng.

|  |
| --- |
| int myFunction(int *x*, int *y*)  {      return *x* \* 2 + *y*;  }  int (\*fn)(int, int) = &myFunction;  int x = 42;  int y = 123;  printf("(\*fn)(%i, %i) = %i\n", x, y, (fn)(x, y)); /\* In ra "fn(42, 123) = 207". \*/  printf("fn(%i, %i) = %i\n", x, y, fn(*x*, y)); /\* Một dạng khác: bạn không cần phải giải tham chiếu một cách rõ ràng \*/ |

## PHẦN 4.11 TOÁN TỬ TĂNG/ TOÁN TỬ GIẢM

Các toán tử tăng và giảm tồn tại ở dạng tiền tố và hậu tố.

|  |
| --- |
| int a = 1;  int b = 1;  int tmp = 0;  tmp = ++a; /\* tăng a lên một đơn vị và trả về giá trị mới; a == 2, tmp == 2 \*/  tmp = a++; /\* tăng a lên một đơn vị nhưng trả về giá trị cũ; a == 3, tmp == 2 \*/  tmp = --b; /\* giảm b đi một đơn vị và trả về giá trị mới; b == 0, tmp == 0 \*/  tmp = b--; /\* giảm b đi một đơn vị nhưng trả về giá trị cũ; b == -1, tmp == 0 \*/ |

Hiểu đơn giản

Toán tử ++: Tăng giá trị lên 1 đơn vị.

**• x++:** thực hiện lệnh trước rồi mới tăng x lên 1 đơn vị.

**• ++x:** tăng x lên 1 đơn vị rồi mới thực hiện lệnh.

Toán tử --: Giảm giá trị đi 1 đơn vị.

**• x--:** thực hiện lệnh trước rồi mới giảm x đi 1 đơn vị.

**• --x:** giảm x đi 1 đơn vị rồi mới thực hiện lệnh.

Lưu ý rằng các phép toán số học không tạo ra điểm chuỗi (sequence points), do đó, một số biểu thức sử dụng toán tử tăng (++) hoặc giảm (--) có thể dẫn đến hành vi không xác định.

## PHẦN 4.12 TOÁN TỬ GÁN

Gán giá trị của toán hạng bên phải cho vị trí lưu trữ được đặt tên bởi toán hạng bên trái và trả về giá trị.

|  |
| --- |
| int x = 5; /\* Biến x chứa giá trị 5. Trả về 5. /  char y = 'c'; /\* Biến y chứa giá trị 99. Trả về 99                 \*(vì ký tự 'c' được đại diện bởi 99 trong bảng ASCII). \*/  float z = 1.5; /\* Biến z chứa giá trị 1.5. Trả về 1.5. \*/  char const\* s = "foo"; /\* Biến s chứa địa chỉ của ký tự đầu tiên trong chuỗi 'foo'. \*/ |

Một số phép tính số học có toán tử gán phức hợp.

|  |
| --- |
| a += b /\* tương đương với: a = a + b \*/  a -= b /\* tương đương với: a = a - b \*/  a = b /\* tương đương với: a = a \* b \*/  a /= b /\* tương đương với: a = a / b \*/  a %= b /\*tương đương với: a = a % b \*/  a &= b /\* tương đương với: a = a & b \*/  a |= b /\* tương đương với: a = a | b \*/  a ^= b /\* tương đương với: a = a ^ b \*/  a <<= b /\* tương đương với: a = a << b \*/  a >>= b /\*tương đương với: a = a >> b \*/ |

Một tính năng quan trọng của các toán tử gán kết hợp này là biểu thức ở phía bên trái (a) chỉ được đánh giá một lần. Ví dụ, nếu p là một con trỏ:

|  |
| --- |
| \*p += 5; |

Trong trường hợp này, giá trị của biểu thức \*p sẽ chỉ được tính toán một lần và sau đó, giá trị mới được gán vào vị trí mà con trỏ p trỏ tới. Điều này giúp tránh các lỗi không xác định hoặc không mong muốn khi giá trị của biểu thức thay đổi trong quá trình đánh giá.

|  |
| --- |
| \*p = \*p + 5 |

Cũng cần lưu ý rằng kết quả của một phép gán chẳng hạn như a = b là cái được gọi là giá trị. Do đó, phép gán thực sự có một giá trị mà sau đó có thể được gán cho một biến khác. Điều này cho phép xâu chuỗi các phép gán để đặt nhiều biến trong một câu lệnh.

Giá trị này có thể được sử dụng trong các biểu thức kiểm soát của câu lệnh if (hoặc vòng lặp hoặc câu lệnh chuyển đổi) để bảo vệ một số mã trên kết quả của một biểu thức hoặc lời gọi hàm khác. Ví dụ:

|  |
| --- |
| char \*buffer;  if ((buffer = malloc(1024)) != NULL)  {   /\* thực hiện một số công việc sử dụng buffer \*/      free(buffer);  }  else  {   /\* báo cáo lỗi khi không cấp phát được bộ nhớ \*/  } |

Do đó, cần phải cẩn thận để tránh mắc lỗi đánh máy phổ biến có thể dẫn đến các lỗi bí ẩn.

|  |
| --- |
| int a = 2;  /\* ... \*/  if (a = 1)  /\* Xóa tất cả các tập tin trên ổ cứng của tôi \*/ |

Điều này sẽ dẫn đến kết quả tai hại, vì a = 1 sẽ luôn đánh giá bằng 1 và do đó, biểu thức kiểm soát của câu lệnh if sẽ luôn đúng (đọc thêm về cạm bẫy phổ biến này tại đây). Tác giả gần như chắc chắn có ý định sử dụng toán tử đẳng thức (==) như hình dưới đây:

|  |
| --- |
| int a = 2;  /\* ... \*/  if (a == 1)  /\* Xóa tất cả các tập tin trên ổ cứng của tôi \*/ |

**Tính kết hợp của toán tử**

|  |
| --- |
| int a, b = 1, c = 2;  a = b = c; |

Điều này gán c cho b, trả về b, được gán cho a. Điều này xảy ra bởi vì tất cả các toán tử gán có tính kết hợp phải, điều đó có nghĩa là phép toán ngoài cùng bên phải trong biểu thức được ước tính trước và tiến hành từ phải sang trái.

## PHẦN 4.13 TOÁN TỬ LOGIC

### 4.13.1 Logic AND

Toán tử logic AND (&&) trong C thực hiện phép AND logic giữa hai toán hạng và trả về giá trị 1 nếu cả hai toán hạng đều khác 0. (Có nghĩa là đúng khi cả 2 đều đúng) Toán tử AND logic có kiểu int.

|  |
| --- |
| 0 && 0 /\* Trả về 0. \*/  0 && 1 /\* Trả về 0. \*/  2 && 0 /\* Trả về 0. \*/  2 && 3 /\* Trả về 1. \*/ |

### 4.13.2 Logic OR

Toán tử logic OR (||) trong C thực hiện phép OR logic giữa hai toán hạng và trả về giá trị 1 nếu bất kỳ một trong hai toán hạng là khác không. Toán tử OR logic có kiểu int.

|  |
| --- |
| 0 || 0 /\* Trả về 0. \*/  0 || 1 /\* Trả về 1. \*/  2 || 0 /\* Trả về 1. \*/  2 || 3 /\* Trả về 1. \*/ |

### 4.13.3 Logic NOT

Toán tử logic NOT (!) trong C thực hiện phép phủ định logic của một biểu thức và trả về giá trị 1 nếu biểu thức là sai (bằng 0), và trả về giá trị 0 nếu biểu thức là đúng (khác 0). Toán tử NOT logic có kiểu int.

Các tính chất quan trọng chung cho cả phép toán logic AND (&&) và phép toán logic OR (||) là như sau, và rất quan trọng để hiểu cách chúng hoạt động trong việc đánh giá ngắn gọn (short-circuit evaluation):

Toán hạng bên trái (LHS) được đánh giá hoàn toàn trước khi toán hạng bên phải (RHS) được đánh giá:

* Trong biểu thức A && B, A được đánh giá trước, và chỉ khi A là đúng (khác không) thì B mới được đánh giá. Nếu A là sai (bằng không), B sẽ không được đánh giá vì kết quả tổng thể của phép toán AND đã xác định là sai.
* Trong biểu thức A || B, A được đánh giá trước, và chỉ khi A là sai (bằng không) thì B mới được đánh giá. Nếu A là đúng (khác không), B sẽ không được đánh giá vì kết quả tổng thể của phép toán OR đã xác định là đúng.

Có một điểm chuỗi (sequence point) giữa việc đánh giá toán hạng bên trái và toán hạng bên phải:

* Điểm chuỗi là điểm trong việc thực thi chương trình nơi tất cả các hiệu ứng phụ của các đánh giá trước đó đã được thực hiện. Điểm chuỗi đảm bảo rằng các đánh giá xảy ra theo một thứ tự được xác định, tránh các vấn đề liên quan đến hành vi không xác định (undefined behavior).

Toán hạng bên phải không được đánh giá nếu kết quả của toán hạng bên trái xác định kết quả tổng thể:

* Đánh giá ngắn gọn (short-circuit evaluation) tránh việc thực hiện các tính toán không cần thiết. Ví dụ, trong A && B, nếu A là sai, kết quả của biểu thức tổng thể đã biết là sai bất kể giá trị của B. Do đó, không cần phải đánh giá B.
* Tương tự, trong A || B, nếu A là đúng, kết quả của biểu thức tổng thể đã biết là đúng bất kể giá trị của B. Do đó, không cần phải đánh giá B.

Các tính chất này là rất quan trọng để viết mã hiệu quả và an toàn, đặc biệt khi sử dụng các biểu thức có hiệu ứng phụ tiềm ẩn hoặc các đánh giá phức tạp. Nó cho phép các lập trình viên tối ưu hóa mã của họ và tránh công việc không cần thiết khi đánh giá biểu thức logic.

## PHẦN 4.14 TOÁN HỌC TRÊN CON TRỎ

### 4.14.1 Cộng con trỏ

### 4.14.2 Trừ con trỏ

## PHẦN 4.15 \_Alignof

# BOOLEAN

## PHẦN 5.1 SỬ DỤNG STDBOOL.H

Tiêu sử dụng tệp tiêu đề hệ thống **stdbool.h** cho phép bạn sử dụng **bool** như một kiểu dữ liệu Boolean. **true** tương đương với **1** và **false** tương đương với **0**.

|  |
| --- |
| #include <stdio.h>  #include <stdbool.h>  int main(void) {      bool x = true; /\* tương đương với bool x = 1; \*/      bool y = false; /\* tương đương với bool y = 0; \*/      if (x) /\* Tương đương với if (x != 0) hoặc if (x != false) \*/      {          puts("This will print!"); /\* In ra màn hình chuỗi "This will print!" \*/      }      if (!y) /\* Tương đương với if (y == 0) hoặc if (y == false) \*/      {          puts("This will also print!"); /\* In ra màn hình chuỗi "This will also print!" \*/      }      return 0;  } |

Trong C, từ khóa "bool" chỉ là một tên gọi khác cho kiểu dữ liệu "\_Bool". Kiểu dữ liệu "\_Bool" là một kiểu dữ liệu tích hợp được giới thiệu trong tiêu chuẩn C99 để biểu diễn giá trị boolean, chỉ có hai giá trị có thể xảy ra: đúng (1) và sai (0).

Kiểu dữ liệu "\_Bool" có các quy tắc đặc biệt khi chuyển đổi các số hoặc con trỏ sang nó.

## PHẦN 5.2 SỬ DỤNG #DEFINE

Trong C, mọi phiên bản, các toán tử so sánh thực sự sẽ xem xét bất kỳ giá trị số nguyên nào khác 0 là đúng (true) và giá trị số nguyên 0 là sai (false). Điều này áp dụng cho các biểu thức điều kiện như trong câu lệnh if, while, for và các phép so sánh khác.

Nếu bạn không có sẵn \_Bool hoặc bool như trong C99, bạn có thể mô phỏng một kiểu dữ liệu Boolean trong C bằng cách sử dụng các macro được định nghĩa bằng #define, và bạn có thể vẫn tìm thấy những điều tương tự trong mã nguồn cũ.

|  |
| --- |
| #include <stdio.h>  #define bool int  #define true 1  #define false 0  int main(void) {      bool x = true; /\* Tương đương với int x = 1; \*/      bool y = false; /\* Tương đương với int y = 0; \*/      if (x) /\* Tương đương với if (x != 0) hoặc if (x != false) \*/      {          puts("This will print!"); /\* In ra màn hình chuỗi "This will print!" \*/      }      if (!y) /\* Tương đương với if (y == 0) hoặc if (y == false) \*/      {          puts("This will also print!"); /\* In ra màn hình chuỗi "This will also print!" \*/      }      return 0;  } |

Trong mã nguồn mới, nên tránh định nghĩa những macro như bool, true, và false, vì chúng có thể gây xung đột với việc sử dụng hiện đại của thư viện <stdbool.h>.

## PHẦN 5.3 SỬ DỤNG KIỂU \_BOOL NỘI TẠI(TÍCH HỢP)

Kiểu dữ liệu \_Bool được thêm vào trong tiêu chuẩn C99 và là một kiểu dữ liệu nguyên thuỷ (native) trong ngôn ngữ C. Nó có khả năng lưu trữ các giá trị **0** (để biểu diễn **false**) và **1** (để biểu diễn **true**).

|  |
| --- |
| #include <stdio.h>  int main(void) {      \_Bool x = 1; /\* \_Bool có thể lưu trữ giá trị 0 hoặc 1 \*/      \_Bool y = 0;      if (x) /\* Tương đương với if (x == 1) \*/      {          puts("This will print!"); /\* In ra màn hình chuỗi "This will print!" \*/      }      if (!y) /\* Tương đương với if (y == 0) \*/      {          puts("This will also print!"); /\* In ra màn hình chuỗi "This will also print!" \*/      }      return 0;  } |

Trong ngôn ngữ C, \_Bool là một kiểu số nguyên, nhưng nó có các quy tắc đặc biệt cho việc chuyển đổi từ các kiểu dữ liệu khác khi được sử dụng trong biểu thức. Kết quả của các biểu thức \_Bool tương tự như việc sử dụng các kiểu số nguyên khác trong các biểu thức if. Trong ví dụ sau:

|  |
| --- |
| \_Bool z = X; |

Nếu X có kiểu số học (là bất kỳ loại số nào), thì z sẽ trở thành 0 nếu X bằng 0. Ngược lại, z sẽ trở thành 1.

Nếu X có kiểu con trỏ (kiểu dữ liệu liên quan đến con trỏ), thì z sẽ trở thành 0 nếu X là một con trỏ null (con trỏ không trỏ tới địa chỉ nào). Ngược lại, z sẽ trở thành 1 nếu X là một con trỏ hợp lệ (con trỏ trỏ tới địa chỉ nào đó).

Để sử dụng cách viết đẹp hơn với từ khóa bool, false, và true, bạn nên bao gồm tiêu đề <stdbool.h> trong mã C của mình.

## PHẦN 5.4 SỐ NGUYÊN VÀ CON TRỎ TRONG BIỂU THỨC BOOLEAN

Trong ngôn ngữ C, tất cả các số nguyên hoặc con trỏ đều có thể được sử dụng trong các biểu thức mà được hiểu là "giá trị đúng" (truth value) trong ngữ cảnh Logic.

|  |
| --- |
| int main(int *argc*, char\* *argv*[]) {      if (*argc* % 4) {          puts("arguments number is not divisible by 4");      } else {          puts("argument number is divisible by 4");      }  ... |

Biểu thức argc % 4 sẽ đánh giá thành một trong các giá trị 0, 1, 2 hoặc 3, tùy thuộc vào phần dư khi argc chia cho 4

* Nếu kết quả của argc % 4 là 0, nó sẽ được coi là "sai" (false) trong ngữ cảnh Logic (Boolean), và chương trình sẽ thực thi mã trong khối else, in ra thông báo "argument number is divisible by 4" (số lượng đối số chia hết cho 4).
* Nếu kết quả của argc % 4 là 1, 2 hoặc 3, nó sẽ được coi là "đúng" (true) trong ngữ cảnh Logic (Boolean), và chương trình sẽ thực thi mã trong khối if, in ra thông báo "arguments number is not divisible by 4" (số lượng đối số không chia hết cho 4).

|  |
| --- |
| double\* A = malloc(n \* sizeof(\*A));  if ( !A ) {      perror("allocation problems");      exit(EXIT\_FAILURE);  } |

Trong đoạn mã trên, ta cấp phát bộ nhớ động cho con trỏ A bằng hàm malloc và sau đó kiểm tra xem A có là con trỏ null hay không. Nếu A là con trỏ null, đoạn mã in ra thông báo lỗi và chương trình thoát. Điều này giúp phát hiện và xử lý vấn đề về cấp phát bộ nhớ một cách chính xác.

|  |
| --- |
| char const\* s = ....; /\* some pointer that we receive \*/  if (s != NULL && s[0] != '\0' && isalpha(s[0])) {   printf("this starts well, %c is alphabetic\n", s[0]);  } |

Để kiểm tra điều này, bạn phải quét mã phức tạp trong biểu thức và chắc chắn về tùy chọn toán tử.

|  |
| --- |
| char const\* s = ....; /\* some pointer that we receive \*/  if (s && s[0] && isalpha(s[0])) {   printf("this starts well, %c is alphabetic\n", s[0]);  } |

tương đối dễ nắm bắt: nếu con trỏ hợp lệ, chúng tôi kiểm tra xem ký tự đầu tiên có khác 0 hay không và sau đó kiểm tra xem đó có phải là một chữ cái hay không.

## PHẦN 5.5 XÁC ĐỊNH KIỂU BOOL BẰNG CÁCH SỬ DỤNG TYPEDEF

Thay vì sử dụng các **#define** macros, ta có thể sử dụng **enum** để định nghĩa các hằng số có tên để biểu thị giá trị true và false.

|  |
| --- |
| #if \_\_STDC\_VERSION\_\_ < 199900L  typedef enum { false, true } bool;  /\* Modern C code might expect these to be macros. \*/  # ifndef bool  # define bool bool  # endif  # ifndef true  # define true true  # endif  # ifndef false  # define false false  # endif  #else  # include <stdbool.h>  #endif  /\* Somewhere later in the code ... \*/  bool b = true; |

Điều này cho phép trình biên dịch cho các phiên bản C lịch sử hoạt động, nhưng vẫn tương thích về phía trước nếu mã được biên dịch bằng trình biên dịch C hiện đại.

Để biết thêm thông tin về typedef, hãy xem Typedef, để biết thêm về enum, hãy xem Enumerations

# CHUỖI

Trong ngôn ngữ C, chuỗi (string) không phải là một kiểu dữ liệu tự nhiên như số nguyên hay số thực. Thay vào đó, một chuỗi C được biểu diễn dưới dạng một mảng một chiều các ký tự (character array), kết thúc bằng ký tự null (null-character) là '\0'.

Điều này có nghĩa là một chuỗi C có nội dung là "abc" sẽ có bốn ký tự 'a', 'b', 'c' và '\0'.

Xem ví dụ giới thiệu cơ bản về chuỗi.

## PHẦN 6.1 PHÂN TÁCH TỪ: STRTOK(), STRTOK\_R() VÀ STRTOK\_S()

Hàm **strtok()** trong ngôn ngữ C được sử dụng để chia một chuỗi thành các chuỗi nhỏ hơn hoặc các phần tử con gọi là "**tokens**" bằng cách sử dụng một tập hợp các ký tự phân tách.

**Ví dụ:**

|  |
| --- |
| #include <stdio.h>  #include <string.h>  int main(void)  {      int toknum = 0;      char src[] = "Hello,, world!";      const char delimiters[] = ", !";      char \*token = strtok(src, delimiters);      while (token != NULL)      {          printf("%d: [%s]\n", ++toknum, token);          token = strtok(NULL, delimiters);      }      /\* source is now "Hello\0, world\0\0" \*/  } |

**Kết quả:**

|  |
| --- |
| 1: [Hello]  2: [world] |

Chuỗi ký tự phân tách có thể chứa một hoặc nhiều ký tự phân tách, và bạn có thể sử dụng các chuỗi phân tách khác nhau trong mỗi lần gọi strtok().

Để tiếp tục phân tách chuỗi nguồn mà không truyền lại chuỗi nguồn, bạn nên truyền NULL làm đối số đầu tiên. Nếu truyền lại chuỗi nguồn, lần gọi đầu tiên của **strtok()** sẽ trả về lại token đầu tiên.

Hàm **strtok()** không cấp phát bộ nhớ mới cho các token, mà thay vào đó nó sửa đổi chuỗi nguồn. Điều này có nghĩa là chuỗi nguồn không thể là **const** (không thể là một con trỏ tới chuỗi cố định). Nó cũng có nghĩa là danh tính của ký tự phân tách bị mất (ví dụ: trong ví dụ trên, các dấu , và ! đã bị xóa khỏi chuỗi nguồn và không thể xác định được ký tự phân tách nào được trùng khớp).

Nếu có nhiều ký tự phân tách liên tiếp trong chuỗi nguồn, chúng sẽ được xem như một ký tự phân tách duy nhất; trong ví dụ, dấu phẩy thứ hai sẽ bị bỏ qua.

strtok() không an toàn khi sử dụng đa luồng hoặc đa ngõ ra do sử dụng **bộ đệm tĩnh** trong quá trình phân tích. Điều này có nghĩa là nếu một hàm gọi strtok(), bất kỳ hàm nào mà nó gọi trong khi sử dụng strtok() cũng không thể sử dụng strtok(), và nó không thể được gọi bởi bất kỳ hàm nào đang sử dụng strtok().

Dưới đây là một ví dụ minh họa cho vấn đề do việc strtok() không đảm bảo tính re-entrant (không đảm bảo khả năng sử dụng lại trong đa luồng). Trong môi trường đa luồng, việc gọi strtok() từ nhiều luồng có thể dẫn đến hành vi không mong muốn và làm mất dữ liệu.

|  |
| --- |
| #include <stdio.h>  #include <string.h>  int main(void)  {      char src[] = "1.2,3.5,4.2";      char \*first = strtok(src, ",");      do      {          char \*part;          /\* Nested calls to strtok do not work as desired \*/          printf("[%s]\n", first);          part = strtok(first, ".");          while (part != NULL)          {                  printf(" [%s]\n", part);                  part = strtok(NULL, ".");          }      } while ((first = strtok(NULL, ",")) != NULL);  } |

**Kết quả:**

|  |
| --- |
| [1.2]  [1]  [2] |

Hoạt động dự kiến là vòng lặp do while bên ngoài sẽ tạo ba mã thông báo bao gồm mỗi chuỗi số thập phân ("1.2", "3.5", "4.2"), đối với mỗi mã mà strtok gọi cho vòng lặp bên trong sẽ chia nó thành các chuỗi riêng biệt chuỗi chữ số ("1", "2", "3", "5", "4", "2").

Tuy nhiên, vì strtok không đăng ký lại nên điều này không xảy ra. Thay vào đó, strtok đầu tiên tạo chính xác mã thông báo "1.2\0" và vòng lặp bên trong tạo chính xác mã thông báo "1" và "2". Nhưng sau đó, strtok trong vòng lặp bên ngoài nằm ở cuối chuỗi được sử dụng bởi vòng lặp bên trong và trả về NULL ngay lập tức. Chuỗi con thứ hai và thứ ba của mảng src hoàn toàn không được phân tích.

Các thư viện C tiêu chuẩn không chứa phiên bản an toàn cho luồng hoặc đăng ký lại nhưng một số phiên bản khác thì có, chẳng hạn như POSIX' strtok\_r. Lưu ý rằng trên MSVC, strtok tương đương, strtok\_s an toàn cho luồng.

C11 có một phần tùy chọn, Phụ lục K, cung cấp phiên bản đăng ký lại và an toàn cho luồng có tên strtok\_s. Bạn có thể kiểm tra tính năng này bằng \_\_STDC\_LIB\_EXT1\_\_. Phần tùy chọn này không được hỗ trợ rộng rãi.

Hàm strtok\_s khác với hàm strtok\_r của POSIX ở chỗ bảo vệ chống lưu trữ bên ngoài chuỗi được mã hóa và bằng cách kiểm tra các giới hạn thời gian chạy. Tuy nhiên, trên các chương trình được viết chính xác, strtok\_s và strtok\_r hoạt động giống nhau.

Sử dụng strtok\_s với ví dụ hiện mang lại phản hồi chính xác, như vậy:

## PHẦN 6.2 STRING LITERALS

Chuỗi hằng (string literals) trong C đại diện cho các mảng ký tự có kết thúc bằng ký tự null và có thời gian sống tĩnh (static-duration). Vì có thời gian sống tĩnh, một chuỗi hằng hoặc một con trỏ trỏ tới cùng một mảng gốc có thể an toàn sử dụng trong nhiều trường hợp mà một con trỏ tới mảng tự động không thể. Ví dụ, việc trả về một chuỗi hằng từ một hàm có hành vi xác định rõ ràng:

|  |
| --- |
| const char \*get\_hello() {      return "Hello, World!"; /\* an toàn \*/  } |

Do lưu trữ tĩnh, các chuỗi hằng không thể bị sửa đổi, và mọi cố gắng thay đổi các phần tử của mảng tương ứng của chuỗi hằng sẽ dẫn đến hành vi không xác định (undefined behavior). Thông thường, một chương trình cố gắng sửa đổi mảng tương ứng của chuỗi hằng sẽ gây ra lỗi hoặc không hoạt động đúng.

|  |
| --- |
| char \*foo = "hello";  foo[0] = 'y'; /\* Hành vi không xác định - KHÔNG TỐT! \*/ |

Để tránh việc sửa đổi không an toàn như vậy, khi một con trỏ trỏ tới một chuỗi hằng hoặc có thể trỏ tới nó, chúng ta nên khai báo con trỏ đó là **const** để tránh vô tình gây ra hành vi không xác định.

|  |
| --- |
| const char \*foo = "hello";  /\* TỐT: không thể sửa đổi chuỗi mà foo trỏ tới \*/ |

Tuy nhiên, một con trỏ trỏ vào hoặc trỏ vào mảng gốc của một chuỗi hằng không có tính đặc biệt về bản chất. Chúng ta có thể tự do sửa đổi giá trị của con trỏ để trỏ tới một địa chỉ khác.

|  |
| --- |
| char \*foo = "hello";  foo = "World!"; /\* OK - chúng ta chỉ đổi địa chỉ mà foo trỏ tới \*/ |

Mặt khác, việc khởi tạo một mảng ký tự bằng cách sử dụng cú pháp giống như chuỗi hằng không làm cho mảng đã khởi tạo có các đặc tính của chuỗi hằng. Khởi tạo chỉ đơn giản xác định độ dài và nội dung ban đầu của mảng. Các phần tử trong mảng này vẫn có thể được sửa đổi nếu chúng không được khai báo là hằng số.

|  |
| --- |
| char foo[] = "hello";  foo[0] = 'y'; /\* OK! \*/ |

## PHẦN 6.3 TÍNH ĐỘ DÀI CHUỖI: STRLEN()

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  int main(int *argc*, char \*\**argv*)  {      /\* Exit if no second argument is found. \*/      if (*argc* != 2)      {          puts("Argument missing.");          return EXIT\_FAILURE;      }  *size\_t* len = strlen(*argv*[1]);      printf("The length of the second argument is %zu.\\n", len);      return EXIT\_SUCCESS;  } |

Chương trình trên tính độ dài của **đối số thứ hai** mà nó nhận được và lưu kết quả vào biến **len**. Sau đó, nó in ra độ dài đó lên màn hình. Ví dụ, khi chạy với các tham số program\_name "Hello, world!", chương trình sẽ in ra **The length of the second argument is 13**. bởi vì chuỗi Hello, world! có độ dài là 13 ký tự.

Hàm **strlen** đếm tất cả các byte từ đầu chuỗi cho đến ký tự kết thúc **NULL ('\0')**. Vì vậy, nó chỉ có thể sử dụng khi đảm bảo chuỗi kết thúc bằng NULL.

Tuy nhiên, hãy lưu ý rằng nếu chuỗi chứa các ký tự Unicode, hàm **strlen** sẽ không cho bạn biết có bao nhiêu ký tự trong chuỗi (vì một số ký tự có thể dài hơn một byte). Trong trường hợp này, bạn phải tự đếm các ký tự (đơn vị mã) của chuỗi. Ví dụ dưới đây giải thích điều này:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  int main(void) {      char asciiString[50] = "Hello world!";      char utf8String[50] = "Γειά σου Κόσμε!"; /\* "Hello World!" in Greek \*/      printf("asciiString co %zu byte trong mang\n", sizeof(asciiString));      printf("utf8String co %zu byte trong mang\n", sizeof(utf8String));      printf("\"%s\" co %zu byte\n", asciiString, strlen(asciiString));      printf("\"%s\" co %zu byte\n", utf8String, strlen(utf8String));      return 0;  } |

**Kết quả**

|  |
| --- |
| asciiString co 50 byte trong mang  utf8String co 50 byte trong mang  "Hello world!" co 12 byte  "╬ô╬╡╬╣╬¼ ╧â╬┐╧à ╬Ü╧î╧â╬╝╬╡!" co 27 byte |

## PHẦN 6.4 GIỚI THIỆU CƠ BẢN VỀ CHUỖI

Trong ngôn ngữ lập trình C, một chuỗi (string) là một chuỗi các ký tự kết thúc bằng ký tự null (‘\0’).

Chúng ta có thể tạo các chuỗi sử dụng chuỗi ký tự (**string literals**), là các chuỗi ký tự được bao quanh bởi dấu ngoặc kép; ví dụ, **"hello world".** Chuỗi ký tự được tự động kết thúc bằng ký tự **null.**

Chúng ta có thể tạo các chuỗi bằng nhiều phương pháp khác nhau. Ví dụ, chúng ta có thể khai báo một con trỏ kiểu char và khởi tạo nó để trỏ đến ký tự đầu tiên của một chuỗi:

|  |
| --- |
| char \*string = "hello world"; |

Khi khởi tạo một con trỏ char bằng một chuỗi hằng như trên, chuỗi thực tế thường được cấp phát trong bộ nhớ chỉ đọc (read-only data); string là một con trỏ trỏ đến phần tử đầu tiên của mảng, tức là ký tự '**h**'.

Vì chuỗi hằng này được cấp phát trong bộ nhớ chỉ đọc, nó không thể thay đổi. Bất kỳ thay đổi nào của nó sẽ dẫn đến hành vi không xác định, do đó tốt hơn hãy thêm từ khóa const để có một thông báo lỗi tại thời gian biên dịch như sau:

|  |
| --- |
| char const \*string = "hello world"; |

Nó có tác dụng tương tự như sau:

|  |
| --- |
| char const string\_arr[] = "hello world"; |

Để tạo một chuỗi có thể sửa đổi, bạn có thể khai báo một mảng ký tự và khởi tạo nội dung của nó bằng chuỗi ký tự, như sau:

|  |
| --- |
| char modifiable\_string[] = "hello world"; |

Điều này tương đương với:

|  |
| --- |
| char modifiable\_string[] = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0'}; |

Vì phiên bản thứ hai sử dụng cách khởi tạo bằng dấu ngoặc nhọn, chuỗi không tự động được kết thúc bằng ký tự null trừ khi ký tự **'\0'** được bao gồm rõ ràng trong mảng ký tự, thường là phần tử cuối cùng của mảng.

**Lưu ý:**

* Từ "không thể sửa đổi" (non-modifiable) chỉ ám chỉ rằng các ký tự trong chuỗi hằng không thể thay đổi, nhưng hãy nhớ rằng con trỏ string có thể được sửa đổi (có thể trỏ tới một vị trí khác hoặc tăng hoặc giảm giá trị).
* Cả hai chuỗi đều có hiệu quả tương tự trong việc không cho phép thay đổi các ký tự của chúng. Lưu ý rằng string là một con trỏ kiểu char và nó là một l-value có thể sửa đổi, nó có thể được tăng giá trị hoặc trỏ đến một vị trí khác trong khi mảng string\_arr là một l-value không thể sửa đổi, nó không thể thay đổi.

## PHẦN 6.5 SAO CHÉP CHUỖI

### 6.5.1 Gán con trỏ không sao chép chuỗi

Trong C, việc sử dụng toán tử = (gán) không sao chép chuỗi. Chuỗi trong C được biểu diễn dưới dạng mảng các ký tự kết thúc bằng ký tự null, vì vậy việc sử dụng toán tử = chỉ lưu địa chỉ (con trỏ) của chuỗi đó.

|  |
| --- |
| #include <stdio.h>  int main(void) {      int a = 10, b;      char c[] = "abc", \*d;      b = a; /\* Integer is copied \*/      a = 20; /\* Modifying a leaves b unchanged - b is a 'deep copy' of a \*/      printf("%d %d\n", a, b); /\* "20 10" will be printed \*/      d = c;      /\* Only copies the address of the string -      there is still only one string stored in memory \*/        c[1] = 'x';      /\* Modifies the original string - d[1] = 'x' will do exactly the same thing \*/      printf("%s %s\n", c, d); /\* "axc axc" will be printed \*/      return 0;  } |

Đoạn mã nguồn trên đã biên dịch thành công vì chúng ta sử dụng con trỏ char \*d thay vì mảng char d[3]. Khi sử dụng mảng char d[3], chương trình sẽ gặp lỗi biên dịch vì trong C, bạn không thể gán trực tiếp cho một mảng.

|  |
| --- |
| #include <stdio.h>  int main(void) {      char a[] = "abc";      char b[8];      b = a; /\* compile error \*/      printf("%s\n", b);      return 0;  } |

### 6.5.2 Sao chép chuỗi dùng các hàm tiêu chuẩn

#### 6.5.2.1 strcpy()

Để sao chép chuỗi, ta có thể sử dụng hàm strcpy() trong thư viện string.h. Trước khi sao chép, ta cần phải cấp phát đủ không gian cho chuỗi đích.

|  |
| --- |
| #include <stdio.h>  #include <string.h>  int main(void) {      char a[] = "abc";      char b[8];      strcpy(b, a); /\* think "b special equals a" \*/      printf("%s\n", b); /\* "abc" will be printed \*/      return 0;  } |

#### 6.5.2.2 snprintf()

Để tránh tràn bộ đệm, snprintf() có thể được sử dụng. Đây không phải là giải pháp tốt nhất về mặt hiệu năng vì nó phải phân tích cú pháp chuỗi mẫu, nhưng đây là chức năng an toàn giới hạn bộ đệm duy nhất để sao chép các chuỗi có sẵn trong thư viện chuẩn, có thể được sử dụng mà không cần thực hiện thêm bất kỳ bước nào.

Hàm snprintf() hữu ích khi ta muốn sao chép một chuỗi vào một bộ đệm cố định với kích thước xác định mà không muốn gây ra lỗi tràn bộ đệm (buffer overflow). Thay vì sao chép toàn bộ nội dung của chuỗi như hàm strcpy(), hàm snprintf() cho phép bạn chỉ định kích thước tối đa cho bộ đệm đích.

|  |
| --- |
| #include <stdio.h>  #include <string.h>  int main(void) {      char a[] = "012345678901234567890";      char b[8];  #if 0   strcpy(b, a); /\* causes buffer overrun (undefined behavior), so do not execute this here! \*/  #endif      snprintf(b, sizeof(b), "%s", a); /\* does not cause buffer overrun \*/      printf("%s\n", b); /\* "0123456" will be printed \*/      return 0;  } |

#### 6.5.2.3 strncat()

Một lựa chọn thứ hai, với hiệu suất tốt hơn, là sử dụng hàm strncat() (phiên bản kiểm tra tràn bộ đệm của strcat()) - nó nhận thêm một đối số thứ ba để xác định số byte tối đa cần sao chép.

|  |
| --- |
| char dest[32];  dest[0] = '\0'; // Đảm bảo dest là chuỗi rỗng ban đầu  strncat(*dest*, *source*, sizeof(dest) - 1);  // Sao chép tối đa (sizeof(dest) - 1) phần tử từ source vào dest  // Sau đó, thêm ký tự null '\0' vào cuối chuỗi dest |

Lưu ý rằng trong công thức trên, chúng ta sử dụng sizeof(dest) - 1. Điều này quan trọng vì hàm strncat() luôn thêm một ký tự null ('\0') vào cuối chuỗi, nhưng không tính ký tự này trong kích thước của chuỗi (điều này có thể gây nhầm lẫn và tràn bộ đệm).

Một ví dụ khác là khi ghép chuỗi sau một chuỗi không rỗng:

|  |
| --- |
| char dst[24] = "Clownfish: ";  char src[] = "Marvin and Nemo";  *size\_t* len = strlen(dst);  strncat(*dst*, *src*, sizeof(dst) - len - 1);  printf("%zu: [%s]\n", strlen(*dst*), *dst*); |

Kết quả khi in ra màn hình sẽ là:

|  |
| --- |
| 23: [Clownfish: Marvin and N] |

Lưu ý rằng kích thước được chỉ định là độ dài còn lại của mảng đích (không tính ký tự null cuối cùng). Điều này có thể gây ra vấn đề ghi đè lớn hơn. Để xác định đúng độ dài cần sao chép, bạn nên chỉ định địa chỉ của ký tự null cuối cùng trong nội dung hiện có của chuỗi đích, giúp strncat() không phải quét lại chuỗi đích trước khi bắt đầu sao chép.

|  |
| --- |
| strcpy(*dst*, "Clownfish: ");  assert(len < sizeof(dst) - 1);  strncat(dst + len, src, sizeof(dst) - len - 1);  printf("%zu: [%s]\n", strlen(*dst*), *dst*); |

Điều này tạo ra cùng kết quả như trước, nhưng strncat() không cần quét lại nội dung hiện có của dst trước khi bắt đầu sao chép.

#### 6.5.2.4 strncpy()

Hàm strncpy() là lựa chọn cuối cùng. Mặc dù có thể nghĩ rằng nó nên đứng đầu, nhưng nó lại là một hàm gây nhầm lẫn với hai vấn đề chính:

* Nếu sao chép bằng strncpy() vượt quá giới hạn của bộ đệm, một ký tự null cuối cùng sẽ không được sao chép vào đích.
* strncpy() luôn điền đầy đủ nội dung của đích, bằng các ký tự null nếu cần thiết.

Hiện tại, việc triển khai kỳ quặc này có lịch sử và ban đầu được dùng để xử lý tên tập tin UNIX.)

Cách duy nhất để sử dụng strncpy() đúng đắn là tự đảm bảo kết thúc chuỗi bằng ký tự null:

|  |
| --- |
| strncpy(*b*, *a*, sizeof(b)); /\* tham số thứ ba là kích thước của bộ đệm đích \*/  b[sizeof(b)/sizeof(\*b) - 1] = '\0'; /\* kết thúc chuỗi \*/  printf("%s\n", b); /\* "0123456" sẽ được in ra \*/ |

Tuy nhiên, thậm chí khi bạn có một bộ đệm lớn, việc sử dụng strncpy() trở nên rất không hiệu quả do cần phải điền ký tự null thừa.

## PHẦN 6.6 LẶP QUA CÁC KÍ TỰ TRONG MỘT CHUỖI

Nếu chúng ta biết độ dài của chuỗi, chúng ta có thể sử dụng vòng lặp for để lặp lại các ký tự của chuỗi:

|  |
| --- |
| char \* string = "hello world"; /\* This 11 chars long, excluding the 0-terminator. \*/  *size\_t* i = 0;  for (; i < 11; i++) {      printf("%c\n", string[i]); /\* Print each character of the string. \*/  } |

Ngoài ra, chúng ta có thể sử dụng hàm tiêu chuẩn strlen() để lấy độ dài của một chuỗi nếu chúng ta không biết chuỗi đó là gì:

|  |
| --- |
| size\_t length = strlen(string);  *size\_t* i = 0;  for (; i < length; i++) {      printf("%c\n", string[i]); /\* Print each character of the string. \*/  } |

Cuối cùng, chúng ta có thể tận dụng lợi thế của thực tế là các chuỗi trong C được đảm bảo là kết thúc null (điều mà chúng ta đã làm khi chuyển nó sang strlen() trong ví dụ trước ;-)). Chúng tôi có thể lặp lại mảng bất kể kích thước của nó và dừng lặp lại khi chúng tôi đạt đến ký tự null:

|  |
| --- |
| size\_t i = 0;  while (string[i] != '\0') { /\* Stop looping when we reach the null-character. \*/      printf("%c\n", string[i]); /\* Print each character of the string. \*/      i++;  } |

## PHẦN 6.7 TẠO MẢNG CHUỖI

Một mảng chuỗi trong C có thể có ý nghĩa như sau:

* Một mảng có các phần tử là các con trỏ char \*.
* Một mảng có các phần tử là các mảng ký tự.

Chúng ta có thể tạo một mảng các con trỏ ký tự như sau:

|  |
| --- |
| char \* string\_array[] = {      "foo",      "bar",      "baz"  }; |

Hãy nhớ rằng khi gán các chuỗi chữ cho con trỏ **char \***, các chuỗi đó thực sự được lưu trữ trong bộ nhớ chỉ đọc (**read-only**). Tuy nhiên, mảng string\_array được lưu trữ trong bộ nhớ có thể đọc và viết (**read/write**). Điều này có nghĩa là chúng ta có thể chỉnh sửa các con trỏ trong mảng, nhưng không thể chỉnh sửa các chuỗi mà chúng trỏ tới.

Trong C, tham số argv của hàm main (mảng các tham số dòng lệnh được truyền khi chạy chương trình) là một mảng các con trỏ char \*: char \*argv[].

Chúng ta cũng có thể tạo mảng các mảng ký tự. Vì chuỗi là mảng ký tự, một mảng chuỗi đơn giản là một mảng có các phần tử là các mảng ký tự:

|  |
| --- |
| char modifiable\_string\_array\_literals[][4] = {      "foo",      "bar",      "baz"  }; |

Điều này tương đương với:

|  |
| --- |
| char modifiable\_string\_array[][4] = {      {'f', 'o', 'o', '\0'},      {'b', 'a', 'r', '\0'},      {'b', 'a', 'z', '\0'}  }; |

Lưu ý rằng chúng ta chỉ định 4 là kích thước của chiều thứ hai của mảng; mỗi chuỗi trong mảng của chúng ta thực sự có kích thước 4 byte vì chúng ta phải bao gồm ký tự kết thúc chuỗi **(null-terminating character).**

## PHẦN 6.8 CHUYỂN ĐỔI CHUỖI THÀNH SỐ: ATOI(), ATOF() (CẨN THẬN KHÔNG NÊN SỬ DỤNG)

Cảnh báo: Các hàm atoi, atol, atoll và atof inherently là không an toàn, bởi vì: Nếu giá trị của kết quả không thể biểu diễn, hành vi của chúng không xác định. (7.20.1p1)

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(int *argc*, char\*\* *argv*)  {      int val;      if (*argc* < 2)      {          printf("Usage: %s <integer>\n", *argv*[0]);          return 0;      }      val = atoi(*argv*[1]);      printf("String value = %s, Int value = %d\n", *argv*[1], val);      return 0;  } |

Khi chuỗi cần chuyển đổi là một số nguyên hợp lệ và nằm trong phạm vi của kiểu dữ liệu int, thì hàm atoi() hoạt động đúng:

|  |
| --- |
| $ ./atoi 100  String value = 100, Int value = 100  $ ./atoi 200  String value = 200, Int value = 200 |

Tuy nhiên, nếu chuỗi bắt đầu bằng một số, tiếp theo là ký tự khác, thì chỉ số đầu tiên của chuỗi đó sẽ được chuyển đổi:

|  |
| --- |
| $ ./atoi 0x200  0  $ ./atoi 0123x300  123 |

Nếu chuỗi không phải là một số hợp lệ hoặc không nằm trong phạm vi kiểu dữ liệu int, thì hành vi của hàm atoi() sẽ là không xác định, có thể gây ra lỗi hoặc dẫn đến các hậu quả không mong muốn. Ví dụ:

|  |
| --- |
| $ ./atoi hello  Formatting the hard disk... |

Do sự mơ hồ ở trên và hành vi không xác định này, họ các chức năng atoi không bao giờ nên được sử dụng.

Để chuyển đổi thành long int, hãy sử dụng strtol() thay vì atol().

Để chuyển đổi thành double, hãy sử dụng strtod() thay vì atof().

Để chuyển đổi thành long long int, hãy sử dụng strtoll() thay vì atoll().

## PHẦN 6.9 ĐỌC GHI DỮ LIỆU DẠNG CHUỖI

Ghi dữ liệu đã định dạng vào chuỗi

|  |
| --- |
| int sprintf ( char \* *str*, const char \* *format*, ... ); |

sử dụng hàm **sprintf** để ghi dữ liệu float vào chuỗi.

|  |
| --- |
| #include <stdio.h>  int main ()  {      char buffer [50];      double PI = 3.1415926;      sprintf (buffer, "PI = %.7f", PI);      printf ("%s\n",buffer);      return 0;  } |

Kết quả

|  |
| --- |
| PI = 3.1415926 |

Đọc dữ liệu được định dạng từ chuỗi

|  |
| --- |
| int sscanf ( const char \* *s*, const char \* *format*, ...); |

sử dụng hàm **sscanf** để phân tích dữ liệu được định dạng.

|  |
| --- |
| #include <stdio.h>  int main ()  {      char sentence []="date : 06-06-2012";      char str [50];      int year;      int month;      int day;      sscanf (sentence,"%s : %2d-%2d-%4d", str, &day, &month, &year);      printf ("%s -> %02d-%02d-%4d\n",str, day, month, year);      return 0;  } |

Kết quả:

|  |
| --- |
| date -> 06-06-2012 |

## PHẦN 6.10 TÌM VỊ TRÍ XUẤT HIỆN ĐẦU TIÊN/ CUỐI CÙNG CỦA MỘT KÍ TỰ CỤ THỂ: STRCHR(), STRRCHR()

Các hàm **strchr** và **strrchr** tìm một ký tự trong một chuỗi, ký tự đó nằm trong một mảng ký tự kết thúc bằng NULL. **strchr** trả về một con trỏ tới lần xuất hiện đầu tiên và **strrchr** tới lần xuất hiện cuối cùng.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  int main(int *argc*, char \**argv*[]) // Thêm khai báo biến argc và argv ở đây  {      char toSearchFor = 'A';      /\* Exit if no second argument is found. \*/      if (*argc* != 2)      {          printf("Argument missing.\n");          return EXIT\_FAILURE;      }      {          char \*firstOcc = strchr(*argv*[1], toSearchFor);          if (firstOcc != NULL)          {              printf("First position of %c in %s is %td.\n",                     toSearchFor, *argv*[1], firstOcc - *argv*[1]);              /\* A pointer difference's result is a signed integer and uses the length modifier 't'. \*/          }          else          {              printf("%c is not in %s.\n", toSearchFor, *argv*[1]);          }      }      {          char \*lastOcc = strrchr(*argv*[1], toSearchFor);          if (lastOcc != NULL)          {              printf("Last position of %c in %s is %td.\n",                     toSearchFor, *argv*[1], lastOcc - *argv*[1]);          }      }      return EXIT\_SUCCESS;  } |

Kết quả đầu ra (sau khi đã tạo một pos có tên thực thi):

|  |
| --- |
| $ ./pos AAAAAAA  First position of A in AAAAAAA is 0.  Last position of A in AAAAAAA is 6.  $ ./pos BAbbbbbAccccAAAAzzz  First position of A in BAbbbbbAccccAAAAzzz is 1.  Last position of A in BAbbbbbAccccAAAAzzz is 15.  $ ./pos qwerty  A is not in qwerty. |

Một cách sử dụng phổ biến cho strrchr là trích xuất tên tệp từ một đường dẫn. Ví dụ: để giải nén myfile.txt từ C:\Users\eak\myfile.txt:

|  |
| --- |
| char \*getFileName(const char \**path*)  {      char \*pend;      if ((pend = strrchr(*path*, '\')) != NULL)          return pend + 1;      return NULL;  } |

## PHẦN 6.11 SAO CHÉP VÀ NỐI CHUỖI: STRCPY(), STRCAT()

|  |
| --- |
| #include <stdio.h>  #include <string.h>  int main(void) {      /\* Khai báo một mảng có đủ độ dài để chứa các ký tự và ký tự null kết thúc chuỗi ('\0'). \*/      char mystring[10];      /\* Sao chép chuỗi "foo" vào `mystring` cho đến khi gặp ký tự null. \*/      strcpy(mystring, "foo");      printf("%s\n", mystring);      /\* Tại thời điểm này, chúng ta đã sử dụng 4 ký tự trong `mystring`, bao gồm 3 ký tự của "foo" và ký tự null kết thúc. \*/      /\* Nối chuỗi "bar" vào `mystring`. \*/      strcat(mystring, "bar");      printf("%s\n", mystring);      /\* Bây giờ chúng ta đã sử dụng 7 ký tự trong `mystring`: "foo" dùng 3 ký tự, "bar" dùng 3 ký tự và ký tự null kết thúc. \*/      /\* Sao chép chuỗi "bar" vào `mystring`, ghi đè nội dung cũ. \*/      strcpy(mystring, "bar");      printf("%s\n", mystring);      return 0;  } |

Kết quả:

|  |
| --- |
| foo  foobar  bar |

Khi bạn nối vào, sao chép từ hoặc nối từ một chuỗi hiện có, hãy đảm bảo rằng chuỗi này đã được kết thúc bằng ký tự null trước khi thực hiện thao tác. Trường hợp của bạn ví dụ, chuỗi từ hằng số (string literals) như "foo" đã được biên dịch sẽ luôn được kết thúc bằng ký tự null bởi trình biên dịch, vì vậy bạn có thể yên tâm sử dụng chúng trong các hàm xử lý chuỗi.

## PHẦN 6.12 SO SÁNH: STRCMP(), STRNCMP(), STRCASECMP(), STRNCASECMP()

Các hàm strcase\*-không phải là Tiêu chuẩn C, mà là phần mở rộng POSIX.

strcmp(): Hàm này so sánh hai chuỗi ký tự kết thúc bằng null (null-terminated) theo thứ tự từ điển. Hàm trả về một giá trị số nguyên:

* Nếu chuỗi thứ nhất xuất hiện trước chuỗi thứ hai trong thứ tự từ điển, hàm trả về một giá trị âm.
* Nếu hai chuỗi bằng nhau, hàm trả về giá trị 0.
* Nếu chuỗi thứ nhất xuất hiện sau chuỗi thứ hai trong thứ tự từ điển, hàm trả về một giá trị dương.

|  |
| --- |
| #include <stdio.h>  #include <string.h>  void compare(char const \**lhs*, char const \**rhs*)  {      int result = strcmp(*lhs*, *rhs*); // compute comparison once      if (result < 0) {          printf("%s comes before %s\n", *lhs*, *rhs*);      } else if (result == 0) {          printf("%s equals %s\n", *lhs*, *rhs*);      } else { // last case: result > 0          printf("%s comes after %s\n", *lhs*, *rhs*);      }  }  int main(void)  {      compare("BBB", "BBB");      compare("BBB", "CCCCC");      compare("BBB", "AAAAAA");      return 0;  } |

Kết quả:

|  |
| --- |
| BBB equals BBB  BBB comes before CCCCC  BBB comes after AAAAAA |

Giống như hàm **strcmp**, hàm **strcasecmp** cũng so sánh các đối số của nó theo từ điển sau khi dịch từng ký tự sang chữ thường tương ứng của nó:

|  |
| --- |
| #include <stdio.h>  #include <string.h>  void compare(char const \**lhs*, char const \**rhs*)  {      int result = strcasecmp(*lhs*, *rhs*); // compute case-insensitive comparison once      if (result < 0) {          printf("%s comes before %s\n", *lhs*, *rhs*);      } else if (result == 0) {          printf("%s equals %s\n", *lhs*, *rhs*);      } else { // last case: result > 0          printf("%s comes after %s\n", *lhs*, *rhs*);      }  }  int main(void)  {      compare("BBB", "bBB");      compare("BBB", "ccCCC");      compare("BBB", "aaaaaa");      return 0;  } |

Kết quả:

|  |
| --- |
| BBB equals bBB  BBB comes before ccCCC  BBB comes after aaaaaa |

**strncmp** và **strncasecmp** so sánh tối đa n ký tự. Hàm này nhận thêm một tham số thứ ba là số lượng ký tự để so sánh:

|  |
| --- |
| #include <stdio.h>  #include <string.h>  void compare(char const \**lhs*, char const \**rhs*, int *n*)  {      int result = strncmp(*lhs*, *rhs*, *n*); // compute comparison once      if (result < 0) {          printf("%s comes before %s\n", *lhs*, *rhs*);      } else if (result == 0) {          printf("%s equals %s\n", *lhs*, *rhs*);      } else { // last case: result > 0          printf("%s comes after %s\n", *lhs*, *rhs*);      }  }  int main(void)  {      compare("BBB", "Bb", 1);      compare("BBB", "Bb", 2);      compare("BBB", "Bb", 3);      return 0;  } |

Kết quả:

|  |
| --- |
| BBB equals Bb  BBB comes before Bb  BBB comes before Bb |

## PHẦN 6.13 CHUYỂN ĐỔI CHUỖI THÀNH SỐ MỘT CÁCH AN TOÀN: HÀM STRTOX

Kể từ C99, thư viện C có một tập hợp các hàm chuyển đổi an toàn diễn giải một chuỗi thành một số. Tên của chúng có dạng strtoX, trong đó X là một trong số l, ul, d, v.v. để xác định loại mục tiêu của chuyển đổi.

strtol: Chuyển đổi chuỗi sang số nguyên dạng long (32-bit hoặc 64-bit tùy thuộc vào hệ điều hành và kiến trúc).

strtoul: Chuyển đổi chuỗi sang số nguyên không dấu dạng unsigned long.

strtoll: Chuyển đổi chuỗi sang số nguyên dạng long long (64-bit).

strtoull: Chuyển đổi chuỗi sang số nguyên không dấu dạng unsigned long long.

strtod: Chuyển đổi chuỗi sang số thực dạng double.

|  |
| --- |
| double strtod(char const\* *p*, char\*\* *endptr*);  long double strtold(char const\* *p*, char\*\* *endptr*); |

Chúng cung cấp khả năng kiểm tra xem chuyển đổi có bị tràn hay thiếu không:

|  |
| --- |
| double ret = strtod(argv[1], 0); /\* attempt conversion \*/  /\* check the conversion result. \*/  if ((ret == HUGE\_VAL || ret == -HUGE\_VAL) && errno == ERANGE)      return; /\* numeric overflow in in string \*/  else if (ret == HUGE\_VAL && errno == ERANGE)      return; /\* numeric underflow in in string \*/  /\* At this point we know that everything went fine so ret may be used \*/ |

Nếu chuỗi không chứa số nào, hàm **strtod** trên sẽ trả về giá trị 0.0.

Tuy nhiên, nếu không muốn nhận giá trị mặc định này, ta có thể sử dụng tham số bổ sung **endptr**. Tham số này là một con trỏ tới con trỏ, và nó sẽ được trỏ tới cuối số đã được phát hiện trong chuỗi. Nếu tham số này được đặt thành 0 hoặc NULL, nó sẽ được bỏ qua.

Sử dụng tham số **endptr**, ta có thể kiểm tra xem quá trình chuyển đổi có thành công hay không và xác định vị trí kết thúc của số trong chuỗi:

|  |
| --- |
| char \*check = 0;  double ret = strtod(argv[1], &*check*); /\* thử chuyển đổi \*/  if (argv[1] == check)      return; /\* Không phát hiện số nào trong chuỗi \*/  else if ((ret == HUGE\_VAL || ret == -HUGE\_VAL) && errno == ERANGE)      return; /\* Tràn số khi chuyển đổi \*/  else if (ret == HUGE\_VAL && errno == ERANGE)      return; /\* Số quá nhỏ khi chuyển đổi \*/  /\* Ở đây, ta biết rằng mọi thứ đã diễn ra suôn sẻ, giá trị ret có thể sử dụng \*/ |

Có các hàm tương tự để chuyển đổi sang các loại số nguyên rộng hơn:

|  |
| --- |
| long strtol(char const\* *p*, char\*\* *endptr*, int *nbase*);  long long strtoll(char const\* *p*, char\*\* *endptr*, int *nbase*);  unsigned long strtoul(char const\* *p*, char\*\* *endptr*, int *nbase*);  unsigned long long strtoull(char const\* *p*, char\*\* *endptr*, int *nbase*); |

Các hàm này có tham số thứ ba nbase giữ cơ sở số trong đó số được viết.

|  |
| --- |
| long a = strtol("101", 0, 2 ); /\* a = 5L \*/  long b = strtol("101", 0, 8 ); /\* b = 65L \*/  long c = strtol("101", 0, 10); /\* c = 101L \*/  long d = strtol("101", 0, 16); /\* d = 257L \*/  long e = strtol("101", 0, 0 ); /\* e = 101L \*/  long f = strtol("0101", 0, 0 ); /\* f = 65L \*/  long g = strtol("0x101", 0, 0 ); /\* g = 257L \*/ |

Giá trị 0 cho nbase có nghĩa là chuỗi sẽ được hiểu như các số trong chương trình C: tiền tố 0x tương ứng với số thập lục phân, tiền tố 0 tương ứng với số bát phân, và tất cả các số khác được hiểu là số thập phân.

Do đó, cách thiết thực nhất để diễn giải một đối số dòng lệnh dưới dạng số sẽ là

|  |
| --- |
| int main(int *argc*, char\* *argv*[] {      if (*argc* < 1)          return EXIT\_FAILURE; /\* No number given. \*/          /\* use strtoull because size\_t may be wide \*/  *size\_t* mySize = strtoull(*argv*[1], 0, 0);          /\* then check conversion results. \*/          ...          return EXIT\_SUCCESS;  } |

Điều này cho phép chương trình được gọi với một tham số dòng lệnh ở hệ thống bát phân, thập phân hoặc thập lục phân.

## PHẦN 6.14 STRSPN VÀ STRCSPN

Cho trước một chuỗi, strspn tính toán độ dài của chuỗi con ban đầu (span) chỉ bao gồm một danh sách ký tự cụ thể. strcspn cũng tương tự, ngoại trừ nó tính toán độ dài của chuỗi con ban đầu bao gồm bất kỳ ký tự nào ngoại trừ những ký tự được liệt kê:

|  |
| --- |
| /\*  Provided a string of "tokens" delimited by "separators", print the tokens along  with the token separators that get skipped.  \*/  #include <stdio.h>  #include <string.h>  int main(void)  {      const char sepchars[] = ",.;!?";      char foo[] = ";ball call,.fall gall hall!?.,";      char \*s;      int n;      for (s = foo; \*s != 0; /\*empty\*/) {          /\* Get the number of token separator characters. \*/          n = (int)strspn(s, sepchars);          if (n > 0)              printf("skipping separators: << %.\*s >> (length=%d)\n", n, s, n);          /\* Actually skip the separators now. \*/          s += n;          /\* Get the number of token (non-separator) characters. \*/          n = (int)strcspn(s, sepchars);          if (n > 0)              printf("token found: << %.\*s >> (length=%d)\n", n, s, n);          /\* Skip the token now. \*/          s += n;      }      printf("== token list exhausted ==\n");      return 0;  } |

Các hàm tương tự sử dụng chuỗi ký tự rộng là wcsspn và wcscspn; chúng được sử dụng theo cùng một cách.

# CHỮ SỐ, KÝ TỰ VÀ CHUỖI

## PHẦN 7.1 FLOATING POINT LITERALS( CHỮ SỐ DẤU CHẤM ĐỘNG)

Chữ dấu chấm động được sử dụng để biểu diễn các số thực có dấu. Các hậu tố sau đây có thể được sử dụng để chỉ định loại chữ:

|  |  |  |
| --- | --- | --- |
| Hậu tố | Kiểu dữ liệu | Ví dụ |
| Không có | double | 3.1415926 -3E6 |
| f, F | float | 3.1415926f 2.1E-6F |
| l, L | long double | 3.1415926L 1E126L |

Để sử dụng các hậu tố này, hằng số phải là một hằng số số thực dấu chấm động. Ví dụ, 3f là không hợp lệ, vì 3 là một hằng số số nguyên, trong khi 3.f hoặc 3.0f là đúng. Đối với kiểu long double, khuyến nghị luôn sử dụng chữ hoa L để cải thiện tính đọc hiểu.

## PHẦN 7.2 STRING LITERALS( CHUỖI KÍ TỰ CHỮ)

Chuỗi ký tự chữ được sử dụng để xác định các mảng ký tự. Chúng là các chuỗi ký tự được bao quanh bởi dấu ngoặc kép (ví dụ: "abcd") và có kiểu dữ liệu là char\*.

Tiền tố L làm cho chữ trở thành một mảng ký tự rộng, thuộc loại wchar\_t\*. Ví dụ: L"abcd".

Kể từ phiên bản C11, có các tiền tố mã hóa khác, tương tự như tiền tố L:

|  |  |  |
| --- | --- | --- |
| Tiền tố | Loại cơ sở | Mã hóa |
| Không có | char | tùy thuộc vào nền tảng |
| L | wchar\_t | tùy thuộc vào nền tảng |
| u8 | char | UTF-8 |
| u | char16\_t | Thường là UTF-16 |
| U | char32\_t | Thường là UTF-32 |

Đối với hai tiền tố cuối cùng, ta có thể kiểm tra với các macro kiểm tra tính năng xem liệu mã hóa có phải là mã hóa UTF tương ứng hay không.

## PHẦN 7.3 CHARACTER LITERALS(KÝ TỰ)

Hằng ký tự là một loại đặc biệt của hằng số số nguyên được sử dụng để biểu diễn một ký tự duy nhất. Chúng được bao quanh bởi dấu ngoặc đơn, ví dụ: 'a' và có kiểu dữ liệu là int. Giá trị của hằng số là một giá trị số nguyên tương ứng với bộ ký tự của máy tính. Chúng không cho phép tiền tố.

Tiền tố L trước một hằng ký tự biến nó thành một ký tự rộng của kiểu wchar\_t. Tương tự, từ phiên bản C11, các tiền tố u và U biến chúng thành ký tự rộng của kiểu char16\_t và char32\_t.

Khi muốn biểu diễn một số ký tự đặc biệt như ký tự không in, ta sử dụng các chuỗi escape. Các chuỗi escape sử dụng một chuỗi ký tự được dịch thành một ký tự khác. Tất cả các chuỗi escape đều bao gồm hai hoặc nhiều ký tự, ký tự đầu tiên trong chuỗi là dấu gạch chéo . Các ký tự ngay sau dấu gạch chéo xác định ký tự mà chuỗi escape được hiểu là gì.

Các chuỗi escape được sử dụng để biểu diễn các ký tự như sau:

|  |  |
| --- | --- |
| \b | Dấu phẩy ngược |
| \f | Lùi trang |
| \n | Xuống dòng |
| \r | Cú pháp |
| \t | Tab ngang |
| \v | Tab dọc |
| \\ | Dấu gạch chéo ngược |
| \’ | Dấu nháy đơn |
| \” | Dấu nháy kép |
| \? | Dấu chấm hỏi |
| \nnn | Giá trị bát phân |
| \xnn | Giá trị thập lục phân |
| \a | Ký tự thông báo (còi) |
| \unnnn | Tên ký tự toàn cầu (universal character name) có 4 chữ số thập lục phân |
| \Unnnnnnnn | Tên ký tự toàn cầu có 8 chữ số thập lục phân |

Một tên ký tự toàn cầu là một mã Unicode. Một tên ký tự toàn cầu có thể ánh xạ thành nhiều ký tự. Các chữ số n được hiểu là các chữ số thập lục phân. Tùy thuộc vào mã hóa UTF đang sử dụng, một chuỗi tên ký tự toàn cầu có thể cho ra một mã Unicode bao gồm nhiều ký tự, thay vì chỉ một ký tự duy nhất.

Khi sử dụng chuỗi escape xuống dòng trong I/O chế độ văn bản, nó sẽ được chuyển đổi thành byte xuống dòng cụ thể của hệ điều hành.

Chuỗi escape dấu chấm hỏi được sử dụng để tránh trigraphs. Ví dụ, ??/ sẽ được biên dịch thành trigraph đại diện cho ký tự gạch chéo '', nhưng sử dụng ??/ sẽ cho ra chuỗi "??/". Có thể có một, hai hoặc ba số octal n trong chuỗi escape giá trị bát phân.

## PHẦN 7.4 INTEGER LITERALS(SỐ NGUYÊN)

Hằng số nguyên được sử dụng để cung cấp các giá trị nguyên. Có ba cơ số số học được hỗ trợ, được chỉ định bằng tiền tố:

|  |  |  |
| --- | --- | --- |
| Cơ số | Tiền tố | Ví dụ |
| Thập phân | Không có | 5 |
| Bát phân | 0 | 0345 |
| Thập lục phân | 0x hoặc 0X | 0x12AB, 0X12AB, 0x12ab, 0x12Ab |

Lưu ý rằng viết không bao gồm bất kỳ dấu hiệu nào, do đó hằng số nguyên luôn luôn dương. Một số như -1 được xem như là biểu thức có một hằng số nguyên (1) được đảo dấu bằng dấu -.

Kiểu dữ liệu của hằng số nguyên thập phân là kiểu dữ liệu đầu tiên có thể chứa giá trị từ int và long. Kể từ C99, kiểu long long cũng được hỗ trợ cho các hằng số rất lớn.

Kiểu dữ liệu của hằng số nguyên bát phân hoặc thập lục phân là kiểu dữ liệu đầu tiên có thể chứa giá trị từ int, unsigned, long và unsigned long. Kể từ C99, kiểu long long và unsigned long long cũng được hỗ trợ cho các hằng số rất lớn.

Sử dụng các hậu tố khác nhau, kiểu mặc định của hằng số có thể được thay đổi.

|  |  |
| --- | --- |
| Hậu tố | Giải thích |
| L, l | long int |
| LL, ll( kể tử C99) | long long int |
| U,u | unsigned |

Các hậu tố U và L/LL có thể được kết hợp trong bất kỳ thứ tự và chữ hoa nào. Lỗi nếu có hậu tố bị trùng lặp (ví dụ cung cấp hai hậu tố U) ngay cả khi chúng có chữ hoa khác nhau.

# COMPOUND LITERALS

## PHẦN 8.1 ĐỊNH NGHĨA/ KHỞI TẠO COMPOUND LITERALS

A compound literal là một đối tượng vô danh được tạo ra trong phạm vi nơi nó được định nghĩa. Khái niệm này được giới thiệu lần đầu trong tiêu chuẩn C99. Dưới đây là một ví dụ về A compound literal:

**Ví dụ từ tiêu chuẩn C, C11-§6.5.2.5/9:**

|  |
| --- |
| int \*p = (int [2]){ 2, 4 }; |

Biến p được khởi tạo với địa chỉ của phần tử đầu tiên của một mảng vô danh gồm hai số nguyên.

Hằng số hợp thành là một lvalue. Tuổi thọ của đối tượng vô danh được xác định bởi thời gian sống của nó, có thể là tĩnh (nếu hằng số hợp thành xuất hiện trong phạm vi tệp tin) hoặc tự động (nếu hằng số hợp thành xuất hiện trong phạm vi khối), và trong trường hợp sau, đối tượng sẽ kết thúc khi điều khiển thoát khỏi khối bao quanh nó.

|  |
| --- |
| void f(void)  {      int \*p;      /\*...\*/      p = (int [2]){ \*p };      /\*...\*/  } |

Biến p được gán địa chỉ của phần tử đầu tiên của một mảng gồm hai số nguyên, phần tử đầu tiên có giá trị trước đó được trỏ bởi p và phần tử thứ hai có giá trị là 0. Trong trường hợp này, biến p vẫn còn hiệu lực cho đến khi kết thúc khối.

**Compound literal với bộ chỉ mục**

|  |
| --- |
| struct *point* {      unsigned x;      unsigned y;  };  extern void drawline(struct *point*, struct *point*);  // Sử dụng nơi đó  drawline((struct point){.x=1, .y=1}, (struct point){.x=3, .y=4}); |

Một hàm giả định drawline nhận hai đối số có kiểu struct point. Đối số đầu tiên có giá trị tọa độ x == 1 và y == 1, trong khi đối số thứ hai có x == 3 và y == 4.

**Compound literal không xác định chiều dài của mảng**

|  |
| --- |
| int \*p = (int []){ 1, 2, 3}; |

Trong trường hợp này, kích thước của mảng không được xác định trước và sẽ được xác định bởi độ dài của danh sách khởi tạo.

**Compound literal có độ dài của danh sách khởi tạo nhỏ hơn kích thước của mảng đã chỉ định**

|  |
| --- |
| int \*p = (int [10]){1, 2, 3}; |

Các phần tử còn lại của Compound literal sẽ được khởi tạo mặc định là 0.

**Compound literal chỉ đọc**

Lưu ý rằng Compound literal là một lvalue và do đó các phần tử của nó có thể thay đổi được. Compound literal chỉ đọc có thể được chỉ định bằng cách sử dụng từ khóa const như sau:

|  |
| --- |
| (const int[]){1, 2} |

Compound literal chứa các biểu thức tùy ý

Trong một hàm, Compound literal, giống như bất kỳ khởi tạo nào từ tiêu chuẩn C99 trở đi, có thể chứa các biểu thức tùy ý.

|  |
| --- |
| void foo()  {      int \*p;      int i = 2; j = 5;      /\*...\*/      p = (int [2]){ i+j, i\*j };      /\*...\*/  } |

Trong trường hợp này, biến p được gán địa chỉ của phần tử đầu tiên của một mảng gồm hai số nguyên, với phần tử đầu tiên có giá trị là i+j và phần tử thứ hai có giá trị là i\*j.

# BIT- FIENDS

# MẢNG

Mảng là các loại dữ liệu kiểu dữ liệu dẫn xuất (derived data type) là kiểu dữ liệu (data type) có nguồn gốc từ các kiểu dữ liệu nguyên thủy (primitive data type) hoặc kiểu dữ liệu xây dựng sẵn (built-in data type). Trong C có bốn loại, cụ thể là: Hàm (function), mảng (array), con trỏ (pointer), tham chiếu (reference), đại diện cho một tập hợp có thứ tự các giá trị ("phần tử") của một loại dữ liệu khác. Hầu hết các mảng trong C có một số lượng cố định các phần tử của bất kỳ loại nào và biểu diễn của chúng lưu trữ các phần tử liền kề trong bộ nhớ mà không có khoảng trống hoặc ghi chú. C cho phép mảng đa chiều, trong đó các phần tử là các mảng khác, và cũng cho phép mảng của con trỏ.

C hỗ trợ các mảng được cấp phát động có kích thước được xác định tại thời điểm chạy. C99 và các phiên bản sau đó hỗ trợ các mảng có độ dài biến đổi hay còn gọi là VLA (Variable Length Arrays).

## PHẦN 10.1 KHAI BÁO VÀ KHỞI TẠO MẢNG

Cú pháp chung để khai báo mảng một chiều là

|  |
| --- |
| type arrName[size]; |

Ở đây, **type** có thể là bất kỳ kiểu dữ liệu cơ bản nào hoặc kiểu dữ liệu do người dùng định nghĩa, ví dụ như các structures, **arrName** là một định danh do người dùng định nghĩa và **size** là một hằng số nguyên.

Khai báo một mảng (một mảng gồm 10 biến kiểu int trong trường hợp này) được thực hiện như sau:

|  |
| --- |
| int array[10]; |

Hiện tại nó chứa các giá trị không xác định. Để đảm bảo nó chứa các giá trị 0 khi khai báo, bạn có thể làm như sau:

|  |
| --- |
| int array[10] = {0}; |

Mảng cũng có thể có bộ khởi tạo, ví dụ này khai báo một mảng gồm 10 số nguyên, trong đó 3 số nguyên đầu tiên sẽ chứa các giá trị 1, 2, 3, các giá trị còn lại sẽ là 0:

|  |
| --- |
| int array[10] = {1, 2, 3}; |

Trong phương pháp khởi tạo trên, giá trị đầu tiên trong danh sách sẽ được gán cho thành viên đầu tiên của mảng, giá trị thứ hai sẽ được gán cho thành viên thứ hai của mảng và cứ tiếp tục như vậy. Nếu kích thước của danh sách nhỏ hơn kích thước của mảng, thì như trong ví dụ trên, các thành viên còn lại của mảng sẽ được khởi tạo thành giá trị 0. Với cách khởi tạo danh sách có chỉ định (ISO C99), khởi tạo rõ ràng cho các thành viên của mảng là có thể. Ví dụ:

|  |
| --- |
| int array[5] = {[2] = 5, [1] = 2, [4] = 9}; /\* mảng là {0, 2, 5, 0, 9} \*/ |

Trong hầu hết các trường hợp, trình biên dịch có thể suy ra độ dài của mảng cho bạn, điều này có thể được đạt bằng cách để trống các dấu ngoặc vuông:

|  |
| --- |
| int array[] = {1, 2, 3}; /\* một mảng gồm 3 số nguyên \*/  int array[] = {[3] = 8, [0] = 9}; /\* kích thước là 4 \*/ |

Khai báo một mảng có độ dài bằng không không được phép.

Phiên bản C99 trở lên hỗ trợ Mảng Động (Variable Length Arrays - VLA). Tuy nhiên, từ phiên bản C11 trở đi, VLA đã trở thành tùy chọn. VLA tương đương với mảng thông thường, nhưng có một điểm khác biệt quan trọng: Độ dài không cần phải biết tại thời điểm biên dịch. VLA có thời gian lưu trữ tự động. Chỉ có con trỏ tới VLA có thể có thời gian lưu trữ tĩnh.

|  |
| --- |
| *size\_t* m = tinh\_do\_dai(); /\* tính độ dài mảng tại thời điểm chạy \*/  int vla[m]; /\* tạo mảng với độ dài tính toán được \*/ |

**Quan trọng:**

VLA có thể nguy hiểm. Nếu mảng vla trong ví dụ trên yêu cầu nhiều không gian trên ngăn xếp hơn khả dụng, thì ngăn xếp sẽ tràn. Việc sử dụng VLA thường không được khuyến khích trong các hướng dẫn về phong cách và sách giáo trình và bài tập.

## PHẦN 10.2 LẶP QUA MẢNG HIỆU QUẢ VÀ THỨ TỰ ƯU TIÊN HÀNG( ROW MAJOR ORDER)

Mảng trong ngôn ngữ lập trình C có thể được xem như một vùng nhớ liên tục. Cụ thể hơn, chiều cuối cùng của mảng là phần liên tục trong bộ nhớ. Chúng ta gọi điều này là "thứ tự ưu tiên hàng" (row-major order). Điều quan trọng là hiểu được và thực tế rằng khi xảy ra "cache fault" (lỗi bộ nhớ đệm), bộ nhớ đệm sẽ tải một "dòng bộ nhớ đệm" hoàn chỉnh vào bộ nhớ đệm khi truy cập dữ liệu chưa được lưu vào bộ nhớ đệm, nhằm ngăn lỗi bộ nhớ đệm sau này. Vì vậy, khi truy cập vào một mảng có kích thước 10000x10000 và bắt đầu từ **array[0][0],** có thể sẽ tải **array[0][1]** vào bộ nhớ đệm, để tránh lỗi bộ nhớ đệm trong trường hợp truy cập tiếp theo. Tuy nhiên, nếu truy cập ngay sau đó là **array[1][0]**, thì điều này sẽ tạo ra một lỗi bộ nhớ đệm thứ hai. Điều này xảy ra vì **array[1][0]** cách **array[0][0]** một khoảng **sizeof(type)\*10000 byte**, và do đó, chắc chắn không nằm trên cùng một dòng bộ nhớ đệm. Vì vậy, khi lặp qua mảng một cách không hiệu quả như sau:

|  |
| --- |
| #define ARRLEN 10000  int array[ARRLEN][ARRLEN];  *size\_t* i, j;  for (i = 0; i < ARRLEN; ++i)  {      for(j = 0; j < ARRLEN; ++j)      {          array[j][i] = 0;      }  } |

Cách hiệu quả hơn là:

|  |
| --- |
| #define ARRLEN 10000  int array[ARRLEN][ARRLEN];  *size\_t* i, j;  for (i = 0; i < ARRLEN; ++i)  {      for(j = 0; j < ARRLEN; ++j)      {          array[i][j] = 0;      }  } |

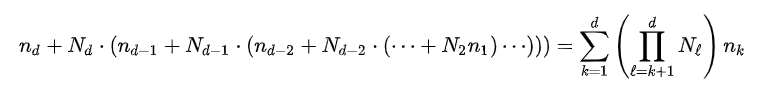
Cũng giống như trên, khi làm việc với mảng một chiều có nhiều chỉ số (ví dụ: 2 chiều với chỉ số i và j), bạn nên lặp qua mảng như sau:

|  |
| --- |
| #define DIM\_X 10  #define DIM\_Y 20  int array[DIM\_X\*DIM\_Y];  *size\_t* i, j;  for (i = 0; i < DIM\_X; ++i)  {      for(j = 0; j < DIM\_Y; ++j)      {          array[i\*DIM\_Y+j] = 0;      }  } |

Và với 3 chiều và chỉ số i, j, k:

|  |
| --- |
| #define DIM\_X 10  #define DIM\_Y 20  #define DIM\_Z 30  int array[DIM\_X\*DIM\_Y\*DIM\_Z];  *size\_t* i, j, k;  for (i = 0; i < DIM\_X; ++i)  {      for(j = 0; j < DIM\_Y; ++j)      {          for (k = 0; k < DIM\_Z; ++k)          {              array[i\*DIM\_Y\*DIM\_Z+j\*DIM\_Z+k] = 0;          }      }  } |

Hoặc một cách tổng quát hơn, khi chúng ta có một mảng với N1 x N2 x ... x Nd phần tử, d chiều và chỉ số được ký hiệu là n1, n2,..., nd, sự dịch chuyển được tính như sau:



Hình ảnh / công thức lấy từ: https://en.wikipedia.org/wiki/Row-major\_order

## PHẦN 10.3 ĐỘ DÀI MẢNG

Độ dài của mảng có giá trị cố định và được xác định khi khai báo mảng. Tuy nhiên, ta có thể tính độ dài của mảng trong một số trường hợp để làm mã nguồn linh hoạt hơn, đặc biệt khi độ dài được xác định tự động từ bộ khởi tạo:

|  |
| --- |
| int array[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  /\* Kích thước của `array` tính bằng byte \*/  *size\_t* size = sizeof(array);  /\* Số phần tử trong `array` \*/  *size\_t* length = sizeof(array) / sizeof(array[0]); |

Tuy nhiên trong hầu hết các ngữ cảnh mà mảng xuất hiện trong biểu thức, nó sẽ tự động chuyển đổi thành một con trỏ trỏ tới phần tử đầu tiên của mảng (gọi là "decay to pointer"). Trường hợp mảng là toán hạng của toán tử sizeof là một trong số ít các ngoại lệ. Con trỏ thu được **không phải là một mảng** và không mang bất kỳ thông tin về độ dài của mảng gốc. Vì vậy, nếu độ dài này cần thiết khi sử dụng con trỏ, ví dụ như khi con trỏ được truyền vào một hàm, thì nó phải được truyền riêng biệt.

Ví dụ, giả sử chúng ta muốn viết một hàm để trả về phần tử cuối cùng của một mảng int. Tiếp tục từ trên, chúng ta có thể gọi nó như vậy:

|  |
| --- |
| /\* Mảng 'array' sẽ tự động chuyển đổi thành một con trỏ,   \*vì vậy kích thước cần được truyền riêng biệt \*/  int last = get\_last(array, length); |

Và đây là cách triển khai hàm **get\_last**:

|  |
| --- |
| int get\_last(int *input*[], *size\_t* *length*) {      return input[length - 1];  } |

Lưu ý đặc biệt là khai báo của tham số **input** giống như khai báo một mảng, nhưng thực tế thì nó được xem như là một **con trỏ** tới kiểu **int**. Điều này tương đương với việc khai báo **int \*input**. Thậm chí nếu kích thước của mảng được chỉ định, thì điều này không ảnh hưởng đến kiểu của input. Điều này có thể được xem như một kỹ thuật nhớ dễ dàng.

Tuy nhiên, không nên cố gắng xác định kích thước của mảng từ con trỏ, như trong hàm **BAD\_get\_last**:

|  |
| --- |
| int BAD\_get\_last(int input[]) {      /\* SAI VÌ KHÔNG THỂ TÍNH KÍCH THƯỚC CỦA MẢNG MÀ input TRỎ TỚI: \*/      size\_t length = sizeof(input) / sizeof(input[0]);      return input[length - 1]; /\* Lỗi - không phải kết quả mong muốn \*/  } |

Trên thực tế, lỗi cụ thể đó phổ biến đến mức một số trình biên dịch nhận ra và cảnh báo về nó. clang, chẳng hạn, sẽ phát ra cảnh báo sau:

Sử dụng toán tử sizeof trên con trỏ không trả về kích thước của mảng mà nó trỏ tới, mà trả về kích thước của con trỏ. Điều này dẫn đến kết quả sai khi cố gắng truy cập vào phần tử cuối cùng của mảng. Do đó, để sử dụng đúng kích thước của mảng trong hàm, bạn phải truyền kích thước riêng biệt vào hàm như trong hàm get\_last.

|  |
| --- |
| warning: sizeof on array function parameter will return size of 'int \*' instead of 'int []' [-  Wsizeof-array-argument]          int length = sizeof(input) / sizeof(input[0]);                              ^  note: declared here  int BAD\_get\_last(int input[])                       ^ |

## PHẦN 10.4 TRUYỀN MẢNG ĐA CHIỀU VÀO HÀM

Mảng đa chiều tuân theo cùng các quy tắc như mảng một chiều khi truyền chúng vào hàm. Tuy nhiên, sự kết hợp giữa decay-to-pointer, ưu tiên toán tử và hai cách khai báo mảng đa chiều (mảng của mảng và mảng của con trỏ) có thể khiến khai báo của các hàm như vậy trở nên không rõ ràng. Ví dụ sau đây chỉ ra các cách đúng để truyền mảng đa chiều vào hàm:

|  |
| --- |
| #include <assert.h>  #include <stdlib.h>  /\* Khi truyền một mảng đa chiều (tức là một mảng của các mảng) vào một hàm,  nó sẽ bị chuyển đổi thành con trỏ đến phần tử đầu tiên như thường lệ.  Nhưng chỉ có cấp cao nhất sẽ bị chuyển đổi,  do đó cái được truyền vào là con trỏ đến một mảng với một kích thước cố định  (trong trường hợp này là 4). \*/  void f(int *x*[][4]) {      assert(sizeof(\**x*) == sizeof(int) \* 4);  }  /\* Nguyên mẫu này tương đương với f(int x[][4]).  Dấu ngoặc đơn xung quanh \*x là bắt buộc vì [index]  có mức độ ưu tiên cao hơn so với \*expr, do đó int \*x[4] thông thường sẽ tương đương với int  \*(x[4]), tức là một mảng gồm 4 con trỏ tới int.  Nhưng nếu nó được khai báo là một tham số của hàm,  nó sẽ bị chuyển đổi thành con trỏ và trở thành int \*\*x,  điều này không tương thích với x[2][4]. \*/  void g(int (\**x*)[4]) {      assert(sizeof(\**x*) == sizeof(int) \* 4);  }  /\* Một mảng của con trỏ có thể được truyền vào hàm này,  vì nó sẽ bị chuyển đổi thành con trỏ đến con trỏ (int \*\*),  nhưng một mảng của các mảng không thể. \*/  void h(int \*\**x*) {      assert(sizeof(\**x*) == sizeof(int\*));  }  int main(void) {      int foo[2][4];      f(foo);      g(foo);      /\* Ở đây, chúng ta đang tạo động một mảng của các con trỏ.      Lưu ý rằng kích thước của mỗi chiều không phải là một phần của kiểu dữ liệu,      vì vậy hệ thống kiểu chỉ coi nó như một con trỏ đến con trỏ (int \*\*),      không phải con trỏ đến mảng hay mảng của các mảng. \*/      int \*\*bar = malloc(sizeof(\*bar) \* 2);      assert(bar);      for (*size\_t* i = 0; i < 2; i++) {          bar[i] = malloc(sizeof(\*bar[i]) \* 4);          assert(bar[i]);      }      h(bar);      for (*size\_t* i = 0; i < 2; i++) {          free(bar[i]);      }      free(bar);  } |

**Xem thêm**

Truyền mảng vào hàm

## PHẦN 10.5 MẢNG ĐA CHIỀU

Ngôn ngữ lập trình C cho phép sử dụng mảng đa chiều. Dưới đây là cú pháp chung của khai báo mảng đa chiều:

|  |
| --- |
| type name[size1][size2]...[sizeN]; |

Ví dụ, khai báo sau tạo ra một mảng số nguyên ba chiều (5 x 10 x 4):

|  |
| --- |
| int arr[5][10][4]; |

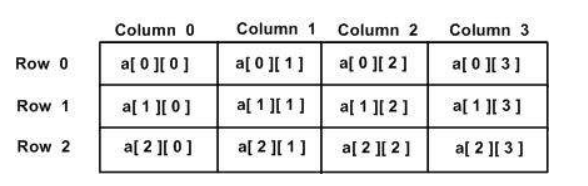
### 10.5.1 Mảng hai chiều

Dạng đơn giản nhất của mảng đa chiều là mảng hai chiều. Mảng hai chiều thực chất là một danh sách của các mảng một chiều. Để khai báo một mảng hai chiều số nguyên với kích thước m x n, chúng ta có thể viết như sau:

|  |
| --- |
| type arrayName[m][n]; |

Trong đó, type có thể là bất kỳ kiểu dữ liệu hợp lệ trong C (int, float, v.v...) và arrayName có thể là bất kỳ định danh hợp lệ trong C. Một mảng hai chiều có thể được hình dung như một bảng với m hàng và n cột. Lưu ý rằng thứ tự cần phải đúng trong C. Mảng int a[4][3] không giống với mảng int a[3][4]. Số hàng được đặt trước vì C là một ngôn ngữ ưu tiên hàng đầu.

Một mảng hai chiều a, chứa ba hàng và bốn cột có thể được biểu diễn như sau:



Do đó, mỗi phần tử trong mảng a được xác định bởi tên phần tử có dạng a[i][j], trong đó a là tên của mảng, i là chỉ số hàng và j là chỉ số cột. Nhớ rằng các hàng và cột được đánh số từ 0. Điều này rất giống với cách đánh chỉ số trong toán học cho ma trận 2D.

### 10.5.2 Khởi tạo mảng 2 chiều

Mảng đa chiều có thể được khởi tạo bằng cách chỉ định các giá trị trong ngoặc đơn cho mỗi hàng. Dưới đây là ví dụ về cách định nghĩa một mảng với 3 hàng, mỗi hàng có 4 cột:

|  |
| --- |
| int a[3][4] = {      {0, 1, 2, 3}, /\* giá trị khởi tạo cho hàng có chỉ số 0 \*/      {4, 5, 6, 7}, /\* giá trị khởi tạo cho hàng có chỉ số 1 \*/      {8, 9, 10, 11} /\* giá trị khởi tạo cho hàng có chỉ số 2 \*/  }; |

Dấu ngoặc nhọn lồng nhau, biểu thị hàng được chọn, là tùy chọn. Khởi tạo sau đây tương đương với ví dụ trước:

|  |
| --- |
| int a[3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}; |

Trong khi việc tạo mảng với ngoặc nhọn lồng nhau là tùy chọn, nó được khuyến khích mạnh mẽ vì nó dễ đọc và rõ ràng hơn.

### 10.5.3 Truy cập phần tử trong mảng hai chiều

Một phần tử trong mảng hai chiều được truy cập bằng cách sử dụng các chỉ mục con, tức là chỉ số hàng và chỉ số cột của mảng. Ví dụ:

|  |
| --- |
| int val = a[2][3]; |

Câu lệnh trên sẽ lấy phần tử thứ 4 từ hàng thứ 3 của mảng. Hãy xem ví dụ sau đây khi chúng ta sử dụng một vòng lặp lồng nhau để xử lý một mảng hai chiều:

|  |
| --- |
| #include <stdio.h>  int main () {        /\* một mảng với 5 hàng và 2 cột \*/      int a[5][2] = { {0, 0}, {1, 2}, {2, 4}, {3, 6}, {4, 8} };      int i, j;      /\* in giá trị của từng phần tử trong mảng \*/      for (i = 0; i < 5; i++) {          for (j = 0; j < 2; j++) {              printf("a[%d][%d] = %d\n", i, j, a[i][j]);          }      }  return 0;  } |

Khi đoạn mã trên được biên dịch và thực thi, nó sẽ tạo ra kết quả như sau:

|  |
| --- |
| a[0][0] = 0  a[0][1] = 0  a[1][0] = 1  a[1][1] = 2  a[2][0] = 2  a[2][1] = 4  a[3][0] = 3  a[3][1] = 6  a[4][0] = 4  a[4][1] = 8 |

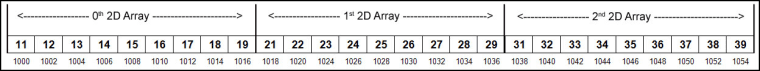
### 10.5.4 Mảng ba chiều

Mảng 3D về cơ bản là một mảng của các mảng của các mảng: đó là một mảng hoặc tập hợp các mảng 2D và mảng 2D là một mảng gồm các mảng 1D.

A diagram of a number

Description automatically generated

### 10.5.5 Bộ nhớ mảng ba chiều



### 10.5.6 Khởi tạo một mảng ba chiều

|  |
| --- |
| double cprogram[3][2][4] = {      {{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},      {{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},      {{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}  }; |

Chúng ta có thể có mảng với bất kỳ số chiều nào, tuy nhiên thường thì hầu hết các mảng mà ta tạo ra sẽ có một hoặc hai chiều.

## PHẦN 10.6 DEFINE MẢNG VÀ TRUY CẬP VÀO PHẦN TỬ MẢNG

|  |
| --- |
| #include <stdio.h>  #define ARRLEN (10)  int main(void) {      int n[ARRLEN]; // n là một mảng gồm 10 số nguyên  *size\_t* i, j; // Sử dụng size\_t để chỉ định vị trí bộ nhớ (địa chỉ) trong mảng      // Khởi tạo giá trị cho các phần tử trong mảng n      for (i = 0; i < ARRLEN; i++) {          n[i] = i + 100; // Đặt giá trị cho phần tử tại vị trí i là i + 100      }      // In giá trị của từng phần tử trong mảng n      for (j = 0; j < ARRLEN; j++) {          printf("Phan tu [%zu] = %d\n", j, n[j]);      }      return 0;  } |

Trong ngôn ngữ lập trình C, size\_t là một kiểu dữ liệu không dấu được sử dụng để địa chỉ bộ nhớ và làm chỉ mục (index) cho các mảng. size\_t đảm bảo rằng nó có đủ độ rộng để địa chỉ tất cả các vùng nhớ có thể có trên hệ thống.

Sử dụng size\_t để địa chỉ bộ nhớ và làm chỉ mục mảng là một cách chắc chắn và an toàn để đảm bảo rằng không có sai sót liên quan đến kích thước và chỉ mục của các mảng. Vì size\_t là kiểu không dấu, nó chỉ đại diện cho các giá trị không âm, điều này ngăn ngừa việc sử dụng chỉ mục âm, nhằm tránh những tình huống phức tạp và không mong muốn.

Trái lại, sử dụng số nguyên có dấu để địa chỉ bộ nhớ và làm chỉ mục mảng nên được xem xét là trường hợp đặc biệt và nên hạn chế trong trường hợp hiếm khi cần sử dụng các chỉ mục âm. Sử dụng số nguyên có dấu trong trường hợp cần thiết đặc biệt có thể gây ra các vấn đề về sai sót và xử lý phức tạp hơn do việc xử lý các giá trị âm.

**Kết quả**

|  |
| --- |
| Phan tu [0] = 100  Phan tu [1] = 101  Phan tu [2] = 102  Phan tu [3] = 103  Phan tu [4] = 104  Phan tu [5] = 105  Phan tu [6] = 106  Phan tu [7] = 107  Phan tu [8] = 108  Phan tu [9] = 109 |

## PHẦN 10.7 XÓA NỘI DUNG CỦA MẢNG( THIẾT LẬP GIÁ TRỊ BẰNG 0)

Đôi khi cần thiết phải đặt mảng về giá trị 0 sau khi đã khởi tạo nó.

|  |
| --- |
| #include <stdlib.h> /\* for EXIT\_SUCCESS \*/  #define ARRLEN (10)  int main(void)  {      int array[ARRLEN]; /\* Đã cấp phát nhưng chưa khởi tạo, vì không được xác định là static hoặc global. \*/  *size\_t* i;      for (i = 0; i < ARRLEN; ++i)      {          array[i] = 0;      }      return EXIT\_SUCCESS;  } |

Một phương pháp rút gọn chung cho vòng lặp trên là sử dụng **memset()** từ **<string.h>.** Truyền **array** như được hiển thị bên dưới sẽ làm cho nó rơi vào một con trỏ đến phần tử đầu tiên của nó.

|  |
| --- |
| memset(*array*, 0, ARRLEN \* sizeof(int)); /\* Sử dụng kiểu dữ liệu cụ thể (int ở đây). \*/ |

Hoặc

|  |
| --- |
| memset(*array*, 0, ARRLEN \* sizeof \**array*); /\* Sử dụng kích thước của kiểu dữ liệu mà con trỏ đang trỏ tới. \*/ |

Vì trong ví dụ này array là một mảng và không chỉ là một con trỏ đến phần tử đầu tiên của mảng (xem Phần 10.3: Độ dài mảng để hiểu tại sao điều này quan trọng), một tùy chọn thứ ba để thiết lập giá trị 0 cho mảng là:

|  |
| --- |
| memset(*array*, 0, sizeof *array*); /\* Sử dụng kích thước của mảng chính nó. \*/ |

## PHẦN 10.8 THIẾT LẬP GIÁ TRỊ TRONG MẢNG

Truy cập các giá trị trong mảng thường được thực hiện thông qua dấu ngoặc vuông:

|  |
| --- |
| int val;  int array[10];  /\* Thiết lập giá trị của phần tử thứ năm là 5: \*/  array[4] = 5;  /\* Cách viết trên tương đương với: \*/  \*(array + 4) = 5;  /\* Đọc giá trị của phần tử thứ năm: \*/  val = array[4]; |

Do tính chất của các toán hạng của toán tử + có thể hoán đổi vị trí (--> luật giao hoán), việc sau đây cũng tương đương:

|  |
| --- |
| \*(array + 4) = 5;  \*(4 + array) = 5; |

Tương tự, các câu lệnh sau cũng tương đương:

|  |
| --- |
| array[4] = 5;  4[array] = 5; /\* Kỳ lạ nhưng hợp lệ trong C ... \*/ |

Và các câu lệnh dưới đây cũng tương tự nhau:

|  |
| --- |
| val = array[4];  val = 4[array]; /\* Kỳ lạ nhưng hợp lệ trong C ... \*/ |

Trong C, không thực hiện bất kỳ kiểm tra ranh giới nào, việc truy cập nội dung bên ngoài mảng đã được khai báo sẽ không xác định (truy cập vào bộ nhớ vượt quá phần được cấp phát):

|  |
| --- |
| int val;  int array[10];  array[4] = 5; /\* ổn \*/  val = array[4]; /\* ổn \*/  array[19] = 20; /\* hành vi không xác định \*/  val = array[15]; /\* hành vi không xác định \*/ |

## PHẦN 10.9 CẤP PHÁT VÀ KHỞI TẠO GIÁ TRỊ BAN ĐẦU CHO MỘT MẢNG VỚI KÍCH THƯỚC DO NGƯỜI DÙNG XÁC ĐỊNH

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main (void)  {      int \*pdata;  *size\_t* n;      printf("Nhap kich thuoc cua mang: ");      fflush(stdout); /\* Đảm bảo in ra thông báo trên stdout được đệm. \*/      if (1 != scanf("%zu", &n)) /\* Nếu không hỗ trợ "zu" (trên Windows?), sử dụng "lu". \*/      {          fprintf("scanf() khong doc duoc gia tri hop le\n");          exit(EXIT\_FAILURE);      }      pdata = calloc(n, sizeof \*pdata);      if (NULL == pdata)      {          perror("calloc() that bai"); /\* In ra thông báo lỗi. \*/          exit(EXIT\_FAILURE);      }      free(pdata); /\* Giải phóng bộ nhớ. \*/      return EXIT\_SUCCESS;  } |

Chương trình này cố gắng đọc vào một giá trị số nguyên không dấu từ đầu vào tiêu chuẩn, sau đó cấp phát một khối bộ nhớ cho một mảng có n phần tử kiểu int bằng cách gọi hàm calloc(). Bộ nhớ này được khởi tạo giá trị 0 cho tất cả các phần tử. Trong trường hợp thành công, bộ nhớ được giải phóng bằng cách gọi hàm free().

## PHẦN 10.10 DUYỆT QUA MẢNG BẰNG CON TRỎ

|  |
| --- |
| #include <stdio.h>  #define SIZE (10)  int main()  {  *size\_t* i = 0;      int \*p = NULL;      int a[SIZE];        /\* Thiết lập giá trị cho mảng a là i\*i \*/      for(i = 0; i < SIZE; ++i)      {          a[i] = i \* i;      }        /\* Đọc các giá trị bằng cách sử dụng con trỏ \*/      for(p = a; p < a + SIZE; ++p)      {          printf("%d\n", \*p);      }      return 0;  } |

Trong đoạn mã này, trong việc khởi tạo biến p trong điều kiện vòng lặp for đầu tiên, mảng a sẽ rơi vào trạng thái con trỏ tới phần tử đầu tiên của nó, như làm vậy ở hầu hết các nơi mà biến mảng được sử dụng.

Sau đó, ++p thực hiện tính toán con trỏ trên con trỏ p và duyệt qua từng phần tử của mảng và trỏ đến chúng bằng cách giải tham chiếu với \*p.

# LINKED LISTS

## PHẦN 11.1 DOUBLY LINKED LIST (DANH SÁCH LIÊN KẾT KÉP)

Dưới đây là một ví dụ về mã nguồn hiển thị cách chèn các nút vào danh sách liên kết kép, cách đảo ngược danh sách và cách in danh sách theo thứ tự ngược.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  /\* Dữ liệu này không phải lúc nào cũng được lưu trữ trong một cấu trúc,   \*nhưng đôi khi nó được dùng để dễ sử dụng \*/  struct *Node* {      /\* Đôi khi một khóa cũng được lưu trữ và sử dụng trong các chức năng \*/      int data;      struct *Node*\* next;      struct *Node*\* previous;  };  void insert\_at\_beginning(struct *Node* \*\**pheadNode*, int *value*);  void insert\_at\_end(struct *Node* \*\**pheadNode*, int *value*);  void print\_list(struct *Node* \**headNode*);  void print\_list\_backwards(struct *Node* \**headNode*);  void free\_list(struct *Node* \**headNode*);  int main(void) {      /\* Đôi khi trong danh sách liên kết kép, nút cuối cùng cũng được lưu trữ \*/      struct *Node* \*head = NULL;        printf("Chen mot nut vao dau danh sach.\n");      insert\_at\_beginning(&head, 5);      print\_list(head);        printf("Chen mot nut vao dau, sau do in danh sach theo thu tu nguoc\n");      insert\_at\_beginning(&head, 10);      print\_list\_backwards(head);        printf("Chen mot nut vao cuoi, sau do in danh sach theo thu tu xuoi.\n");      insert\_at\_end(&head, 15);      print\_list(head);        free\_list(head);      return 0;  }  void print\_list\_backwards(struct *Node* \**headNode*) {      if (NULL == *headNode*) {          return;      }      /\*      Lặp qua danh sách và khi chúng ta đi đến cuối,      lặp ngược lại để in ra các mục theo thứ tự ngược lại      (điều này được thực hiện với con trỏ tới nút trước đó).      Điều này thậm chí có thể được thực hiện dễ dàng hơn      nếu một con trỏ tới nút cuối cùng được lưu trữ.      \*/      struct *Node* \*i = *headNode*;      while (i->next != NULL) {          i = i->next;/\* Di chuyển đến cuối danh sách \*/      }      while (i != NULL) {          printf("Gia tri: %d\n", i->data);          i = i->previous;      }  }  void print\_list(struct *Node* \**headNode*) {      /\* Duyệt qua danh sách và in ra thành viên dữ liệu của mỗi nút \*/      struct *Node* \*i;      for (i = *headNode*; i != NULL; i = i->next) {          printf("Gia tri: %d\n", i->data);      }  }  void insert\_at\_beginning(struct *Node* \*\**pheadNode*, int *value*) {      struct *Node* \*currentNode;      if (NULL == *pheadNode*) {          return;      }      /\*      Điều này được thực hiện tương tự như cách chúng ta chèn một nút vào đầu danh sách liên kết đơn,      thay vào đó chúng ta cũng đặt thành viên trước đó của cấu trúc      \*/      currentNode = malloc(sizeof \*currentNode);      currentNode->next = NULL;      currentNode->previous = NULL;      currentNode->data = *value*;      if (\**pheadNode* == NULL) { /\* Danh sách rỗng \*/          \**pheadNode* = currentNode;          return;      }      currentNode->next = \**pheadNode*;      (\**pheadNode*)->previous = currentNode;      \**pheadNode* = currentNode;  }  void insert\_at\_end(struct *Node* \*\**pheadNode*, int *value*) {      struct *Node* \*currentNode;      if (NULL == *pheadNode*) {          return;      }      /\*      Điều này một lần nữa có thể được thực hiện dễ dàng bằng cách có thể có phần tử trước đó.      Nó cũng sẽ hữu ích hơn nếu có một con trỏ tới nút cuối cùng, thường được sử dụng.      \*/      currentNode = malloc(sizeof \*currentNode);      struct *Node* \*i = \**pheadNode*;      currentNode->data = *value*;      currentNode->next = NULL;      currentNode->previous = NULL;      if (\**pheadNode* == NULL) {          \**pheadNode* = currentNode;          return;      }      while (i->next != NULL) { /\* Đi đến cuối danh sách \*/          i = i->next;      }      i->next = currentNode;      currentNode->previous = i;  }  void free\_list(struct *Node* \**node*) {      while (*node* != NULL) {          struct *Node* \*next = *node*->next;          free(*node*);  *node* = next;      }  } |

Lưu ý rằng đôi khi, việc lưu trữ một con trỏ tới nút cuối cùng là hữu ích (chỉ đơn giản là có thể chuyển thẳng đến cuối danh sách sẽ hiệu quả hơn là cần phải lặp lại từ đầu đến cuối):

|  |
| --- |
| struct *Node* \*lastNode = NULL; |

Trong trường hợp đó, cần cập nhật nó khi thay đổi danh sách.

Đôi khi, một khóa cũng được sử dụng để xác định các phần tử. Nó chỉ đơn giản là một thành viên của cấu trúc Node:

|  |
| --- |
| struct *Node* {      int data;      int key;      struct *Node*\* next;      struct *Node*\* previous;  }; |

Sau đó, khóa này được sử dụng khi bất kỳ tác vụ nào được thực hiện trên một phần tử cụ thể, chẳng hạn như xóa phần tử

## PHẦN 11.2 REVERSING A LINKED LIST (ĐẢO NGƯỢC DANH SÁCH LIÊN KẾT)

Bạn cũng có thể thực hiện nhiệm vụ này theo cách đệ quy, nhưng tôi đã chọn trong ví dụ này để sử dụng phương pháp lặp. Tác vụ này hữu ích nếu bạn đang chèn tất cả các nút của mình vào đầu danh sách được liên kết. Đây là một ví dụ:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #define NUM\_ITEMS 10  struct *Node* {      int data;      struct *Node* \*next;  };  void chen\_nut(struct *Node* \*\**dauDanhSach*, int *giaTriNut*, int *viTri*);  void in\_danh\_sach(struct *Node* \**dauDanhSach*);  void dao\_nguoc\_danh\_sach(struct *Node* \*\**dauDanhSach*);  int main(void) {      int i;      struct *Node* \*dauDanhSach = NULL;        // Tạo danh sách ban đầu từ 1 đến NUM\_ITEMS      for(i = 1; i <= NUM\_ITEMS; i++) {          chen\_nut(&dauDanhSach, i, i);      }        // In ra danh sách ban đầu      printf("Danh sach ban dau:\n");      in\_danh\_sach(dauDanhSach);        // Đảo ngược danh sách      printf("Danh sach sau khi dao nguoc:\n");      dao\_nguoc\_danh\_sach(&dauDanhSach);      in\_danh\_sach(dauDanhSach);        return 0;  }  // Hàm chèn một nút mới vào danh sách liên kết đơn tại vị trí đã cho  void chen\_nut(struct *Node* \*\**dauDanhSach*, int *giaTriNut*, int *viTri*) {      int i;      struct *Node* \*nutHienTai = (struct *Node* \*)malloc(sizeof(struct *Node*));      struct *Node* \*nutTruocViTri = \**dauDanhSach*;      nutHienTai->data = *giaTriNut*;        // Nếu vị trí là 1, thì nút mới trở thành nút đầu danh sách      if(*viTri* == 1) {          nutHienTai->next = \**dauDanhSach*;          \**dauDanhSach* = nutHienTai;          return;      }        // Di chuyển đến nút trước vị trí cần chèn      for(i = 0; i < *viTri* - 2; i++) {          nutTruocViTri = nutTruocViTri->next;      }        nutHienTai->next = nutTruocViTri->next;      nutTruocViTri->next = nutHienTai;  }  // Hàm in ra các giá trị trong danh sách liên kết đơn  void in\_danh\_sach(struct *Node* \**dauDanhSach*) {      struct *Node* \*dieuHuong;      for(dieuHuong = *dauDanhSach*; dieuHuong != NULL; dieuHuong = dieuHuong->next) {          printf("Gia tri: %d\n", dieuHuong->data);      }  }  // Hàm đảo ngược danh sách liên kết đơn  void dao\_nguoc\_danh\_sach(struct *Node* \*\**dauDanhSach*) {      struct *Node* \*dieuHuong = \**dauDanhSach*;      struct *Node* \*nutTruoc = NULL;      struct *Node* \*nutSau = NULL;        while(dieuHuong != NULL) {          nutSau = dieuHuong->next;          dieuHuong->next = nutTruoc;          nutTruoc = dieuHuong;          dieuHuong = nutSau;      }        \**dauDanhSach* = nutTruoc;  } |

**Giải thích cho phương pháp REVERSING A LINKED LIST (ĐẢO NGƯỢC DANH SÁCH LIÊN KẾT)**

Chúng tôi bắt đầu nutTruoc là NULL, vì chúng tôi biết trong lần lặp đầu tiên của vòng lặp, nếu chúng tôi đang tìm kiếm nút trước nút đầu tiên, thì nó sẽ là NULL. Nút đầu tiên sẽ trở thành nút cuối cùng trong danh sách và biến tiếp theo đương nhiên phải là NULL.

Về cơ bản, khái niệm đảo ngược danh sách liên kết ở đây là chúng ta thực sự đảo ngược chính các liên kết. Thành viên tiếp theo của mỗi nút sẽ trở thành nút trước nó, giống như:

|  |
| --- |
| Head -> 1 -> 2 -> 3 -> 4 -> 5 |

Trong đó mỗi số đại diện cho một nút. Danh sách này sẽ trở thành:

|  |
| --- |
| 1 <- 2 <- 3 <- 4 <- 5 <- Head |

Cuối cùng, thay vào đó, phần đầu sẽ trỏ đến nút thứ 5 và mỗi nút sẽ trỏ đến nút trước nó.

Nút 1 phải trỏ đến NULL vì không có gì trước nó. Nút 2 phải trỏ đến nút 1, nút 3 phải trỏ đến nút 2, v.v.

Tuy nhiên, có một vấn đề nhỏ với phương pháp này. Nếu chúng ta ngắt liên kết đến nút tiếp theo và thay đổi nó thành nút trước đó, chúng ta sẽ không thể đi qua nút tiếp theo trong danh sách vì liên kết đến nó đã biến mất.

Giải pháp cho vấn đề này là chỉ cần lưu trữ phần tử tiếp theo trong một biến (nextNode) trước khi thay đổi liên kết.

## PHẦN 11.3 INSERTING A NODE AT THE NTH POSITION (CHÈN NÚT Ở VỊ TRÍ THỨ N)

Cho đến giờ, chúng ta đã xem xét việc chèn một nút vào đầu danh sách liên kết đơn. Tuy nhiên, hầu hết các trường hợp bạn cũng muốn có thể chèn các nút ở nơi khác. Đoạn mã được viết dưới đây cho thấy cách có thể viết hàm insert() để chèn các nút vào bất kỳ đâu trong danh sách được liên kết.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  struct *Node* {      int data;      struct *Node*\* next;  };  struct *Node*\* insert(struct *Node*\* *head*, int *value*, *size\_t* *position*);  void print\_list(struct *Node*\* *head*);  int main(int *argc*, char\* *argv*[]) {      struct *Node*\* head = NULL; /\* Khởi tạo danh sách rỗng \*/      /\* Chèn các node vào vị trí và giá trị tương ứng: \*/      head = insert(head, 1, 0);      head = insert(head, 100, 1);      head = insert(head, 21, 2);      head = insert(head, 2, 3);      head = insert(head, 5, 4);      head = insert(head, 42, 2);      print\_list(head);        return 0;  }  struct *Node*\* insert(struct *Node*\* *head*, int *value*, *size\_t* *position*) {  *size\_t* i = 0;      struct *Node*\* currentNode;      /\* Tạo node mới \*/      currentNode = malloc(sizeof \*currentNode);      /\* Kiểm tra thành công của malloc() ở đây! \*/        /\* Gán giá trị dữ liệu \*/      currentNode->data = *value*;      /\* Con trỏ trỏ tới trường 'next' mà chúng ta phải liên kết với node mới.       \* Bằng cách khởi tạo nó thành &head, chúng ta xử lý trường hợp chèn vào đầu danh sách. \*/      struct *Node*\*\* nextForPosition = &*head*;      /\* Lặp để lấy trường 'next' mà chúng ta đang tìm kiếm.       \* Lưu ý: Chèn vào cuối nếu vị trí lớn hơn số phần tử hiện tại. \*/      for (i = 0; i < *position* && \*nextForPosition != NULL; i++) {          /\* nextForPosition trỏ tới trường 'next' của node.           \* Vì vậy, \*nextForPosition là một con trỏ tới node tiếp theo.           \* Cập nhật nó thành một con trỏ tới trường 'next' của node tiếp theo. \*/          nextForPosition = &(\*nextForPosition)->next;      }      /\* Ở đây, chúng ta lấy liên kết tới node tiếp theo (node mà node mới của chúng ta nên trỏ tới)       \* bằng cách giải tham chiếu nextForPosition, con trỏ này trỏ tới trường 'next' của node       \* đang ở vị trí chúng ta muốn chèn node mới vào.       \* Chúng ta gán liên kết này cho giá trị next của node mới. \*/      currentNode->next = \*nextForPosition;      /\* Bây giờ, chúng ta muốn sửa liên kết của node trước vị trí của node mới:       \* nó sẽ được thay đổi thành con trỏ tới node mới của chúng ta. \*/      \*nextForPosition = currentNode;      return *head*;  }  void print\_list(struct *Node*\* *head*) {      /\* Duyệt qua danh sách các node và in ra giá trị dữ liệu trong mỗi node \*/      struct *Node*\* i = *head*;      while (i != NULL) {          printf("%d\n", i->data);          i = i->next;      }  } |

## PHẦN 11.4 INSERTING A NODE AT THE BEGINNING OF A SINGLY LINKED LIST (CHÈN NÚT VÀO ĐẦU DANH SÁCH LIÊN KẾT ĐƠN)

Đoạn mã dưới đây sẽ nhắc các số và tiếp tục thêm chúng vào đầu danh sách được liên kết.

|  |
| --- |
| /\* Chương trình này sẽ thể hiện cách chèn một node vào đầu danh sách liên kết \*/  #include <stdio.h>  #include <stdlib.h>  struct *Node* {      int data;      struct *Node*\* next;  };  void insert\_node(struct *Node*\*\* *head*, int *nodeValue*);  void print\_list(struct *Node*\* *head*);  int main(int *argc*, char\* *argv*[]) {      struct *Node*\* headNode;      headNode = NULL; /\* Khởi tạo con trỏ của node đầu tiên thành NULL. \*/  *size\_t* listSize, i;      do {          printf("Nhập số lượng số bạn muốn nhập:\n");      } while (1 != scanf("%zu", &listSize));      for (i = 0; i < listSize; i++) {          int numToAdd;          do {              printf("Nhập một số:\n");          } while (1 != scanf("%d", &numToAdd));          insert\_node(&headNode, numToAdd);          printf("Danh sách hiện tại sau khi chèn node mới: \n");          print\_list(headNode);      }      return 0;  }  void print\_list(struct *Node*\* *head*) {      struct *Node*\* currentNode = *head*;      /\* Lặp qua từng liên kết \*/      while (currentNode != NULL) {          printf("Giá trị: %d\n", currentNode->data);          currentNode = currentNode->next;      }  }  void insert\_node(struct *Node*\*\* *head*, int *nodeValue*) {      struct *Node*\* currentNode = malloc(sizeof \*currentNode);      currentNode->data = *nodeValue*;      currentNode->next = (\**head*);      \**head* = currentNode;  } |

**Giải thích cho việc chèn các nút**

Để hiểu cách chúng tôi thêm các nút ngay từ đầu, hãy xem xét các tình huống có thể xảy ra:

1. Danh sách trống, vì vậy chúng ta cần thêm một nút mới. Trong trường hợp đó, bộ nhớ của chúng ta trông như thế này khi HEAD là một con trỏ tới nút đầu tiên:

|  |
| --- |
| | HEAD | --> NULL |

Dòng currentNode->next = \*headNode; sẽ gán giá trị của currentNode->next là NULL vì headNode ban đầu bắt đầu với giá trị NULL.

Bây giờ, chúng tôi muốn đặt con trỏ nút đầu trỏ đến nút hiện tại của chúng tôi.

|  |
| --- |
| ----- -------------  |HEAD | --> |CURRENTNODE| --> NULL /\* Nút đầu trỏ tới nút hiện tại \*/  ----- ------------- |

Điều này được thực hiện với \*headNode = currentNode;

2. Danh sách đã được điền; chúng ta cần thêm một nút mới vào đầu. Để đơn giản, hãy bắt đầu với 1 nút:

|  |
| --- |
| ----- -----------  HEAD --> FIRST NODE --> NULL  ----- ----------- |

Với currentNode->next = \*headNode, cấu trúc dữ liệu sẽ như sau:

|  |
| --- |
| --------- ----- ---------------------  currentNode --> HEAD --> POINTER TO FIRST NODE --> NULL  --------- ----- --------------------- |

Mà, rõ ràng cần phải được thay đổi vì \*headNode nên trỏ đến currentNode.

|  |
| --- |
| ---- ----------- ---------------  HEAD -> currentNode --> NODE -> NULL  ---- ----------- --------------- |

Điều này được thực hiện với \*headNode = currentNode;

# ENUMERATIONS

## PHẦN 12.1 SIMPLE ENUMERATION (ENUMERATION ĐƠN GIẢN)

Một "enumeration" là một kiểu dữ liệu do người dùng định nghĩa, bao gồm các hằng số số nguyên và mỗi hằng số số nguyên được đặt tên. Từ khóa "enum" được sử dụng để định nghĩa kiểu dữ liệu được liệt kê.

Nếu bạn sử dụng "enum" thay vì "int" hoặc "string/char\*", bạn sẽ tăng tính kiểm tra thời gian biên dịch và tránh lỗi khi truyền các hằng số không hợp lệ, cũng như làm cho các giá trị hợp lệ sử dụng trở nên rõ ràng và dễ hiểu hơn.

**Ví dụ 1:**

|  |
| --- |
| enum *color*{ RED, GREEN, BLUE };  void printColor(enum *color* *chosenColor*)  {      const char \*color\_name = "Invalid color";      switch (*chosenColor*)      {          case RED:              color\_name = "RED";              break;            case GREEN:              color\_name = "GREEN";              break;          case BLUE:              color\_name = "BLUE";              break;      }      printf("%s\n", color\_name);  } |

Với một hàm main được định nghĩa như sau (ví dụ):

|  |
| --- |
| int main(){      enum *color* chosenColor;      printf("Enter a number between 0 and 2");      scanf("%d", (int\*)&chosenColor);      printColor(chosenColor);      return 0;  } |

**Ví dụ 2:**

(Ví dụ này sử dụng các bộ khởi tạo được chỉ định đã được chuẩn hóa từ C99.)

|  |
| --- |
| enum *week*{ MON, TUE, WED, THU, FRI, SAT, SUN };    static const char\* const dow[] = {      [MON] = "Mon", [TUE] = "Tue", [WED] = "Wed",      [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };    void printDayOfWeek(enum *week* *day*)  {      printf("%s\n", dow[*day*]);  } |

Ví dụ tương tự sử dụng kiểm tra phạm vi:

|  |
| --- |
| enum *week*{ DOW\_INVALID = -1,      MON, TUE, WED, THU, FRI, SAT, SUN,      DOW\_MAX };    static const char\* const dow[] = {      [MON] = "Mon", [TUE] = "Tue", [WED] = "Wed",      [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };    void printDayOfWeek(enum *week* *day*)  {      assert(*day* > DOW\_INVALID && *day* < DOW\_MAX);      printf("%s\n", dow[*day*]);  } |

## PHẦN 12.2 ENUMERATION CONSTANT WITHOUT TYPENAME (HẰNG SỐ ENUMERATION KHÔNG CẦN TÊN KIỂU)

Kiểu enumerations cũng có thể được khai báo mà không cần đặt tên cho chúng:

|  |
| --- |
| enum { buffersize = 256, };  static unsigned char buffer [buffersize] = { 0 }; |

Điều này cho phép chúng ta xác định các hằng số thời gian biên dịch kiểu int mà có thể như trong ví dụ này được sử dụng làm độ dài mảng.

## PHẦN 12.3 ENUMERATION WITH DUPLICATE VALUE (ENUMERATION VỚI GIÁ TRỊ TRÙNG LẶP)

Một enumeration không nhất thiết phải là duy nhất:

|  |
| --- |
| #include <stdlib.h> /\* cho EXIT\_SUCCESS \*/  #include <stdio.h> /\* cho printf() \*/  enum *Dupes*  {      Base,          /\* Lấy giá trị 0 \*/      One,           /\* Lấy giá trị của Base + 1 \*/      Two,           /\* Lấy giá trị của One + 1 \*/      Negative = -1, /\* Gán giá trị là -1 \*/      AnotherZero    /\* Lấy giá trị của Negative + 1 == 0, thật đáng tiếc \*/  };  int main(void)  {      printf("Base = %d\n", Base);      printf("One = %d\n", One);      printf("Two = %d\n", Two);      printf("Negative = %d\n", Negative);      printf("AnotherZero = %d\n", AnotherZero);      return EXIT\_SUCCESS;  } |

Kết quả:

|  |
| --- |
| Base = 0  One = 1  Two = 2  Negative = -1  AnotherZero = 0 |

## PHẦN 12.4 TYPEDEF ENUM

Có một số khả năng và quy ước để đặt tên cho một enumeration. Đầu tiên là sử dụng một thẻ (tag) ngay sau từ khóa "**enum**".

|  |
| --- |
| enum *color*  {      RED,      GREEN,      BLUE  }; |

Kiểu liệt kê này phải luôn được sử dụng cùng với từ khóa "enum" và thẻ tương ứng như sau:

|  |
| --- |
| enum *color* chosenColor = RED; |

Nếu chúng ta sử dụng "**typedef**" trực tiếp khi khai báo **enum**, chúng ta có thể bỏ qua thẻ và sau đó sử dụng kiểu mà không có từ khóa "enum":

|  |
| --- |
| typedef enum  {      RED,      GREEN,      BLUE  } *color*; |

Sau đó, chúng ta có thể sử dụng "color" mà không cần từ khóa "enum":

|  |
| --- |
| *color* chosenColor = RED; |

Tuy nhiên, trong trường hợp này, chúng ta không thể sử dụng "enum color" vì chúng ta không sử dụng thẻ trong định nghĩa. Một quy ước phổ biến là sử dụng cả hai, để cùng một tên có thể được sử dụng với hoặc không có từ khóa "enum". Điều này có lợi thế đặc biệt là tương thích với C++.

|  |
| --- |
| enum *color* /\* như ví dụ đầu tiên \*/  {      RED,      GREEN,      BLUE  };  typedef enum *color* *color*; /\* cũng là một typedef với cùng một thay đổi \*/  *color* chosenColor = RED;  enum *color* defaultColor = BLUE; |

Hàm

|  |
| --- |
| void printColor()  {      if (chosenColor == RED)      {          printf("RED\n");      }      else if (chosenColor == GREEN)      {          printf("GREEN\n");      }      else if (chosenColor == BLUE)      {          printf("BLUE\n");      }  } |

Để biết thêm về "typedef", xem phần "Typedef".

# STRUCTS (CẤU TRÚC)

Cấu trúc (Structs) cung cấp một cách để nhóm một tập hợp các biến liên quan thuộc nhiều loại khác nhau vào một đơn vị bộ nhớ. Toàn bộ cấu trúc có thể được tham chiếu bằng một tên hoặc con trỏ duy nhất; các thành viên cấu trúc cũng có thể được truy cập riêng lẻ. Các cấu trúc có thể được truyề006E đến các hàm và được trả về từ các hàm. Chúng được định nghĩa bằng cách sử dụng từ khóa **struct**.

## PHẦN 13.1 FLEXIBLE ARRAY MEMBERS (THÀNH PHẦN MẢNG LINH HOẠT)

### 13.1.1 Khai báo kiểu dữ liệu

Một cấu trúc có ít nhất một thành viên có thể chứa thêm một thành viên mảng có độ dài không xác định ở cuối cấu trúc. Đây được gọi là thành viên mảng linh hoạt:

|  |
| --- |
| struct *ex1*  {  *size\_t* foo;      int flex[]  };  struct *ex2\_header*  {      int foo;      char bar;  };  struct *ex2*  {      struct *ex2\_header*;      int flex;  };  /\* Hợp nhất cấu trúc ex2\_header và ex2. \*/  struct *ex3*  {      int foo;      char bar      int flex[];  }; |

### 13.1.2 Ảnh hưởng đến kích thước và phần đệm

Thành viên mảng linh hoạt được coi là không có kích thước khi tính toán kích thước của cấu trúc, mặc dù phần đệm giữa thành viên đó và thành viên trước đó của cấu trúc vẫn có thể tồn tại:

|  |
| --- |
| /\* In ra "8,8" trên máy tính của tôi, vì vậy không có phần trống. \*/  printf("%zu,%zu\n", sizeof(size\_t), sizeof(struct ex1));  /\* Cũng in ra "8,8" trên máy tính của tôi, nên không có phần trống trong cấu trúc ex2 chính nó. \*/  printf("%zu,%zu\n", sizeof(struct ex2\_header), sizeof(struct ex2));  /\* In ra "5,8" trên máy tính của tôi, nên có 3 byte phần trống. \*/  printf("%zu,%zu\n", sizeof(int) + sizeof(char), sizeof(struct ex3)); |

Thành viên mảng linh hoạt (Flexible Array Member) được coi là có kiểu mảng không hoàn chỉnh, do đó không thể tính kích thước của nó bằng cách sử dụng "**sizeof**".

### 13.1.3 Sử dụng

Bạn có thể khai báo và khởi tạo một đối tượng có kiểu cấu trúc chứa một thành viên mảng linh hoạt (Flexible Array Member), nhưng bạn không được thử khởi tạo thành viên mảng linh hoạt (Flexible Array Member) vì nó được xử lý như không tồn tại. Thử làm điều này sẽ bị cấm và gây ra lỗi biên dịch.

Tương tự, bạn không nên cố gắng gán giá trị cho bất kỳ phần tử nào của thành viên mảng linh hoạt (Flexible Array Member) khi khai báo cấu trúc theo cách này vì có thể không có đủ phần trống ở cuối cấu trúc để chứa bất kỳ đối tượng nào cần thiết cho thành viên mảng linh hoạt (Flexible Array Member). Trình biên dịch không nhất thiết phải ngăn bạn làm điều này, do đó có thể dẫn đến hành vi không xác định.

|  |
| --- |
| /\* Không hợp lệ: không thể khởi tạo thành viên mảng linh hoạt (Flexible Array Member) \*/  struct *ex1* e1 = {1, {2, 3}};  /\* Không hợp lệ: hdr={foo=1, bar=2} OK, nhưng không thể khởi tạo thành viên mảng linh hoạt (Flexible Array Member) \*/  struct *ex2* e2 = {{1, 2}, {3}};  /\* Hợp lệ: khởi tạo thành viên foo=1, bar=2 \*/  struct *ex3* e3 = {1, 2};  e1.flex[0] = 3; /\* Hành vi không xác định, trong trường hợp của tôi \*/  e3.flex[0] = 2; /\* Hành vi không xác định lại \*/  e2.flex[0] = e3.flex[0]; /\* Hành vi không xác định \*/ |

Thay vào đó, bạn có thể chọn sử dụng malloc, calloc hoặc realloc để cấp phát cấu trúc có lưu trữ bổ sung và sau đó giải phóng nó, cho phép bạn sử dụng thành viên mảng linh hoạt (Flexible Array Member) theo ý muốn:

|  |
| --- |
| /\* Hợp lệ: cấp phát một đối tượng có kiểu cấu trúc ex1 cùng với một mảng gồm 2 số nguyên \*/  struct *ex1* \*pe1 = malloc(sizeof(\*pe1) + 2 \* sizeof(pe1->flex[0]));  /\* Hợp lệ: cấp phát một đối tượng có kiểu cấu trúc ex2 cùng với một mảng gồm 4 số nguyên \*/  struct *ex2* \*pe2 = malloc(sizeof(struct *ex2*) + sizeof(int[4]));  /\* Hợp lệ: cấp phát 5 đối tượng kiểu cấu trúc ex3 cùng với một mảng gồm 3 số nguyên cho mỗi đối tượng \*/  struct *ex3* \*pe3 = malloc(5 \* (sizeof(\*pe3) + sizeof(int[3])));  pe1->flex[0] = 3; /\* có hiệu lực \*/  pe3[0]->flex[0] = pe1->flex[0]; /\* có hiệu lực \*/ |

### 13.1.4 Struct hack

Phiên bản C trước C99 không hỗ trợ "flexible array members," bạn có thể sử dụng một kỹ thuật được gọi là "struct hack" để thay thế. Trong"struct hack," bạn sẽ khai báo một mảng có kích thước là 1 trong cấu trúc để thay thế "flexible array members."

|  |
| --- |
| struct *ex1*  {  *size\_t* foo;      int flex[1];  }; |

Tuy nhiên, điều này sẽ ảnh hưởng đến kích thước của cấu trúc, không giống như một thành viên mảng linh hoạt thực sự:

|  |
| --- |
| /\* In "8,4,16" trên máy của tôi, biểu thị rằng có 4 byte đệm. \*/  printf("%d,%d,%d\n", (int)sizeof(size\_t), (int)sizeof(int[1]), (int)sizeof(struct ex1)); |

Để sử dụng thành viên "flex" như một thành viên mảng linh hoạt, bạn sẽ cấp phát bộ nhớ cho nó bằng cách sử dụng malloc như đã được trình bày ở trên, nhưng thay vì sử dụng **sizeof**(\*pe1) (hoặc **sizeof(struct** ex1)), bạn sẽ thay thế bằng offsetof(**struct** ex1, flex) hoặc biểu thức dài hơn, không phụ thuộc vào kiểu dữ liệu **sizeof**(\*pe1) - **sizeof**(pe1->flex). Hoặc có thể trừ đi 1 từ độ dài mong muốn của mảng "linh hoạt" vì nó đã được bao gồm trong kích thước của cấu trúc, giả sử độ dài mong muốn là lớn hơn 0. Cùng logic này có thể được áp dụng vào các ví dụ sử dụng khác.

### 13.1.5 Khả năng tương thích

Nếu muốn tương thích với các trình biên dịch không hỗ trợ các thành viên mảng linh hoạt, bạn có thể sử dụng macro được xác định như FLEXMEMB\_SIZE bên dưới:

|  |
| --- |
| #if \_\_STDC\_VERSION\_\_ < 199901L  #define FLEXMEMB\_SIZE 1  #else  #define FLEXMEMB\_SIZE /\* nothing \*/  #endif  struct *ex1*  {  *size\_t* foo;      int flex[FLEXMEMB\_SIZE];  }; |

Khi cấp phát các đối tượng, bạn nên sử dụng offsetof(struct ex1, flex) để tham chiếu đến kích thước của cấu trúc (loại trừ thành viên mảng linh hoạt) vì nó là biểu thức duy nhất giữ nguyên tính nhất quán giữa các trình biên dịch hỗ trợ thành viên mảng linh hoạt và các trình biên dịch không hỗ trợ:

|  |
| --- |
| struct *ex1* \*pe10 = malloc(offsetof(struct *ex1*, flex) + n \* sizeof(pe10->flex[0])); |

Một phương pháp thay thế khác là sử dụng tiền xử lý để trừ 1 từ độ dài được chỉ định. Tuy nhiên, do tiềm năng tăng cao về không nhất quán và lỗi của con người trong hình thức này, tôi đã di chuyển logic vào một hàm riêng biệt:

|  |
| --- |
| struct *ex1* \*ex1\_alloc(*size\_t* *n*)  {      struct *ex1* tmp;  #if \_\_STDC\_VERSION\_\_ < 199901L      if (n != 0)          n--;  #endif      return malloc(sizeof(tmp) + *n* \* sizeof(tmp.flex[0]));  }  ...  /\* cấp phát một đối tượng ex1 với mảng "flex" có độ dài là 3 \*/  struct ex1 \*pe1 = ex1\_alloc(3); |

## PHẦN 13.2 TYPEDEF STRUCTS

Kết hợp từ khóa typedef với struct có thể làm mã nguồn rõ ràng hơn. Ví dụ:

|  |
| --- |
| typedef struct  {      int x, y;  } *Point*; |

Thay vì:

|  |
| --- |
| struct *Point*  {      int x, y;  }; |

có thể được khai báo như sau:

|  |
| --- |
| *Point* point; |

Thay vì

|  |
| --- |
| struct *Point* point; |

Một cách tốt hơn là sử dụng cách khai báo sau:

|  |
| --- |
| typedef struct *Point* *Point*;  struct *Point*  {      int x, y;  }; |

để có lợi ích của cả hai cách định nghĩa point. Khai báo như vậy là tiện lợi nhất nếu bạn đã học C++ trước, nơi bạn có thể bỏ qua từ khóa struct nếu tên không gây nhiễu.

typedef cho các tên struct có thể xung đột với các bộ nhớ đệm khác trong chương trình. Một số người coi điều này là một nhược điểm, nhưng đối với hầu hết mọi người, khi có một struct và một bộ nhớ đệm khác có cùng tên sẽ gây khó chịu.

Một điểm nổi tiếng là hàm stat của POSIX:

|  |
| --- |
| int stat(const char \**pathname*, struct *stat* \**buf*); |

trong đó bạn thấy hàm stat có một đối số là struct stat.

Các struct được định nghĩa bằng typedef mà không có tên thẻ luôn yêu cầu rằng toàn bộ khai báo struct phải được hiển thị trong mã sử dụng nó. Toàn bộ khai báo struct phải được đặt trong một tệp header.

Coi như:

|  |
| --- |
| #include "bar.h"  struct *foo*  {      bar \*aBar;  }; |

Khi sử dụng **typedef** cho **struct** mà không có tên thẻ, tệp **bar.h** luôn phải bao gồm toàn bộ định nghĩa của cấu trúc **bar**. Nếu chúng ta sử dụng typedef như sau trong tệp bar.h:

|  |
| --- |
| typedef struct *bar* *bar*; |

trong **bar.h**, các chi tiết của cấu trúc bar có thể được ẩn đi.

Xem thêm về Typedef

## PHẦN 13.3 POINTERS TO STRUCTS (CON TRỎ TỚI CẤU TRÚC)

Khi bạn có một biến chứa một cấu trúc, bạn có thể truy cập các trường của nó bằng toán tử chấm (.). Tuy nhiên, nếu bạn có một con trỏ tới một cấu trúc, điều này sẽ không hoạt động. Bạn phải sử dụng toán tử mũi tên (->) để truy cập các trường của nó. Dưới đây là một ví dụ về cách triển khai cực kỳ đơn giản (một số người có thể gọi là "đơn giản và kém chất lượng") của một ngăn xếp sử dụng con trỏ tới cấu trúc và thể hiện toán tử mũi tên.

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  /\* cấu trúc \*/  struct *stack*  {      struct *node* \*top;      int size;  };  struct *node*  {      int data;      struct *node* \*next;  };  /\* khai báo hàm \*/  int push(int, struct *stack* \*);  int pop(struct *stack* \*);  void destroy(struct *stack* \*);  int main(void)  {      int result = EXIT\_SUCCESS;  *size\_t* i;      /\* cấp phát bộ nhớ cho một struct stack và ghi nhận con trỏ của nó \*/      struct *stack* \*stack = malloc(sizeof \*stack);      if (NULL == stack)      {          perror("malloc() failed");          return EXIT\_FAILURE;      }      /\* khởi tạo ngăn xếp \*/      stack->top = NULL;      stack->size = 0;      /\* đẩy 10 số nguyên vào ngăn xếp \*/      {          int data = 0;          for (i = 0; i < 10; i++)          {              printf("Pushing: %d\n", data);              if (-1 == push(data, stack))              {                  perror("push() failed");                  result = EXIT\_FAILURE;                  break;              }              ++data;          }      }      if (EXIT\_SUCCESS == result)      {          /\* lấy ra 5 số nguyên từ ngăn xếp \*/          for (i = 0; i < 5; i++)          {              printf("Popped: %i\n", pop(stack));          }      }      /\* hủy ngăn xếp \*/      destroy(stack);      return result;  }  /\* Đẩy một giá trị vào ngăn xếp. \*/  /\* Trả về 0 nếu thành công và -1 nếu thất bại. \*/  int push(int *data*, struct *stack* \**stack*)  {      int result = 0;      /\* cấp phát bộ nhớ cho nút mới \*/      struct *node* \*new\_node = malloc(sizeof \*new\_node);      if (NULL == new\_node)      {          result = -1;      }      else      {          new\_node->data = *data*;          new\_node->next = *stack*->top;  *stack*->top = new\_node;  *stack*->size++;      }      return result;  }  /\* Lấy ra một giá trị từ ngăn xếp. \*/  /\* Trả về giá trị lấy ra từ ngăn xếp \*/  int pop(struct *stack* \**stack*)  {      struct *node* \*top = *stack*->top;      int data = top->data;  *stack*->top = top->next;  *stack*->size--;      free(top);      return data;  }  /\* hủy ngăn xếp \*/  void destroy(struct *stack* \**stack*)  {      /\* giải phóng tất cả con trỏ \*/      while (*stack*->top != NULL)      {          pop(*stack*);      }  } |

## PHẦN 13.4 PASSING STRUCTS TO FUNCTIONS (TRUYỀN STRUCTS VÀO HÀM)

Trong C, tất cả các đối số được truyền vào các hàm theo giá trị, bao gồm cả các cấu trúc. Đối với các cấu trúc nhỏ, điều này rất tốt vì không có chi phí từ việc truy cập dữ liệu thông qua con trỏ. Tuy nhiên, điều này cũng khiến việc truyền ngẫu nhiên một cấu trúc lớn dễ dàng, dẫn đến hiệu suất kém, đặc biệt nếu lập trình viên đã sử dụng các ngôn ngữ khác nơi đối số được truyền theo tham chiếu.

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  /\* cấu trúc \*/  struct *coordinates*  {      int x;      int y;      int z;  };  /\* Hàm trả về cấu trúc nhỏ bằng giá trị, rất nhanh \*/  struct *coordinates* move(struct *coordinates* *position*, struct *coordinates* *movement*)  {  *position*.x += *movement*.x;  *position*.y += *movement*.y;  *position*.z += *movement*.z;      return *position*;  }  /\* Cấu trúc rất lớn \*/  struct *lotsOfData*  {      int param1;      char param2[80000];  };  /\* Hàm trả về cấu trúc lớn bằng giá trị, rất chậm! \*/  /\* Với kích thước lớn của cấu trúc này, có thể gây tràn ngăn xếp (stack overflow) \*/  struct *lotsOfData* doubleParam1(struct *lotsOfData* *value*)  {  *value*.param1 \*= 2;      return *value*;  }  /\* Hàm trả về cấu trúc lớn bằng con trỏ thay vì giá trị, tương đối nhanh \*/  void doubleParam1ByPtr(struct *lotsOfData* \**value*)  {  *value*->param1 \*= 2;  } |

## PHẦN 13.5 OBJECT- BASED PROGRAMMING USING STRUCTS (LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG SỬ DỤNG STRUCTS)

Cấu trúc có thể được sử dụng để triển khai mã theo cách lập trình hướng đối tượng. Một cấu trúc tương tự như một lớp, nhưng thiếu các hàm mà thông thường cũng là một phần của một lớp, chúng ta có thể thêm chúng vào dưới dạng biến thành viên con trỏ hàm. Để giữ nguyên ví dụ về tọa độ của chúng ta:

|  |
| --- |
| /\* coordinates.h \*/  typedef struct *coordinate\_s*  {      /\* Con trỏ tới các hàm phương thức \*/      void (\*setx)(*coordinate* \*this, int x);      void (\*sety)(*coordinate* \*this, int y);      void (\*print)(*coordinate* \*this);      /\* Dữ liệu \*/      int x;      int y;  } *coordinate*;  /\* Constructor \*/  *coordinate* \*coordinate\_create(void);  /\* Destructor \*/  void coordinate\_destroy(*coordinate* \**this*); |

Và sau đó tạo tệp thực hiện C:

|  |
| --- |
| /\* coordinates.c \*/  #include "coordinates.h"  #include <stdio.h>  #include <stdlib.h>  /\* Constructor \*/  coordinate \*coordinate\_create(void)  {      coordinate \*c = malloc(sizeof(\*c));      if (c != 0)      {          c->setx = &coordinate\_setx;          c->sety = &coordinate\_sety;          c->print = &coordinate\_print;          c->x = 0;          c->y = 0;      }      return c;  }  /\* Destructor \*/  void coordinate\_destroy(coordinate \**this*)  {      if (*this* != NULL)      {          free(*this*);      }  }  /\* Phương thức \*/  static void coordinate\_setx(coordinate \**this*, int *x*)  {      if (*this* != NULL)      {  *this*->x = *x*;      }  }  static void coordinate\_sety(coordinate \**this*, int *y*)  {      if (*this* != NULL)      {  *this*->y = *y*;      }  }  static void coordinate\_print(coordinate \**this*)  {      if (*this* != NULL)      {          printf("Coordinate: (%i, %i)\n", *this*->x, *this*->y);      }      else      {          printf("NULL pointer exception!\n");      }  } |

Một ví dụ về việc sử dụng lớp tọa độ của chúng ta sẽ là:

|  |
| --- |
| /\* main.c \*/  #include "coordinates.h"  #include <stddef.h>  int main(void)  {      /\* Tạo và khởi tạo các con trỏ tới các đối tượng tọa độ \*/      coordinate \*c1 = coordinate\_create();      coordinate \*c2 = coordinate\_create();      /\* Bây giờ chúng ta có thể sử dụng các đối tượng của chúng ta bằng cách sử dụng các phương thức và truyền đối tượng như là tham số \*/      c1->setx(c1, 1);      c1->sety(c1, 2);      c2->setx(c2, 3);      c2->sety(c2, 4);      c1->print(c1);      c2->print(c2);      /\* Sau khi sử dụng các đối tượng của chúng ta, chúng ta hủy chúng bằng cách sử dụng hàm "hủy" của chúng ta \*/      coordinate\_destroy(c1);      c1 = NULL;      coordinate\_destroy(c2);      c2 = NULL;      return 0;  } |

## PHẦN 13.6 SIMPLE DATA STRUCTURES (CÁC CẤU TRÚC DỮ LIỆU ĐƠN GIẢN)

Cấu trúc dữ liệu kiểu struct là cách hữu ích để đóng gói các dữ liệu liên quan và cho phép chúng hoạt động như một biến duy nhất.

Khai báo một cấu trúc đơn giản chứa hai trường kiểu int:

|  |
| --- |
| struct *point*  {      int x;      int y;  }; |

x và y được gọi là các thành viên (hoặc trường) của cấu trúc point.

Định nghĩa và sử dụng các cấu trúc:

|  |
| --- |
| struct *point* p; // khai báo p là một cấu trúc point  p.x = 5;        // gán các biến thành viên của p  p.y = 3; |

Cấu trúc có thể được khởi tạo tại định nghĩa. Đoạn mã trên tương đương với:

|  |
| --- |
| struct *point* p = {5, 3}; |

Cấu trúc cũng có thể được khởi tạo bằng cách sử dụng các trình khởi tạo được chỉ định (designated initializers).

Truy cập các trường cũng được thực hiện bằng cách sử dụng toán tử chấm (.)

|  |
| --- |
| printf("point is (x = %d, y = %d)", p.x, p.y); |

# STANDARD MATH (TOÁN HỌC TIÊU CHUẨN)

## PHẦN 14.1 CÁC HÀM MŨ – POW(), POWF(), POWL()

Đoạn mã ví dụ sau tính tổng của dãy 1 + 4(3 + 3^2 + 3^3 + 3^4 + ... + 3^N) sử dụng các hàm gia đình pow() trong thư viện toán học tiêu chuẩn.

|  |
| --- |
| #include <stdio.h>  #include <math.h>  #include <errno.h>  #include <fenv.h>  int main()  {      double pwr, sum = 0;      int i, n;      printf("\n1 + 4(3 + 3^2 + 3^3 + 3^4 + ... + 3^N) = ?\nEnter N:");      scanf("%d", &n);      if (n <= 0)      {          printf("Invalid power N=%d", n);          return -1;      }      for (i = 0; i < n + 1; i++)      {          errno = 0;          feclearexcept(FE\_ALL\_EXCEPT);          pwr = powl(3, i);          if (fetestexcept(FE\_INVALID | FE\_DIVBYZERO | FE\_OVERFLOW | FE\_UNDERFLOW))          {              perror("Math Error");          }          sum += i ? pwr : 0;          printf("N = %d\tS = %g\n", i, 1 + 4 \* sum);      }      return 0;  } |

Kết quả:

|  |
| --- |
| 1 + 4(3 + 3^2 + 3^3 + 3^4 + ... + 3^N) = ?  Enter N:10  N = 0 S = 1  N = 1 S = 13  N = 2 S = 49  N = 3 S = 157  N = 4 S = 481  N = 5 S = 1453  N = 6 S = 4369  N = 7 S = 13117  N = 8 S = 39361  N = 9 S = 118093  N = 10 S = 354289 |

## PHẦN 14.2 DOUBLE PRECISION FLOATING- POINT REMAINDER FMOD() (PHẦN DƯ SỐ THỰC VỚI ĐỘ CHÍNH XÁC KÉP FMOD())

Hàm fmod() trả về phần dư số thực độ chính xác kép của phép chia x/y. Giá trị trả về có cùng dấu với x.

|  |
| --- |
| #include <math.h> /\* for fmod() \*/  #include <stdio.h> /\* for printf() \*/  int main(void)  {      double x = 10.0;      double y = 5.1;      double modulus = fmod(x, y);      printf("%lf\n", modulus); /\* f tương đương với lf \*/      return 0;  } |

Kết quả:

|  |
| --- |
| 4.900000 |

**Lưu ý:** Sử dụng hàm này cẩn thận, vì nó có thể trả về các giá trị không mong đợi do hoạt động của các giá trị số thực.

|  |
| --- |
| #include <math.h>  #include <stdio.h>  int main(void)  {      printf("%f\n", fmod(1, 0.1));      printf("%19.17f\n", fmod(1, 0.1));      return 0;  } |

Kết quả:

|  |
| --- |
| 0.100000  0.09999999999999995 |

## PHẦN 14.3 SINGLE PRECISION AND LONG DOUBLE PRECISION FLOATING- POINT REMAINDER: FMODF(), FMODL()

Hàm fmodf() và fmodl() trả về phần dư số thực độ chính xác đơn và đôi của phép chia x/y. Giá trị trả về có cùng dấu với x.

Số thực độ chính xác đơn (Single Precision):

|  |
| --- |
| #include <math.h> /\* for fmodf() \*/  #include <stdio.h> /\* for printf() \*/  int main(void)  {      float x = 10.0;      float y = 5.1;      float modulus = fmodf(x, y);      printf("%f\n", modulus); /\* Lf sẽ hoạt động tốt khi modulus được thăng cấp thành kiểu double. \*/      return 0;  } |

Kết quả:

|  |
| --- |
| 4.900000 |

Số thực độ chính xác kép (Double Precision):

|  |
| --- |
| #include <math.h> /\* for fmodl() \*/  #include <stdio.h> /\* for printf() \*/  int main(void)  {      long double x = 10.0;      long double y = 5.1;      long double modulus = fmodl(x, y);      printf("%Lf\n", modulus); /\* Lf dùng cho số thực độ chính xác kép (long double). \*/      return 0;  } |

Kết quả:

|  |
| --- |
| 4.90000 |

# CÁC LỆNH LẶP/ VÒNG LẶP: FOR, WHILE, DO WHILE

## PHẦN 15.1 VÒNG LẶP FOR

Để thực hiện một khối mã lệnh nhiều lần, ta sử dụng vòng lặp (loop). Vòng lặp **for** được sử dụng khi một khối mã lệnh cần được thực thi một số lần cụ thể. Ví dụ, để điền vào một mảng có kích thước n với đầu vào từ người dùng, chúng ta cần thực hiện lệnh **scanf()** **n** lần.

|  |
| --- |
| #include <stddef.h> // for size\_t  int array[10]; // mảng gồm 10 số nguyên  for (size\_t i = 0; i < 10; i++) // i bắt đầu từ 0 và kết thúc ở 9  {  scanf("%d", &array[i]);  } |

Theo cách này, lệnh gọi scanf() được thực hiện n lần (10 lần trong ví dụ của chúng ta), nhưng được viết một lần duy nhất.

Ở đây, biến i là chỉ mục vòng lặp và nó nên được khai báo như trình bày. Kiểu size\_t (kiểu kích thước) nên được sử dụng cho mọi thứ đếm hoặc lặp qua các đối tượng dữ liệu.

Cách khai báo biến bên trong vòng lặp for chỉ có sẵn cho các trình biên dịch đã được cập nhật lên tiêu chuẩn C99. Nếu vì một lý do nào đó bạn vẫn phải sử dụng trình biên dịch cũ hơn, bạn có thể khai báo chỉ mục vòng lặp trước vòng lặp for:

|  |
| --- |
| #include <stddef.h> /\* for size\_t \*/  *size\_t* i;  int array[10]; /\* mảng gồm 10 số nguyên \*/  for (i = 0; i < 10; i++) /\* i bắt đầu từ 0 và kết thúc ở 9 \*/  {      scanf("%d", &array[i]);  } |

## PHẦN 15.2 LOOP UNROLLING AND DUFF’S DEVICE

Đôi khi, vòng lặp thông thường không thể được hoàn toàn chứa trong thân vòng lặp. Điều này là vì, vòng lặp cần được chuẩn bị bởi một số lệnh B. Sau đó, vòng lặp bắt đầu với một số lệnh A, sau đó tiếp tục với B trước khi lặp lại.

|  |
| --- |
| do\_B();  while (condition) {      do\_A();      do\_B();  } |

Để tránh vấn đề coppy/paste tiềm ẩn khi lặp lại B hai lần trong mã, **Duff's Device** có thể được áp dụng để bắt đầu vòng lặp từ giữa phần thân của while, bằng cách sử dụng một câu lệnh switch và cách thức chuyển hướng dưới đây.

|  |
| --- |
| switch (true) while (condition) {  case false: do\_A(); /\* FALL THROUGH \*/  default: do\_B(); /\* FALL THROUGH \*/  } |

Thực tế, Duff's Device được phát minh để thực hiện giải thuật gỡ vòng lặp. Giả sử bạn áp dụng một mặt nạ vào một khối bộ nhớ, trong đó n là một kiểu số nguyên có giá trị dương.

|  |
| --- |
| do {      \*ptr++ ^= mask;  } while (--n > 0); |

Nếu n luôn chia hết cho 4, bạn có thể dễ dàng gỡ vòng lặp này như sau:

|  |
| --- |
| do {      \*ptr++ ^= mask;      \*ptr++ ^= mask;      \*ptr++ ^= mask;      \*ptr++ ^= mask;  } while ((n -= 4) > 0); |

Nhưng, với Duff's Device, mã có thể tuân theo giải thuật gỡ vòng lặp này bằng cách nhảy vào vị trí đúng giữa vòng lặp nếu n không chia hết cho 4.

|  |
| --- |
| switch (n % 4) do {  case 0: \*ptr++ ^= mask; /\* FALL THROUGH \*/  case 3: \*ptr++ ^= mask; /\* FALL THROUGH \*/  case 2: \*ptr++ ^= mask; /\* FALL THROUGH \*/  case 1: \*ptr++ ^= mask; /\* FALL THROUGH \*/  } while ((n -= 4) > 0); |

Loại bỏ vòng lặp thủ công như vậy hiếm khi cần thiết với các trình biên dịch hiện đại, vì trình tối ưu hóa của trình biên dịch có thể thực hiện gỡ vòng lặp thay mặt người lập trình.

## PHẦN 15.3 VÒNG LẶP WHILE

Vòng lặp while được sử dụng để thực thi một đoạn mã khi điều kiện là đúng. Vòng lặp while được sử dụng khi một khối mã cần được thực thi một số lần có thể thay đổi. Ví dụ mã dưới đây nhận đầu vào từ người dùng, miễn là người dùng nhập các số khác không. Nếu người dùng nhập số 0, điều kiện của vòng lặp while không còn đúng nữa, vòng lặp sẽ kết thúc và tiếp tục thực thi các mã tiếp theo:

|  |
| --- |
| int num = 1;  while (num != 0)  {      scanf("%d", &num);  } |

## PHẦN 15.4 VÒNG LẶP DO – WHILE

Khác với vòng lặp for và while, vòng lặp do-while kiểm tra điều kiện sau khi thực thi phần thân của vòng lặp, điều này có nghĩa là khối mã trong do sẽ được thực thi ít nhất một lần và sau đó kiểm tra điều kiện của while ở cuối khối mã. Điều này đồng nghĩa với việc vòng lặp do-while luôn chạy ít nhất một lần.

Ví dụ vòng lặp do-while sau sẽ lấy các số từ người dùng cho đến khi tổng các giá trị này lớn hơn hoặc bằng 50:

|  |
| --- |
| int num, sum;  num = sum = 0;  do  {      scanf("%d", &num);      sum += num;  } while (sum < 50); |

Vòng lặp do-while có thể gặp ít hơn trong hầu hết các phong cách lập trình.

## PHẦN 15.5 CẤU TRÚC VÀ LUỒNG ĐIỀU KHIỂN TRONG VÒNG LẶP FOR

|  |
| --- |
| for([khai báo hoặc biểu thức 1]; [biểu thức 2]; [biểu thức 3]){      /\*Phần thân của vòng lặp\*/  } |

Trong một vòng lặp for, điều kiện lặp có ba biểu thức, tất cả đều là tùy chọn.

* Biểu thức đầu tiên, được gọi là khai báo hoặc biểu thức "declaration-or-expression", khởi tạo vòng lặp. Nó chỉ được thực thi một lần duy nhất ở đầu của vòng lặp.

Phiên bản ≥ C99:

Nó có thể là một khai báo và khởi tạo của biến vòng lặp, hoặc là một biểu thức tổng quát. Nếu nó là một khai báo, phạm vi của biến được khai báo được **giới hạn** bởi câu lệnh for.

Phiên bản < C99:

Các phiên bản C cũ chỉ cho phép sử dụng biểu thức ở đây, và khai báo biến vòng lặp phải được đặt trước vòng lặp for.

* Biểu thức thứ hai, được gọi là biểu thức 2"expression2", là điều kiện kiểm tra. Nó được thực thi lần đầu tiên sau khi khởi tạo. Nếu điều kiện **đúng**, thì điều khiển vào phần thân của vòng lặp. Nếu **không đúng,** nó chuyển đến bên ngoài phần thân của vòng lặp ở cuối vòng lặp. Sau đó, điều kiện này được kiểm tra sau mỗi thực thi của phần thân cũng như biểu thức cập nhật. Khi điều kiện đúng, điều khiển quay lại đầu phần thân của vòng lặp. Điều kiện này thường được dùng để kiểm tra số lần thực thi của phần thân vòng lặp. Đây là cách chính để thoát khỏi vòng lặp, cách khác là sử dụng các câu lệnh nhảy (jump statements).
* Biểu thức thứ ba, được gọi là biểu thức 3"expression3", là **câu lệnh cập nhật**. Nó được thực thi sau mỗi thực thi của phần thân của vòng lặp. Thường thì nó được sử dụng để tăng giá trị của biến lưu số lần thực thi của phần thân vòng lặp, và biến này được gọi là "**iterator**".

Mỗi lần thực thi của phần thân vòng lặp được gọi là một "**iteration**".

Ví dụ:

Phiên bản ≥ C99

|  |
| --- |
| for (int i = 0; i < 10; i++)  {      printf("%d", i);  } |

Kết quả

|  |
| --- |
| 0123456789 |

Trong ví dụ trên, trước tiên i = 0 được thực thi, khởi tạo i. Sau đó, điều kiện i < 10 được kiểm tra, nó đánh giá là đúng. Điều khiển vào phần thân của vòng lặp và giá trị của i được in ra. Sau đó, điều khiển chuyển đến i++, cập nhật giá trị của i từ 0 lên 1. Sau đó, điều kiện lại được kiểm tra và quá trình tiếp tục. Điều này tiếp tục cho đến khi giá trị của i trở thành 10. Sau đó, điều kiện i < 10 đánh giá sai, sau đó điều khiển di chuyển ra khỏi vòng lặp.

## PHẦN 15.6 VÒNG LẶP VÔ HẠN

Một vòng lặp được gọi là vòng lặp vô hạn nếu điều khiển vào nhưng không bao giờ rời khỏi phần thân của vòng lặp. Điều này xảy ra khi điều kiện kiểm tra của vòng lặp không bao giờ đánh giá là **sai**.

Ví dụ:

Phiên bản ≥ C99

|  |
| --- |
| for (int i = 0; i >= 0; )  {      /\* phần thân của vòng lặp, nơi i không bị thay đổi\*/  } |

Trong ví dụ trên, biến i, bộ đếm, được khởi tạo thành 0. Điều kiện kiểm tra ban đầu đúng. Tuy nhiên, i không được thay đổi ở bất kỳ nơi nào trong phần thân và biểu thức cập nhật không có gì. Do đó, i sẽ giữ nguyên là 0 và điều kiện kiểm tra sẽ không bao giờ đánh giá sai, dẫn đến một vòng lặp vô hạn.

Giả định rằng không có câu lệnh nhảy, một cách khác mà một vòng lặp vô hạn có thể được tạo ra là bằng cách giữ cho điều kiện luôn luôn đúng:

|  |
| --- |
| while (true)  {      /\* phần thân của vòng lặp \*/  } |

Trong vòng lặp for, câu lệnh điều kiện là tùy chọn. Trong trường hợp này, điều kiện luôn luôn đúng, dẫn đến một vòng lặp vô hạn.

|  |
| --- |
| for (;;)  {      /\* phần thân của vòng lặp \*/  } |

Tuy nhiên, trong một số trường hợp, điều kiện có thể được giữ đúng một cách có ý định, với mục đích thoát khỏi vòng lặp bằng một câu lệnh nhảy như **break**.

|  |
| --- |
| while (true)  {      /\* các câu lệnh \*/      if (condition)      {          /\* các câu lệnh khác \*/      break;      }  } |

# SELECTION STATEMENTS (CÂU LỆNH LỰA CHỌN)

## PHẦN 16.1 CÂU LỆNH IF()

Một trong những cách đơn giản nhất để điều khiển luồng chương trình là sử dụng câu lệnh if (). Có thể quyết định liệu một khối mã có được thực thi hay không dựa vào câu lệnh này.

Cú pháp cho câu lệnh if trong C có thể như sau:

|  |
| --- |
| if (điều\_kiện)  {      các\_câu\_lệnh; /\* được thực thi khi điều kiện là true \*/  } |

Ví dụ:

|  |
| --- |
| if (a > 1) {      puts("a lớn hơn 1");  } |

Ở đây, a > 1 là một điều kiện phải đánh giá là **true**(đúng) để thực thi các câu lệnh bên trong khối if. Trong ví dụ này, "a lớn hơn 1" chỉ được in ra màn hình nếu a > 1 là **true**(đúng).

Câu lệnh if có thể bỏ qua cặp dấu ngoặc nhọn { and } nếu chỉ có một câu lệnh trong khối. Ví dụ trên có thể viết lại thành:

|  |
| --- |
| if (a > 1)      puts("a lớn hơn 1"); |

Tuy nhiên, để thực thi nhiều câu lệnh trong khối thì phải sử dụng cặp dấu ngoặc nhọn { }.

Điều kiện cho câu lệnh if có thể bao gồm nhiều biểu thức. Câu lệnh if chỉ thực hiện hành động nếu kết quả cuối cùng của biểu thức là **true**(đúng).

Ví dụ

|  |
| --- |
| if ((a > 1) && (b > 1)) {      puts("a lớn hơn 1");      a++;  } |

sẽ chỉ thực hiện printf và a++ nếu cả a và b đều lớn hơn 1.

## PHẦN 16.2 NESTED IF()… ELSE VS IF ELSE LADDER ( CÂU LỆNH IF LỒNG NHAU VÀ CÂU LỆNH IF BẬC THANG)

Các câu lệnh **if()...else lồng nhau** tốn nhiều thời gian thực thi hơn (chúng chạy chậm hơn) so với một chuỗi **if()...else**, bởi vì các câu lệnh **if()...else** lồng nhau kiểm tra tất cả các điều kiện bên trong sau khi câu lệnh **if()** bên ngoài được thỏa mãn, trong khi chuỗi **if()..else** dừng kiểm tra điều kiện ngay khi bất kỳ điều kiện **if()** hoặc **else if()** nào được thỏa mãn.

**if()… else bậc thang**

|  |
| --- |
| #include<stdio.h>  int main(int *argc*, char \**argv*[])  {      int a, b, c;      printf("\nEnter Three numbers = ");      scanf("%d%d%d", &a, &b, &c);      if ((a < b) && (a < c))      {          printf("\na = %d is the smallest.", a);      }      else if ((b < a) && (b < c))      {          printf("\nb = %d is the smallest.", b);      }      else if ((c < a) && (c < b))      {          printf("\nc = %d is the smallest.", c);      }      else      {          printf("\nImprove your coding logic");      }      return 0;  } |

**if()… else lồng nhau**

|  |
| --- |
| #include <stdio.h>  int main(int *argc*, char \**argv*[])  {      int a, b, c;      printf("\nEnter Three numbers = ");      scanf("%d%d%d", &a, &b, &c);      if (a < b)      {          if (a < c)          {              printf("\na = %d is the smallest.", a);          }          else          {              printf("\nc = %d is the smallest.", c);          }      }      else      {          if(b < c)          {              printf("\nb = %d is the smallest.", b);          }          else          {          printf("\nc = %d is the smallest.", c);          }      }      return 0;  } |

## PHẦN 16.3 CÂU LỆNH SWITCH()

Câu lệnh **switch** được sử dụng khi bạn muốn chương trình thực hiện nhiều hành động khác nhau tùy theo giá trị của một biến kiểm tra cụ thể.

Ví dụ sử dụng câu lệnh **switch** như sau:

|  |
| --- |
| int a = 1;  switch (a) {  case 1:      puts("a is 1");      break;  case 2:      puts("a is 2");      break;  default:      puts("a is neither 1 nor 2");      break;  } |

Ví dụ này tương đương với việc sử dụng câu lệnh if()...else if()...else:

|  |
| --- |
| int a = 1;  if (a == 1) {      puts("a is 1");  } else if (a == 2) {      puts("a is 2");  } else {      puts("a is neither 1 nor 2");  } |

Nếu giá trị của a là 1 khi sử dụng câu lệnh switch, sẽ in ra "a is 1". Nếu giá trị của a là 2, sẽ in ra "a is 2". Nếu không phù hợp với bất kỳ lựa chọn nào của case, sẽ in ra "a is neither 1 nor 2".

Trong câu lệnh switch, các trường hợp **case n**: được sử dụng để mô tả vị trí mà luồng thực thi sẽ nhảy vào khi giá trị được truyền vào câu lệnh switch là n. Giá trị của n phải là hằng số tại thời điểm biên dịch và n có thể tồn tại nhiều nhất một lần trong một câu lệnh switch.

Trường hợp **default**: được sử dụng để mô tả khi giá trị không phù hợp với bất kỳ lựa chọn nào của case. Một thói quen tốt là luôn bao gồm trường hợp default trong mỗi câu lệnh switch để xử lý hành vi không mong muốn.

Một câu lệnh **break**; được yêu cầu để thoát khỏi khối **switch**.

**Lưu ý:** Nếu bạn vô tình quên thêm **break** sau mỗi trường hợp **case**, trình biên dịch sẽ giả định rằng bạn có ý định thực hiện "rơi vào" (**fall through**) và tất cả các trường hợp case tiếp theo, nếu có, sẽ được thực thi (trừ khi có break trong bất kỳ trường hợp case nào sau đó). Điều này có thể được sử dụng để triển khai Duff's Device. Tính năng này thường được coi là một khuyết điểm trong quy định ngôn ngữ C.

Dưới đây là một ví dụ cho thấy hiệu ứng khi không có **break;:**

|  |
| --- |
| int a = 1;  switch (a) {  case 1:  case 2:      puts("a is 1 or 2");  case 3:      puts("a is 1, 2 or 3");      break;  default:      puts("a is neither 1, 2 nor 3");      break;  } |

Khi giá trị của a là 1 hoặc 2, cả "a is 1 or 2" và "a is 1, 2 or 3" đều được in ra. Khi a là 3, chỉ có "a is 1, 2 or 3" được in ra. Nếu không phù hợp với bất kỳ lựa chọn nào của case, sẽ in ra "a is neither 1, 2 nor 3".

Lưu ý rằng *case* **default** không nhất thiết phải có, đặc biệt khi tập hợp các giá trị bạn nhận được trong câu lệnh switch đã được hoàn thành và biết trước tại thời điểm biên dịch.

Ví dụ tốt nhất là sử dụng **switch** trên một **enum**.

|  |
| --- |
| enum msg\_type { ACK, PING, ERROR };  void f(enum *msg\_type* *t*)  {      switch (*t*) {      case ACK:          // không làm gì cả      break;      case PING:          // thực hiện một số công việc      break;      case ERROR:          // thực hiện công việc khác      break;      }  } |

Có nhiều lợi ích khi làm như vậy:

* Hầu hết các trình biên dịch sẽ báo lỗi nếu bạn không xử lý một giá trị (điều này sẽ không được báo lỗi nếu có trường hợp default).
* Vì cùng một lý do, nếu bạn thêm một giá trị mới vào enum, bạn sẽ nhận được thông báo về tất cả các vị trí mà bạn đã quên xử lý giá trị mới (nếu có trường hợp default, bạn sẽ phải tìm thủ công trong mã của bạn để tìm các trường hợp đó).
* Người đọc không cần phải xem xét "được ẩn bởi trường hợp default:", liệu có các giá trị enum khác hay không hoặc liệu nó là một sự bảo vệ "just in case". Nếu có các giá trị enum khác, liệu người viết mã có ý dùng trường hợp default cho chúng hay không, hay có lỗi xuất hiện khi người viết thêm giá trị?
* Xử lý từng giá trị enum khiến mã trở nên dễ hiểu vì bạn không thể che giấu sau một ký tự đại diện (wild card), bạn phải xử lý từng giá trị cụ thể.

Tuy nhiên, bạn không thể ngăn người khác viết mã xấu như sau:

|  |
| --- |
| enum *msg\_type* t = (enum *msg\_type*)666; // Tôi là evil |

Do đó, nếu bạn thực sự cần, bạn có thể thêm một kiểm tra bổ sung trước câu lệnh switch để phát hiện điều này.

|  |
| --- |
| void f(enum *msg\_type* *t*)  {      if (!is\_msg\_type\_valid(*t*)) {          // Xử lý lỗi không thường xảy ra này      }      switch(*t*) {          // Cùng mã như trước đó      }  } |

## PHẦN 16.4 CÂU LỆNH IF()… ELSE VÀ CÚ PHÁP

Trong khi lệnh if thực hiện một hành động chỉ khi điều kiện của nó **đúng**, if / else cho phép bạn xác định các hành động khác nhau khi điều kiện **đúng** và khi điều kiện **sai.**

Ví dụ:

|  |
| --- |
| if (a > 1)      puts("a is larger than 1");  else      puts("a is not larger than 1"); |

Tương tự như câu lệnh if, khi khối lệnh trong if hoặc else chỉ gồm một câu lệnh, dấu ngoặc nhọn có thể được bỏ qua (nhưng không nên làm điều này vì có thể dễ dàng gây ra các vấn đề không cần thiết). Tuy nhiên, nếu có nhiều hơn một câu lệnh trong khối lệnh if hoặc else, thì phải sử dụng dấu ngoặc nhọn cho khối đó.

|  |
| --- |
| if (a > 1)  {      puts("a is larger than 1");      a--;  }  else  {      puts("a is not larger than 1");      a++;  } |

## PHẦN 16.5 NỐI HAI HOẶC NHIỀU LỆNH IF()… ELSE

Trong khi lệnh if () ... else chỉ cho phép xác định một (mặc định) hành vi xảy ra khi điều kiện trong lệnh if không được đáp ứng, việc nối hai hoặc nhiều lệnh if () ... else cho phép xác định một số hành vi khác trước khi điều kiện chuyển đến nhánh else cuối cùng, hành vi này được xem như "default", nếu có.

Ví dụ:

|  |
| --- |
| int a = ... /\* khởi tạo với một giá trị nào đó. \*/  if (a >= 1)  {      printf("a is greater than or equals 1.\n");  }  else if (a == 0) // ta đã biết rằng a nhỏ hơn 1  {      printf("a equals 0.\n");  }  else /\* a nhỏ hơn 1 và không bằng 0, do đó: \*/  {      printf("a is negative.\n");  } |

# INITIALIZATION (KHỞI TẠO)

## PHẦN 17.1 KHỞI TẠO BIẾN TRONG C

Trong trường hợp không có khởi tạo rõ ràng, các biến bên ngoài (biến toàn cục) và biến tĩnh(static) được đảm bảo được khởi tạo thành giá trị **không**; các biến tự động (bao gồm cả biến đăng ký) có giá trị ban đầu không xác định (tức là giá trị rác).

Các biến nguyên thủy (scalar) có thể được khởi tạo khi được định nghĩa bằng cách đặt dấu bằng và biểu thức sau tên biến:

|  |
| --- |
| int x = 1;  char squota = '\'';  long day = 1000L \* 60L \* 60L \* 24L; /\* milliseconds/day \*/ |

Đối với biến bên ngoài và tĩnh(static) bộ khởi tạo phải là một biểu thức hằng số; việc khởi tạo được thực hiện một lần, trước khi chương trình bắt đầu thực thi.

Đối với biến tự động và biến đăng ký, bộ khởi tạo không bị giới hạn chỉ là hằng số: nó có thể là bất kỳ biểu thức nào liên quan đến các giá trị đã được xác định trước, kể cả các lời gọi hàm.

Ví dụ xem đoạn mã dưới đây

|  |
| --- |
| int binsearch(int *x*, int *v*[], int *n*)  {      int low = 0;      int high = *n* - 1;      int mid;      ...  } |

Thay vì

|  |
| --- |
| int low, high, mid;  low = 0;  high = n - 1; |

Trong thực tế, việc khởi tạo các biến tự động chỉ là cách viết ngắn gọn cho các câu lệnh gán. Loại nào nên được ưu tiên phụ thuộc phần lớn vào sở thích. Chúng ta thường sử dụng các phép gán rõ ràng, vì việc khởi tạo trong các khai báo khó nhìn thấy hơn và xa so với điểm sử dụng. Tuy nhiên, biến chỉ nên được khai báo khi chúng được sử dụng bất cứ khi nào có thể.

### 17.1.1 Khởi tạo một mảng

Một mảng có thể được khởi tạo bằng cách theo sau khai báo của nó bằng một danh sách các giá trị khởi tạo nằm trong dấu ngoặc nhọn và cách nhau bằng dấu phẩy.

Ví dụ, để khởi tạo một mảng days với số ngày trong mỗi tháng:

|  |
| --- |
| int days\_of\_month[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }; |

(Lưu ý rằng tháng 1 được mã hóa là tháng số 0 trong cấu trúc này.)

Khi kích thước của mảng bị bỏ qua, trình biên dịch sẽ tính độ dài bằng cách đếm các giá trị khởi tạo, trong trường hợp này có 12 giá trị.

Nếu có ít giá trị khởi tạo hơn số lượng chỉ định, những giá trị còn lại sẽ là giá trị không(0) cho tất cả các loại biến. Nếu có quá nhiều bộ khởi tạo, đó là lỗi. Không có cách tiêu chuẩn nào để chỉ định lặp lại bộ khởi tạo - nhưng GCC có một phần mở rộng để làm điều đó.

Phiên bản < C99

Trong C89/C90 hoặc các phiên bản C trước đó, không có cách nào để khởi tạo một phần tử ở giữa một mảng mà không cung cấp tất cả các giá trị trước đó.

Phiên bản ≥ C99

Với C99 và các phiên bản sau đó, các khởi tạo đặc chỉ định cho phép bạn khởi tạo các phần tử tùy ý của một mảng, để lại các giá trị chưa được khởi tạo như số 0.

### 17.1.2 Khởi tạo mảng kí tự

Trong ngôn ngữ lập trình C, khởi tạo mảng ký tự có một cách viết ngắn gọn đặc biệt. Chúng ta có thể sử dụng một chuỗi (string) để khởi tạo mảng ký tự thay vì sử dụng cặp dấu ngoặc nhọn và dấu phẩy.

Ví dụ:

|  |
| --- |
| char chr\_array[] = "hello"; |

Đoạn mã trên là một cách viết ngắn gọn tương đương với:

|  |
| --- |
| char chr\_array[] = { 'h', 'e', 'l', 'l', 'o', '\0' }; |

Trong trường hợp này, mảng chr\_array có kích thước là 6 (gồm 5 ký tự và ký tự null '\0' để đánh dấu kết thúc chuỗi).

1. Biến không được khởi tạo (uninitialized variable) trong C sẽ có giá trị không xác định (rác). Giá trị của biến không được đảm bảo và có thể chứa các giá trị ngẫu nhiên từ bộ nhớ trước đó. Việc sử dụng biến không được khởi tạo có thể dẫn đến hành vi không xác định và kết quả không đáng tin cậy trong chương trình.
2. Một biểu thức hằng số (constant expression) trong C là một biểu thức có thể được đánh giá ngay tại thời điểm biên dịch (compile-time). Một biến có từ khóa "const" chỉ là biến chỉ đọc (read-only) và không phải là một biểu thức hằng số. Do đó, các biến có từ khóa "const" không thể được sử dụng để xác định kích thước của mảng trong C. Ví dụ sau không hợp lệ:

|  |
| --- |
| const int SIZE = 10;  int global\_arr[SIZE]; |

Và cũng không thể sử dụng biến "const" để khởi tạo mảng, như sau:

|  |
| --- |
| const int SIZE = 10;  int global\_var = SIZE; |

Điều này không hợp lệ trong C vì kích thước của mảng phải là một hằng số đã biết tại thời điểm biên dịch, không thể là một biến có từ khóa "const" có giá trị không biết trước.

## PHẦN 17.2 SỬ DỤNG DESIGNATED INITIALIZERS

Trong phiên bản C99 trở lên, ngôn ngữ C hỗ trợ (designated initializers). Điều này cho phép bạn chỉ định các phần tử của một array, structure hoặc union sẽ được khởi tạo bằng các giá trị theo sau.

### 17.2.1 Designated initializers cho các phần tử mảng

Với kiểu dữ liệu đơn giản như int:

|  |
| --- |
| int array[] = { [4] = 29, [5] = 31, [17] = 101, [18] = 103, [19] = 107, [20] = 109 }; |

Phần tử nằm trong dấu ngoặc vuông và là một biểu thức nguyên hằng số, xác định phần tử của mảng sẽ được khởi tạo bằng giá trị của biểu thức sau dấu bằng. Các phần tử không được chỉ định sẽ được khởi tạo mặc định là 0. Ví dụ này chỉ ra việc “designated initializers” theo thứ tự; không cần phải theo thứ tự. Ví dụ cho thấy có một số khoảng trống; điều đó là hợp lệ. Ví dụ không hiển thị hai khởi tạo khác nhau cho cùng một phần tử; điều này cũng được phép.

Trong ví dụ này, kích thước của mảng không được xác định rõ ràng, do đó chỉ số tối đa được chỉ định trong các khởi tạo dùng định danh xác định kích thước của mảng - trong ví dụ này là 21 phần tử. Nếu kích thước được xác định rõ ràng, việc khởi tạo một phần tử vượt quá kích thước của mảng sẽ là lỗi, như thường lệ.

### 17.2.2 Designated initializers cho structures

Bạn có thể chỉ định các thành phần của một cấu trúc được khởi tạo bằng cách sử dụng cú pháp ***.element***:

|  |
| --- |
| struct *Date*  {      int year;      int month;      int day;  };  struct *Date* us\_independence\_day = { .day = 4, .month = 7, .year = 1776 }; |

Nếu các thành phần không được liệt kê, chúng sẽ được khởi tạo mặc định (bằng 0).

### 17.2.3 Designated initializer cho unions

Bạn có thể chỉ định phần tử nào của union được khởi tạo bằng Designated initializer.

Trước tiêu chuẩn C, không có cách nào để khởi tạo một union. Tiêu chuẩn C89/C90 cho phép bạn khởi tạo thành viên đầu tiên của union — vì vậy việc lựa chọn thành viên nào được liệt kê đầu tiên là vấn đề quan trọng.

|  |
| --- |
| struct *discriminated\_union*  {      enum { DU\_INT, DU\_DOUBLE } discriminant;      union      {          int du\_int;          double du\_double;      } du;  };  struct *discriminated\_union* du1 = { .discriminant = DU\_INT, .du = { .du\_int = 1 } };  struct *discriminated\_union* du2 = { .discriminant = DU\_DOUBLE, .du = { .du\_double = 3.14159 } }; |

Lưu ý rằng C11 cho phép bạn sử dụng các thành viên union ẩn danh bên trong một structure, do đó bạn không cần tên du trong ví dụ trước:

|  |
| --- |
| struct *discriminated\_union*  {      enum { DU\_INT, DU\_DOUBLE } discriminant;      union      {          int du\_int;          double du\_double;      };  };  struct *discriminated\_union* du1 = { .discriminant = DU\_INT, .du\_int = 1 };  struct *discriminated\_union* du2 = { .discriminant = DU\_DOUBLE, .du\_double = 3.14159 }; |

### 17.2.4 Designated initializers cho arrays của structures, etc

Các cấu trúc này có thể được kết hợp cho các mảng cấu trúc chứa các phần tử là mảng, v.v. Việc sử dụng các bộ dấu ngoặc nhọn đầy đủ đảm bảo rằng ký hiệu là rõ ràng.

|  |
| --- |
| typedef struct *Date* {      int year;      int month;      int day;  } *Date*;  struct *date\_range* {  *Date* dr\_from;  *Date* dr\_to;      char dr\_what[80];  };  struct *date\_range* ranges[] = {      [3] = { .dr\_from = { .year = 1066, .month = 10, .day = 14 },              .dr\_to = { .year = 1066, .month = 12, .day = 25 },              .dr\_what = "Chiến tranh Hastings đến Lễ đăng quang của William the Conqueror"            },      [2] = { .dr\_from = { .month = 7, .day = 4, .year = 1776 },              .dr\_to = { .month = 5, .day = 14, .year = 1787 },              .dr\_what = "Tuyên ngôn Độc lập Mỹ đến Hội nghị Hiến pháp"            }  }; |

### 17.2.5 Specifying ranges in array initializers (Chỉ định phạm vi trong bộ khởi tạo mảng)

GCC cung cấp một phần mở rộng cho phép bạn chỉ định một dải các phần tử trong mảng sẽ được khởi tạo bằng cùng một giá trị. Điều này giúp viết mã nguồn ngắn gọn hơn khi bạn muốn khởi tạo một số phần tử liên tiếp trong mảng.

|  |
| --- |
| int array[] = { [3 ... 7] = 29, [19] = 107 }; |

các dấu ba chấm (...) cần phải được tách rời với các số bằng khoảng trống hoặc các ký tự khác, nếu không chúng có thể bị hiểu là một phần của một số thực (floating-point number) do "quy tắc ăn nhiều nhất" (maximal munch rule).

## PHẦN 17.3 INITIALIZING STRUCTURES AND ARRAYS OF STRUCTURES (KHỞI TẠO CẤU TRÚC VÀ MẢNG CẤU TRÚC)

Cấu trúc và mảng cấu trúc có thể được khởi tạo bởi một loạt các giá trị được đặt trong dấu ngoặc nhọn, một giá trị cho mỗi thành viên của cấu trúc.

|  |
| --- |
| struct *Date*  {      int year;      int month;      int day;  };  struct *Date* us\_independence\_day = { 1776, 7, 4 };  struct *Date* uk\_battles[] =  {      { 1066, 10, 14 }, // Battle of Hastings      { 1815, 6, 18 }, // Battle of Waterloo      { 1805, 10, 21 }, // Battle of Trafalgar  }; |

Chú ý rằng việc khởi tạo mảng có thể được viết mà không cần dấu ngoặc nhọn bên trong, và trong quá khứ (trước năm 1990, chẳng hạn) thường sẽ được viết mà không có chúng:

|  |
| --- |
| struct *Date* uk\_battles[] =  {      1066, 10, 14, // Battle of Hastings      1815, 6, 18, // Battle of Waterloo      1805, 10, 21, // Battle of Trafalgar  }; |

Mặc dù cách viết này vẫn hoạt động, nhưng không phải là phong cách hiện đại tốt - bạn không nên cố gắng sử dụng cú pháp này trong mã mới và nên sửa các cảnh báo của trình biên dịch nó thường thông báo.

Hãy xem thêm về designated initializers.

# KHAI BÁO VÀ ĐỊNH NGHĨA

## PHẦN 18.1 HIỂU VỀ KHAI BÁO VÀ ĐỊNH NGHĨA

Một khai báo giới thiệu một định danh và mô tả kiểu dữ liệu của nó, cho dù là kiểu, đối tượng hoặc hàm. Một khai báo là điều mà trình biên dịch cần để chấp nhận các tham chiếu đến định danh đó. Dưới đây là các khai báo:

|  |
| --- |
| extern int bar;  extern int g(int, int);  double f(int, double); /\* từ khóa extern có thể bỏ qua cho khai báo hàm \*/  double h1(); /\* khai báo hàm không có nguyên mẫu \*/  double h2(); /\* tương tự cho khai báo hàm \*/ |

Một định nghĩa thực sự khởi tạo/hiện thực hóa định danh này. Đây là điều mà trình liên kết cần để liên kết các tham chiếu đến các thực thể đó. Dưới đây là các định nghĩa tương ứng với các khai báo ở trên:

|  |
| --- |
| int bar;  int g(int *lhs*, int *rhs*) { return *lhs* \* *rhs*; }  double f(int *i*, double *d*) { return *i* + *d*; }  double h1(int *a*, int *b*) { return -1.5; }  double h2() {} /\* nguyên mẫu được ngụ ý trong định nghĩa, tương tự double h2(void) \*/ |

Một định nghĩa có thể được sử dụng thay thế cho một khai báo.

Tuy nhiên, nó phải được định nghĩa đúng một lần. Nếu bạn quên định nghĩa một tên mà đã được khai báo và được tham chiếu ở nơi khác, trình liên kết sẽ không biết liên kết các tham chiếu đó với gì và báo lỗi về các ký hiệu bị thiếu. Ngược lại, nếu bạn định nghĩa một tên nhiều hơn một lần, trình liên kết không biết liên kết các tham chiếu với phiên bản nào và báo lỗi về các ký hiệu bị trùng lặp.

**Ngoại lệ:**

|  |
| --- |
| extern int i = 0; /\* định nghĩa 'i' với giá trị khởi tạo là 0 \*/  extern int j;     /\* chỉ khai báo 'j', không định nghĩa \*/ |

Trường hợp này có thể giải thích bằng các khái niệm "Strong symbols vs Weak symbols" (từ góc nhìn của trình liên kết). Vui lòng xem tại đây (Slide 22) để biết thêm giải thích.

|  |
| --- |
| /\* All are definitions. \*/  struct *S* { int a; int b; }; /\* defines S \*/  struct *X* { /\* defines X \*/      int x; /\* defines non-static data member x \*/  };  struct *X* anX; /\* defines anX \*/ |

# COMMAND- LINE ARGUMENTS (ĐỐI SỐ DÒNG LỆNH)

|  |  |
| --- | --- |
| Tham số | Chi tiết |
| argc | argument count(số lượng đối số) - được khởi tạo thành số lượng đối số được phân tách bằng dấu cách được cung cấp cho chương trình từ dòng lệnh cũng như chính tên chương trình. |
| argv | argument vector (vectơ đối số) - được khởi tạo thành một mảng các con trỏ ký tự (chuỗi) chứa các đối số (và tên chương trình) đã được đưa ra trên dòng lệnh. |

## PHẦN 19.1 IN RA CÁC ĐỐI SỐ CỦA CHƯƠNG TRÌNH VÀ CHUYỂN ĐỔI THÀNH CÁC GIÁ TRỊ SỐ NGUYÊN

Đoạn mã dưới đây sẽ in ra các đối số của chương trình và cố gắng chuyển đổi mỗi đối số thành một số (kiểu long):

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  #include <errno.h>  #include <limits.h>  int main(int *argc*, char\* *argv*[]) {      for (int i = 1; i < *argc*; i++) {          printf("Tham so %d la: %s\n", i, *argv*[i]);          errno = 0;          char \*p;          long argument\_numValue = strtol(*argv*[i], &p, 10);          if (p == *argv*[i]) {              fprintf(stderr, "Tham so %d khong phai la mot so.\n", i);          }          else if ((argument\_numValue == LONG\_MIN || argument\_numValue == LONG\_MAX) && errno == ERANGE) {              fprintf(stderr, "Tham so %d vuot qua pham vi.\n", i);          }          else {              printf("Tham so %d la mot so, va gia tri la: %ld\n", i, argument\_numValue);          }      }      return 0;  } |

Tham khảo:

* strtol() trả về giá trị không chính xác
* Cách sử dụng chính xác của strtol

## PHẦN 19.2 IN RA CÁC ĐỐI SỐ CỦA DÒNG LỆNH

Sau khi nhận được các đối số, bạn có thể in chúng như sau:

|  |
| --- |
| int main(int *argc*, char \*\**argv*)  {      for (int i = 1; i < *argc*; i++)      {          printf("Doi so %d: [%s]\n", i, *argv*[i]);      }  } |

**Ghi chú:**

1. Tham số argv có thể được định nghĩa là char \*argv[].
2. argv[0] có thể chứa tên chương trình chính (phụ thuộc vào cách chương trình được thực thi). Tham số dòng lệnh đầu tiên là argv[1], và đây là lý do tại sao biến vòng lặp i được khởi tạo bằng 1.
3. Trong câu lệnh in, bạn có thể sử dụng \*(argv + i) thay vì argv[i] - nó trả về cùng một giá trị, nhưng dài dòng hơn.
4. Dấu ngoặc vuông xung quanh giá trị tham số giúp xác định điểm bắt đầu và kết thúc. Điều này rất hữu ích nếu có các dấu cách cuối, ký tự mới, ký tự nằm ngang, hoặc các ký tự lạ trong tham số. Một biến thể của chương trình này là một công cụ hữu ích để gỡ lỗi các shell script khi bạn cần hiểu thực sự tham số dòng lệnh chứa gì (mặc dù có một số lựa chọn đơn giản trong shell gần như tương tự).

## PHẦN 19.3 SỬ DỤNG CÔNG CỤ GETOPT CỦA GNU

Các tùy chọn dòng lệnh cho các ứng dụng không được ngôn ngữ C xử lý khác với các đối số dòng lệnh. Chúng chỉ là các đối số, trong môi trường Linux hoặc Unix, theo truyền thống bắt đầu bằng dấu gạch ngang (-).

Với glibc trong môi trường Linux hoặc Unix, bạn có thể sử dụng các công cụ getopt để dễ dàng xác định, xác thực và phân tích các **command line** từ phần còn lại của các đối số của bạn.

Các công cụ này yêu cầu các tùy chọn của bạn được định dạng theo Tiêu chuẩn mã hóa GNU, đây là phần mở rộng của những gì POSIX chỉ định cho định dạng của các **command line**.

Ví dụ dưới đây minh họa việc xử lý các tùy **command line** với các công cụ getopt của GNU.

|  |
| --- |
|  |

Nó có thể được biên dịch với gcc:

|  |
| --- |
| gcc example.c -o example |

Chương trình này hỗ trợ ba tùy chọn dòng lệnh (--help, --file và --msg). Tất cả đều có một "dạng rút gọn" (-h, -f và -m).

Tùy chọn "file" và "msg" đều có thể nhận đối số. Nếu bạn chỉ định tùy chọn "msg", đối số của nó là bắt buộc.

Các đối số cho các tùy chọn được định dạng như sau:

* --option=value (cho các tùy chọn dạng dài)
* -o value hoặc -o "value" (cho các tùy chọn dạng ngắn)