LẬP TRÌNH C

**MỤC LỤC**

# XỬ LÝ TÍN HIỆU

|  |  |
| --- | --- |
| Tham số | Chi tiết |
| sig | Tín hiệu để đặt bộ xử lý tín hiệu thành một trong SIGABRT, SIGFPE, SIGILL, SIGTERM, SIGINT hoặc SIGSEGV một số giá trị được xác định triển khai |
| func | Trình xử lý tín hiệu, là một trong hai cách sau: SIG\_DFL, đối với trình xử lý mặc định, SIG\_IGN để bỏ qua tín hiệu hoặc một con trỏ hàm có chữ ký void foo(int sig);. |

## Xử lý tín hiệu với “signal()”

Các tín hiệu số có thể đồng bộ (như SIGSEGV– lỗi phân đoạn) khi chúng được kích hoạt bởi sự cố của chính chương trình hoặc không đồng bộ (như SIGINT- chú ý tương tác) khi chúng được bắt đầu từ bên ngoài chương trình, ví dụ: bằng cách nhấn phím như Ctrl-C.

Chức năng này signal() là một phần của tiêu chuẩn ISO C và có thể được sử dụng để gán chức năng xử lý một tín hiệu cụ thể

|  |
| --- |
| #include <stdio.h> /\* printf() \*/  #include <stdlib.h> /\* abort() \*/  #include <signal.h> /\* signal() \*/  void handler\_nonportable(int sig) {  /\* undefined behavior, maybe fine on specific platform \*/  printf("Catched: %d\n", sig);    /\* abort is safe to call \*/  abort();  }  sig\_atomic\_t volatile finished = 0;  void handler(int sig) {  switch (sig) {  /\* hardware interrupts should not return \*/  case SIGSEGV:  case SIGFPE:  case SIGILL:  C11  /\* quick\_exit is safe to call \*/  quick\_exit(EXIT\_FAILURE);  C11  /\* use \_Exit in pre-C11 \*/  \_Exit(EXIT\_FAILURE);  default:  /\* Reset the signal to the default handler,  so we will not be called again if things go  wrong on return. \*/  signal(sig, SIG\_DFL);  /\* let everybody know that we are finished \*/  finished = sig;  return;  }  }  int main(int argc, char\* argv[]) {  /\* Catch the SIGSEGV signal, raised on segmentation faults (i.e NULL ptr access \*/  if (signal(SIGSEGV, &handler) == SIG\_ERR) {  perror("could not establish handler for SIGSEGV");  return EXIT\_FAILURE;  }  /\* Catch the SIGTERM signal, termination request \*/  if (signal(SIGTERM, &handler) == SIG\_ERR) {  perror("could not establish handler for SIGTERM");  return EXIT\_FAILURE;  }  /\* Ignore the SIGINT signal, by setting the handler to `SIG\_IGN`. \*/  signal(SIGINT, SIG\_IGN);  /\* Do something that takes some time here, and leaves  the time to terminate the program from the keyboard. \*/  /\* Then: \*/  if (finished) {  fprintf(stderr, "we have been terminated by signal %d\n", (int)finished);  return EXIT\_FAILURE;  }  /\* Try to force a segmentation fault, and raise a SIGSEGV \*/  {  char\* ptr = 0;  \*ptr = 0;  }  /\* This should never be executed \*/  return EXIT\_SUCCESS;  } |

Việc sử dụng signal()áp đặt những giới hạn quan trọng mà bạn được phép thực hiện bên trong bộ xử lý tín hiệu, hãy xem phần nhận xét để biết thêm thông tin.

POSIX khuyến nghị sử dụng sigaction()thay vì signal(), do hành vi chưa được chỉ định và các biến thể triển khai quan trọng của nó. POSIX cũng định nghĩa nhiều tín hiệu hơn tiêu chuẩn ISO C, bao gồm SIGUSR1và SIGUSR2, lập trình viên có thể sử dụng tự do cho bất kỳ mục đích nào.

# ĐỐI SỐ BIẾN

|  |  |
| --- | --- |
| Tham số | Chi tiết |
| va\_list ap | con trỏ đối số, vị trí hiện tại trong danh sách đối số biến đổi |
| last | tên của đối số hàm không biến đổi cuối cùng, để trình biên dịch tìm đúng vị trí để bắt đầu xử lý các đối số biến đổi; có thể không được khai báo dưới dạng một registerbiến, một hàm hoặc một kiểu mảng |
| type | loại đối số biến đổi được quảng cáoint để đọc (ví dụ đối với một short intđối số) |
| va\_list src | con trỏ đối số hiện tại để sao chép |
| va\_list dst | danh sách đối số mới được điền vào |

**Các đối số biến** được sử dụng bởi các hàm trong họ printf ( printf, fprintf, v.v.) và các hàm khác để cho phép một hàm được gọi với số lượng đối số khác nhau mỗi lần, do đó có tên varargs (biến số đối số)

Để triển khai các hàm sử dụng tính năng biến đối số, sử dụng #include <stdarg.h>.

Để gọi các hàm có chấp nhận số lượng đối số biến đổi, đảm bảo rằng có một nguyên mẫu đầy đủ với dấu ba chấm ở cuối: ví dụ void err\_exit(const char \*format, ...);

## Sử dụng đối số đếm rõ ràng để xác định độ dài của VA\_LIST

Với bất kỳ hàm biến đổi nào, hàm phải biết cách giải thích danh sách đối số biến đổi. Với các hàm printf() hoặc scanf(), chuỗi định dạng nói cho hàm biết điều gì cần mong đợi.

Kỹ thuật đơn giản nhất là chuyển một đếm rõ ràng của các đối số khác (mà thường là cùng loại).

Điều này được thể hiện trong hàm biến đối số trong mã dưới đây, tính tổng của một loạt các số nguyên, nơi có thể có bất kỳ số nguyên nào nhưng số lượng đó được chỉ định là một đối số trước danh sách đối số biến đổi.

|  |
| --- |
| #include <stdio.h>  #include <stdarg.h>  /\* đối số đầu tiên là số lượng đối số int tiếp theo cần tổng. \*/  int sum(int n, ...) {  int sum = 0;  va\_list it; /\* lưu thông tin về danh sách đối số biến đổi. \*/  va\_start(it, n); /\* bắt đầu xử lý đối số biến đổi \*/  while (n--)  sum += va\_arg(it, int); /\* lấy và cộng các đối số biến đổi tiếp theo \*/  va\_end(it); /\* kết thúc xử lý đối số biến đổi \*/  return sum;  }  int main(int argc, char\* argv[]) {  printf("%d\n", sum(5, 1, 2, 3, 4, 5)); /\* in ra 15 \*/  printf("%d\n", sum(10, 5, 9, 2, 5, 111, 6666, 42, 1, 43, -6218)); /\* in ra 666 \*/  return 0;  } |

## SỬ DỤNG GIÁ TRỊ KẾT THÚC ĐỂ XÁC ĐỊNH PHẦN CUỐI CỦA VA\_LIST

Với bất kỳ hàm biến đổi nào, hàm phải biết cách giải thích danh sách đối số biến đổi. Phương pháp "truyền thống" (được minh họa bởi printf) là xác định số lượng đối số từ đầu. Tuy nhiên, điều này không phải lúc nào cũng là một ý tưởng tốt:

|  |
| --- |
| /\* Đối số đầu tiên xác định số lượng tham số; số còn lại cũng là int \*/  extern int sum(int n, ...);  /\* Nhưng điều này không rõ ràng từ code. \*/  sum(5, 2, 1, 4, 3, 6)  /\* Điều gì sẽ xảy ra nếu một đối số được loại bỏ sau này? \*/  sum(5, 2, 1, 3, 6) /\* Thảm họa \*/ |

Đôi khi, việc thêm một trình kết thúc rõ ràng, được minh họa bằng execlp() hàm POSIX, sẽ hiệu quả hơn. Đây là một hàm khác để tính tổng của một loạt các số thực

|  |
| --- |
| #include <stdarg.h>  #include <stdio.h>  #include <math.h>  /\* Cộng các đối số cho đến giá trị kết thúc NAN \*/  double sum (double x, ...) {  double sum = 0;  va\_list va;  va\_start(va, x);  for (; !isnan(x); x = va\_arg(va, double)){  sum += x;  }  va\_end(va);  return sum;  }  int main(int argc, char\* argv[]) {  printf("%g\n", sum(5., 2., 1., 4., 3., 6., NAN));  printf("%g\n", sum(1, 0.5, 0.25, 0.125, 0.0625, 0.03125, NAN));  } |

Một số giá trị kết thúc tốt:

* Kiểu số nguyên (giả định là tất cả đều là số dương hoặc không âm) - 0 hoặc -1
* Các kiểu số dấu chấm động - NAN
* Các kiểu con trỏ - NULL
* Các kiểu enumerator - một số giá trị đặc biệt

## TRIỂN KHAI CÁC HÀM VỚI GIAO DIỆN GIỐNG NHƯ ‘ PRINTF() ’

Một trong những cách thường thấy sử dụng danh sách đối số biến đổi là triển khai các hàm mỏng manh bọc xung quanh gia đình hàm printf(). Một ví dụ như vậy là một tập hợp các hàm báo cáo lỗi.

**errmsg.h**

|  |
| --- |
| #ifndef ERRMSG\_H\_INCLUDED  #define ERRMSG\_H\_INCLUDED  #include <stdarg.h>  #include <stdnoreturn.h> // C11  void verrmsg(int errnum, const char \*fmt, va\_list ap);  noreturn void errmsg(int exitcode, int errnum, const char \*fmt, ...);  void warnmsg(int errnum, const char \*fmt, ...);  #endif |

Đây là một ví dụ cơ bản; những gói như vậy có thể phức tạp hơn. Thông thường, người lập trình sẽ sử dụng hàm errmsg() hoặc warnmsg(), chính chúng sử dụng verrmsg() bên trong. Nếu ai đó cần làm nhiều hơn, thì hàm verrmsg() được hiển thị sẽ hữu ích. Bạn có thể tránh hiển thị nó cho đến khi bạn cần (YAGNI - bạn không cần nó), nhưng cần sẽ xuất hiện sớm hoặc muộn (bạn sẽ cần nó - YAGNI).

**errmsg.c**

Đoạn mã này chỉ cần chuyển các đối số biến đổi cho hàm vfprintf() để in ra tiêu chuẩn lỗi. Nó cũng thông báo về thông báo lỗi hệ thống tương ứng với số lỗi hệ thống (errno) được chuyển đến các hàm.

|  |
| --- |
| #include "errmsg.h"  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  void verrmsg(int errnum, const char \*fmt, va\_list ap) {  if (fmt)  vfprintf(stderr, fmt, ap);  if (errnum != 0)  fprintf(stderr, ": %s", strerror(errnum));  putc('\n', stderr);  }  void errmsg(int exitcode, int errnum, const char \*fmt, ...) {  va\_list ap;  va\_start(ap, fmt);  verrmsg(errnum, fmt, ap);  va\_end(ap);  exit(exitcode);  }  void warnmsg(int errnum, const char \*fmt, ...) {  va\_list ap;  va\_start(ap, fmt);  verrmsg(errnum, fmt, ap);  va\_end(ap);  } |

**Sử dụng errmsg.h**

Bây giờ bạn có thể sử dụng các hàm như sau:

|  |
| --- |
| #include "errmsg.h"  #include <errno.h>  #include <fcntl.h>  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  int main(int argc, char \*\*argv) {  char buffer[BUFSIZ];  int fd;  if (argc != 2)  {  fprintf(stderr, "Usage: %s filename\n", argv[0]);  exit(EXIT\_FAILURE);  }  const char \*filename = argv[1];  if ((fd = open(filename, O\_RDONLY)) == -1)  errmsg(EXIT\_FAILURE, errno, "cannot open %s", filename);  if (read(fd, buffer, sizeof(buffer)) != sizeof(buffer))  errmsg(EXIT\_FAILURE, errno, "cannot read %zu bytes from %s", sizeof(buffer), filename);  if (close(fd) == -1)  warnmsg(errno, "cannot close %s", filename);  /\* tiếp tục chương trình \*/  return 0;  } |

Nếu lệnh gọi open() hoặc read() hệ thống không thành công, lỗi được ghi thành lỗi tiêu chuẩn và chương trình thoát với mã thoát 1. Nếu lệnh close() gọi hệ thống không thành công, lỗi chỉ được in dưới dạng thông báo cảnh báo và chương trình sẽ tiếp tục.

**Kiểm tra việc sử dụng đúng các định dạng printf()**

Nếu bạn đang sử dụng GCC (Trình biên dịch GNU C, là một phần của Bộ sưu tập Trình biên dịch GNU) hoặc sử dụng Clang, thì bạn có thể yêu cầu trình biên dịch kiểm tra xem các đối số mà bạn chuyển đến các hàm thông báo lỗi có khớp với những gì mong đợi hay không printf(). Vì không phải tất cả các trình biên dịch đều hỗ trợ phần mở rộng, nên nó cần được biên dịch theo điều kiện, điều này hơi khó một chút. Tuy nhiên, sự bảo vệ mà nó mang lại rất đáng để nỗ lực.

Trước tiên, chúng ta cần biết cách phát hiện trình biên dịch là GCC hoặc Clang mô phỏng GCC. Câu trả lời là GCC định nghĩa \_\_GNUC\_\_để chỉ ra điều đó.

Xem các thuộc tính chức năng phổ biến để biết thông tin về các thuộc tính — cụ thể là format thuộc tính.

Viết lại **errmsg.h**

|  |
| --- |
| #ifndef ERRMSG\_H\_INCLUDED  #define ERRMSG\_H\_INCLUDED  #include <stdarg.h>  #include <stdnoreturn.h> // C11  #if !defined(PRINTFLIKE)  #if defined(\_\_GNUC\_\_)  #define PRINTFLIKE(n,m) \_\_attribute\_\_((format(printf,n,m)))  #else  #define PRINTFLIKE(n,m) /\* If only \*/  #endif /\* \_\_GNUC\_\_ \*/  #endif /\* PRINTFLIKE \*/  void verrmsg(int errnum, const char \*fmt, va\_list ap);  void noreturn errmsg(int exitcode, int errnum, const char \*fmt, ...)  PRINTFLIKE(3, 4);  void warnmsg(int errnum, const char \*fmt, ...)  PRINTFLIKE(2, 3);  #endif |

Bây giờ, nếu bạn mắc lỗi như:

|  |
| --- |
| rrmsg(EXIT\_FAILURE, errno, "Failed to open file '%d' for reading", filename); |

(nơi %d nên là %s), thì trình biên dịch sẽ phàn nàn:

|  |
| --- |
| $ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes \  > -Wold-style-definition -c erruse.c  erruse.c: In function ‘main’:  erruse.c:20:64: error: format ‘%d’ expects argument of type ‘int’, but argument 4 has type ‘const char \*’ [-Werror=format=]  errmsg(EXIT\_FAILURE, errno, "Failed to open file '%d' for reading", filename);  ~^  %s  cc1: all warnings being treated as errors  $ |

## SỬ DỤNG CHUỖI ĐỊNH DẠNG

Việc sử dụng một chuỗi định dạng cung cấp thông tin về số lượng và loại dự kiến ​​của các đối số biến thiên tiếp theo theo cách để tránh nhu cầu về đối số đếm rõ ràng hoặc giá trị của dấu kết thúc.

Ví dụ dưới đây thể hiện một hàm bọc chức năng printf() tiêu chuẩn, chỉ cho phép sử dụng các đối số biến đổi có kiểu char, int và double (trong định dạng điểm động thập phân thập phân). Ở đây, tương tự như với printf(), đối số đầu tiên của hàm bọc là chuỗi định dạng. Khi chuỗi định dạng được phân tích cú pháp, hàm có thể xác định xem có một đối số biến đổi khác được mong đợi và loại của nó là gì.

|  |
| --- |
| #include <stdio.h>  #include <stdarg.h>  int simple\_printf(const char \*format, ...) {  va\_list ap; /\* lưu thông tin về danh sách đối số biến đổi \*/  int printed = 0; /\* đếm số ký tự được in \*/  va\_start(ap, format); /\* bắt đầu xử lý đối số biến đổi \*/  while (\*format != '\0') /\* đọc chuỗi định dạng cho đến ký tự kết thúc chuỗi \*/ {  int f = 0;  if (\*format == '%')  {  ++format;  switch(\*format)  {  case 'c' :  f = printf("%d", va\_arg(ap, int)); /\* in đối số biến đổi tiếp theo, lưu ý kiểu  thăng cấp từ char thành int \*/  break;  case 'd' :  f = printf("%d", va\_arg(ap, int)); /\* in đối số biến đổi tiếp theo \*/  break;  case 'f' :  f = printf("%f", va\_arg(ap, double)); /\* in đối số biến đổi tiếp theo \*/  break;  default :  f = -1; /\* thông báo định dạng không hợp lệ \*/  break;  }  }  else  {  f = printf("%c", \*format); /\* in các ký tự khác \*/  }  if (f < 0) /\* kiểm tra lỗi \*/  {  printed = f;  break;  }  else  {  printed += f;  }  ++format; /\* tiếp tục đến ký tự tiếp theo trong chuỗi \*/  }  va\_end(ap); /\* kết thúc xử lý đối số biến đổi \*/  return printed;  }  int main (int argc, char \*argv[]) {  int x = 40;  int y = 0;  y = simple\_printf("There are %d characters in this sentence", x);  simple\_printf("\n%d were printed\n", y);  } |

# KHẲNG ĐỊNH

|  |  |
| --- | --- |
| **Tham số** | **Chi tiết** |
| expression | Biểu thức có kiểu scalar. |
| message | chuỗi ký tự được bao gồm trong thông báo chẩn đoán. |

Một khẳng định là một tiền đề rằng điều kiện đã được trình bày phải là đúng tại thời điểm khẳng định được gặp bởi phần mềm. Thông thường, khẳng định đơn giản là các kiểm tra được thực hiện tại thời điểm thực thi. Tuy nhiên, khẳng định tĩnh được kiểm tra tại thời gian biên dịch.

## KHẲNG ĐỊNH ĐƠN GIẢN

Khẳng định là một câu lệnh được sử dụng để khẳng định rằng một sự thật phải đúng khi dòng mã đó đến. Khẳng định rất hữu ích để đảm bảo các điều kiện mong đợi được đáp ứng. Khi điều kiện được chuyển vào một khẳng định là đúng, không có hành động nào được thực hiện. Hành vi trên điều kiện sai phụ thuộc vào các cờ trình biên dịch. Khi khẳng định được kích hoạt, đầu vào sai sẽ dẫn đến dừng chương trình ngay lập tức. Khi chúng được vô hiệu hóa, không có hành động nào được thực hiện. Thông thường, thực hành phổ biến là bật khẳng định trong các phiên bản nội bộ và gỡ lỗi, và tắt chúng trong các phiên bản phát hành, mặc dù các khẳng định thường được bật trong các phiên bản phát hành. (Việc dừng chương trình tốt hơn hay xấu hơn lỗi phụ thuộc vào chương trình.) Khẳng định nên chỉ được sử dụng để phát hiện lỗi lập trình nội bộ, điều này thường đồng nghĩa với việc chuyển các tham số không hợp lệ.

|  |
| --- |
| #include <stdio.h>  /\* Uncomment to disable `assert()` \*/  /\* #define NDEBUG \*/  #include <assert.h>  int main(int argc, char \*argv[]) {  int x = -1;  assert(x >= 0);  printf("x = %d\n", x);  return 0;  } |

Kết quả có thể có khi NDEBUG không được định nghĩa:

|  |
| --- |
| a.out: main.c:9: main: Assertion `x >= 0' failed. |

Kết quả có thể có khi NDEBUG được định nghĩa:

|  |
| --- |
| x = -1 |

Việc thực hành tốt là định nghĩa NDEBUG một cách toàn cầu, để bạn có thể dễ dàng biên dịch mã của mình với tất cả các khẳng định bật hoặc tắt. Một cách dễ dàng để làm điều này là định nghĩa NDEBUG như một tùy chọn cho trình biên dịch, hoặc định nghĩa nó trong một tiêu đề cấu hình chia sẻ (ví dụ: config.h).

## KHẲNG ĐỊNH TĨNH

Phiên bản ≥ C11

Khẳng định tĩnh được sử dụng để kiểm tra xem điều kiện có đúng khi mã được biên dịch hay không. Nếu không đúng, trình biên dịch sẽ phải phát ra thông báo lỗi và dừng quá trình biên dịch.

Khẳng định tĩnh là một kiểm tra được thực hiện tại thời gian biên dịch, không phải thời gian chạy. Điều kiện phải là một biểu thức hằng số, và nếu sai sẽ dẫn đến lỗi trình biên dịch. Đối số đầu tiên, điều kiện được kiểm tra, phải là một biểu thức hằng số, và tham số thứ hai là một chuỗi ký tự cố định.

Khác với **assert**, **\_Static\_assert** là một từ khóa. Một tiện ích macro **static\_assert** được định nghĩa trong **<assert.h>.**

|  |
| --- |
| #include <assert.h>  enum {N = 5};  \_Static\_assert(N == 5, "N does not equal 5");  static\_assert(N > 10, "N is not greater than 10"); /\* lỗi biên dịch \*/ |

Phiên bản = C99

Trước C11, không có hỗ trợ trực tiếp cho khẳng định tĩnh. Tuy nhiên, trong C99, khẳng định tĩnh có thể được mô phỏng bằng các macro sẽ kích hoạt lỗi biên dịch nếu điều kiện thời gian biên dịch là sai. Khác với \_Static\_assert, tham số thứ hai cần là một tên mã thích hợp để có thể tạo ra một tên biến từ nó. Nếu khẳng định thất bại, tên biến sẽ được thấy trong thông báo lỗi của trình biên dịch, vì biến đó đã được sử dụng trong một khai báo mảng cú pháp không chính xác.

|  |
| --- |
| #define STATIC\_MSG(msg, l) STATIC\_MSG2(msg, l)  #define STATIC\_MSG2(msg,l) on\_line\_##l##\_\_##msg  #define STATIC\_ASSERT(x, msg) extern char STATIC\_MSG(msg, \_\_LINE\_\_) [(x)?1:-1]  enum { N = 5 };  STATIC\_ASSERT(N == 5, N\_must\_equal\_5);  STATIC\_ASSERT(N > 5, N\_must\_be\_greater\_than\_5); /\* lỗi biên dịch \*/ |

Trước C99, bạn không thể khai báo biến tại các vị trí tùy ý trong một khối, vì vậy bạn phải cực kỳ cẩn trọng khi sử dụng macro này, đảm bảo rằng nó chỉ xuất hiện ở nơi mà một khai báo biến sẽ hợp lệ

## KHẲNG ĐỊNH TIN NHẮN LỖI

Tồn tại một thủ thuật có thể hiển thị thông báo lỗi cùng với xác nhận. Thông thường, bạn sẽ viết mã như thế này

|  |
| --- |
| void f(void \*p) {  assert(p != NULL);  /\* more code \*/  } |

Nếu xác nhận không thành công, một thông báo lỗi sẽ giống như

Assertion failed: p != NULL, file main.c, line 5

Tuy nhiên, bạn cũng có thể sử dụng logic AND (&&) để đưa ra thông báo lỗi

|  |
| --- |
| void f(void \*p) {  assert(p != NULL && "function f: p cannot be NULL");  /\* more code \*/  } |

Bây giờ, nếu xác nhận không thành công, một thông báo lỗi sẽ đọc như thế này

Assertion failed: p != NULL && "function f: p cannot be NULL", file main.c, line 5

Lý do tại sao điều này hoạt động là một chuỗi ký tự luôn đánh giá khác không (đúng). Việc thêm && 1 vào một biểu thức Boolean không có tác dụng. Do đó, việc thêm vào && "**error message**"cũng không có tác dụng gì, ngoại trừ việc trình biên dịch sẽ hiển thị toàn bộ biểu thức bị lỗi.

## KHẲNG ĐỊNH MÃ KHÔNG THỂ TRUY CẬP ĐƯỢC

Trong quá trình phát triển, khi một số lối đi mã phải được ngăn cản khỏi việc kiểm soát luồng, bạn có thể sử dụng assert(0) để chỉ ra rằng điều kiện như vậy là sai:

|  |
| --- |
| switch (color) {  case COLOR\_RED:  case COLOR\_GREEN:  case COLOR\_BLUE:  break;  default:  assert(0);  } |

Khi đối số của macro assert() được đánh giá sai, macro sẽ viết thông tin chẩn đoán vào luồng lỗi chuẩn và sau đó dừng chương trình. Thông tin này bao gồm tệp và số dòng của câu lệnh assert() và có thể rất hữu ích trong gỡ lỗi. Khẳng định có thể bị vô hiệu hóa bằng cách định nghĩa macro NDEBUG.

Một cách khác để kết thúc chương trình khi xảy ra lỗi là sử dụng các hàm thư viện tiêu chuẩn exit, quick\_exit hoặc abort. exit và quick\_exit lấy một đối số có thể được truyền lại cho môi trường của bạn. abort() (và do đó assert) có thể là một kết thúc rất nghiêm trọng của chương trình của bạn, và một số công việc dọn dẹp mà sẽ được thực hiện ở cuối thực thi, có thể không được thực hiện.

Lợi thế chính của assert() là nó tự động in thông tin gỡ lỗi. Gọi abort() có lợi thế là không thể vô hiệu hóa như một assert, nhưng nó có thể không gây ra bất kỳ thông tin gỡ lỗi nào được hiển thị. Trong một số tình huống, sử dụng cả hai cấu trúc cùng nhau có thể có lợi:

|  |
| --- |
| if (color == COLOR\_RED || color == COLOR\_GREEN) {  ...  } else if (color == COLOR\_BLUE) {  ...  } else {  assert(0), abort();  } |

Khi asserts được kích hoạt, cuộc gọi assert() sẽ in thông tin gỡ lỗi và kết thúc chương trình. Thực thi không bao giờ đạt đến cuộc gọi abort(). Khi asserts được vô hiệu hóa, cuộc gọi assert() không làm gì cả và abort() được gọi. Điều này đảm bảo rằng chương trình luôn kết thúc cho điều kiện lỗi này; kích hoạt và vô hiệu hóa asserts chỉ ảnh hưởng đến việc có hay không có đầu ra gỡ lỗi được in.

Bạn không bao giờ nên để một assert như vậy trong mã sản xuất, vì thông tin gỡ lỗi không hữu ích cho người dùng cuối và vì abort thường là một cách kết thúc quá mức nghiêm trọng và ngăn cản các trình xử lý dọn dẹp được cài đặt để chạy cho exit hoặc quick\_exit.

## TIỀN ĐIỀU KIỆN VÀ HẬU ĐIỀU KIỆN

Một trong những trường hợp sử dụng cho khẳng định là kiểm tra điều kiện tiên quyết và điều kiện hậu quả. Điều này rất hữu ích để duy trì các điều kiện bất biến và thiết kế theo hợp đồng. Ví dụ, một độ dài luôn luôn là số không hoặc dương, do đó hàm này phải trả về một số không hoặc dương.

|  |
| --- |
| #include <stdio.h>  /\* Uncomment to disable `assert()` \*/  /\* #define NDEBUG \*/  #include <assert.h>  int length2 (int \*a, int count) {  int i, result = 0;  /\* Điều kiện tiên quyết: \*/  /\* NULL là một vector không hợp lệ \*/  assert (a != NULL);  /\* Số chiều không thể là số âm. \*/  assert (count >= 0);  /\* Tính toán \*/  for (i = 0; i < count; ++i)  {  result = result + (a[i] \* a[i]);  }  /\* Điều kiện hậu quả: \*/  /\* Độ dài kết quả không thể là số âm. \*/  assert (result >= 0);  return result;  }  #define COUNT 3  int main(int argc, char \*argv[]) {  int a[COUNT] = {1, 2, 3};  int \*b = NULL;  int r;  r = length2 (a, COUNT);  printf ("r = %i\n", r);  r = length2 (b, COUNT);  printf ("r = %i\n", r);  return 0;  } |

# LỰA CHỌN TỔNG QUÁT

|  |  |
| --- | --- |
| **Tham số** | **Chi tiết** |
| generic-assoc-list | generic-association OR generic-assoc-list , generic-association |
| generic-association | type-name : assignment-expression OR default : assignment-expression |

## KIỂM TRA XEM BIẾN CÓ PHẢI LÀ MỘT KIỂU CỤ THỂ ĐƯỢC CHÚ THÍCH

Mã chương trình dưới đây dùng \_Generic macro để kiểm tra xem biến có phải là kiểu con trỏ int hằng hay kiểu con trỏ int không phải hằng, và đưa ra thông báo tương ứng:

|  |
| --- |
| #include <stdio.h>  #define is\_const\_int(x) \_Generic((&x), \  const int \*: "a const int", \  int \*: "a non-const int", \  default: "of other type")\  int main(int argc, char \*argv[]) {  const int i = 1;  int j = 1;  double k = 1.0;  printf("i is %s\n", is\_const\_int(i));  printf("j is %s\n", is\_const\_int(j));  printf("k is %s\n", is\_const\_int(k));  } |

Kết quả

|  |
| --- |
| i is a const int  j is a non-const int  k is of other type |

Tuy nhiên, nếu macro generic type được triển khai như sau:

|  |
| --- |
| #define is\_const\_int(x) \_Generic((x), \  const int: "a const int", \  int: "a non-const int", \  default: "of other type") |

Kết quả sẽ là

|  |
| --- |
| i is a non-const int  j is a non-const int  k is of other type |

Điều này xảy ra vì tất cả các kiểu bổ sung đều bị loại bỏ khi đánh giá biểu thức điều khiển trong một \_Generic primary expression.

## LỰA CHỌN THÔNG QUA KIỂU DỮ LIỆU CHUNG DỰA TRÊN NHIỀU ĐỐI SỐ

Nếu muốn thực hiện lựa chọn dựa trên nhiều đối số cho biểu thức kiểu dữ liệu chung, và tất cả các kiểu dữ liệu đều là kiểu số học, một cách dễ dàng để tránh việc nhúng các biểu thức \_Generic lồng nhau là sử dụng phép cộng của các tham số trong biểu thức kiểm soát:

|  |
| --- |
| int max\_int(int, int);  unsigned max\_unsigned(unsigned, unsigned);  double max\_double(double, double);  #define MAX(X, Y) \_Generic((X)+(Y), \  int: max\_int, \  unsigned: max\_unsigned, \  default: max\_double) \  ((X), (Y)) |

Ở đây, biểu thức kiểm soát (X)+(Y) chỉ được kiểm tra theo kiểu dữ liệu và không được đánh giá. Các chuyển đổi thông thường cho các toán hạng số học được thực hiện để xác định kiểu dữ liệu được chọn.

Để giải quyết tình huống phức tạp hơn, lựa chọn có thể được thực hiện dựa trên nhiều hơn một đối số của toán tử, bằng cách lồng chúng lại với nhau.

Ví dụ này lựa chọn giữa bốn hàm được triển khai bên ngoài, lấy các kết hợp của hai đối số int và/ hoặc string, và trả về tổng của chúng.

|  |
| --- |
| int AddIntInt(int a, int b);  int AddIntStr(int a, const char\* b);  int AddStrInt(const char\* a, int b );  int AddStrStr(const char\* a, const char\* b);  #define AddStr(y) \  \_Generic((y), int: AddStrInt, \  char\*: AddStrStr, \  const char\*: AddStrStr )  #define AddInt(y) \  \_Generic((y), int: AddIntInt, \  char\*: AddIntStr, \  const char\*: AddIntStr )  #define Add(x, y) \  \_Generic((x) , int: AddInt(y) , \  char\*: AddStr(y) , \  const char\*: AddStr(y)) \  ((x), (y))  int main(int argc, char \*argv[]) {  int result = 0;  result = Add( 100 , 999 );  result = Add( 100 , "999" );  result = Add( "100" , 999 );  result = Add( "100" , "999" );  const int a = -123;  char b[] = "4321";  result = Add( a , b );  int c = 1;  const char d[] = "0";  result = Add( d , ++c );  } |

Mặc dù có vẻ như đối số yđược đánh giá nhiều lần, nhưng đó không phải là 1 . Cả hai đối số chỉ được đánh giá một lần, ở cuối macro Add: ( x , y ), giống như trong một lệnh gọi hàm thông thường.

1 (Trích dẫn từ: ISO:IEC 9899:201X 6.5.1.1 Lựa chọn chung 3)

Biểu thức kiểm soát của lựa chọn chung không được đánh giá.

## LOẠI MACRO IN CHUNG

|  |
| --- |
| #include <stdio.h>  void print\_int(int x) { printf("int: %d\n", x); }  void print\_dbl(double x) { printf("double: %g\n", x); }  void print\_default() { puts("unknown argument"); }  #define print(X) \_Generic((X), \  int: print\_int, \  double: print\_dbl, \  default: print\_default)(X)  int main(int argc, char \*argv[]) {  print(42);  print(3.14);  print("hello, world");  } |

Kết quả

|  |
| --- |
| int: 42  double: 3.14  unknown argument |

Lưu ý rằng nếu kiểu không phải là int hoặc double, một cảnh báo sẽ được tạo ra. Để loại bỏ cảnh báo, bạn có thể thêm kiểu đó vào macro print(X).

# X- MACROS

X-macro là một kỹ thuật dựa trên bộ tiền xử lý để giảm thiểu mã lặp lại và duy trì sự tương ứng giữa dữ liệu/mã. Nhiều phần mở rộng macro riêng biệt dựa trên một tập hợp dữ liệu chung được hỗ trợ bằng cách đại diện cho toàn bộ nhóm phần mở rộng thông qua một macro chính duy nhất, với văn bản thay thế của macro đó bao gồm một chuỗi các phần mở rộng của macro bên trong, mỗi phần mở rộng cho mỗi mốc dữ liệu. Macro trong thực tế thường được đặt tên là X(), vì vậy được gọi là X macros.

## SỬ DỤNG ĐƠN GIẢN CỦA X- MACROS CHO CÁC CÂU LỆNH PRINTF()

|  |
| --- |
| /\* Define một danh sách các mã thông báo trình biên dịch mà X sẽ được gọi tên \*/  #define X\_123 X(1) X(2) X(3)  /\* Define X mà sẽ được sử dụng \*/  #define X(val) printf("X(%d) made this print\n", val);  X\_123  #undef X  /\* Thói quen tốt là undef X để dễ dàng sử dụng lại sau này \*/ |

Ví dụ này sẽ dẫn đến trình biên dịch sinh ra mã sau đây:

|  |
| --- |
| printf("X(%d) thực hiện in này\n", 1);  printf("X(%d) thực hiện in này\n", 2);  printf("X(%d) thực hiện in này\n", 3); |

## MỞ RỘNG : SỬ DỤNG X- MACRO NHƯ LÀ ĐỐI SỐ

Cách tiếp cận X-macro có thể được khái quát hóa một chút bằng cách đặt tên của macro "X" thành đối số của macro chính. Điều này có ưu điểm là giúp tránh xung đột tên macro và cho phép sử dụng macro có mục đích chung làm macro "X".

Như mọi khi với macro X, macro chính đại diện cho danh sách các mục có ý nghĩa cụ thể đối với macro đó. Trong biến thể này, một macro như vậy có thể được định nghĩa như sau:

|  |
| --- |
| /\* declare list of items \*/  #define ITEM\_LIST(X) \  X(item1) \  X(item2) \  X(item3) \  /\* end of list \*/ |

Sau đó, người ta có thể tạo mã để in tên vật phẩm như vậy:

|  |
| --- |
| /\* define macro to apply \*/  #define PRINTSTRING(value) printf( #value "\n");  /\* apply macro to the list of items \*/  ITEM\_LIST(PRINTSTRING) |

Điều đó mở rộng sang mã này:

|  |
| --- |
| printf( "item1" "\n"); printf( "item2" "\n"); printf( "item3" "\n"); |

Trái ngược với macro X tiêu chuẩn, trong đó tên "X" là một đặc điểm tích hợp sẵn của macro chính, với kiểu này, có thể không cần thiết hoặc thậm chí không mong muốn sau đó xác định macro được sử dụng làm đối số ( trong ví dụ này) **PRINTSTRING.**

## GIÁ TRỊ ENUM VÀ MÃ ĐỊNH DANH

|  |
| --- |
| /\* declare items of the enum \*/  #define FOREACH \  X(item1) \  X(item2) \  X(item3) \  /\* end of list \*/  /\* define the enum values \*/  #define X(id) MyEnum\_ ## id,  enum MyEnum { FOREACH };  #undef X  /\* convert an enum value to its identifier \*/  const char \* enum2string(int enumValue)  {  const char\* stringValue = NULL;  #define X(id) if (enumValue == MyEnum\_ ## id) stringValue = #id;  FOREACH  #undef X  return stringValue;  } |

Tiếp theo, bạn có thể sử dụng giá trị liệt kê trong mã của mình và dễ dàng in mã định danh của nó bằng cách sử dụng:

|  |
| --- |
| printf("%s\n", enum2string(MyEnum\_item2)); |

## TẠO MÃ

X-Macro có thể được sử dụng để tạo mã, bằng cách viết mã lặp đi lặp lại: lặp qua một danh sách để thực hiện một số công việc hoặc để khai báo một tập hợp các hằng số, đối tượng hoặc hàm.

**Ở đây, chúng ta sử dụng X-Macro để khai báo một enum chứa 4 lệnh và một bảng ánh xạ tên của chúng thành chuỗi.**

Sau đó, chúng ta có thể in ra các giá trị chuỗi của enum.

|  |
| --- |
| /\* All our commands \*/  #define COMMANDS(OP) OP(Open) OP(Close) OP(Save) OP(Quit)  /\* generate the enum Commands: {cmdOpen, cmdClose, cmdSave, cmdQuit, }; \*/  #define ENUM\_NAME(name) cmd##name,  enum Commands {  COMMANDS(ENUM\_NAME)  };  #undef ENUM\_NAME  /\* generate the string table \*/  #define COMMAND\_OP(name) #name,  const char\* const commandNames[] = {  COMMANDS(COMMAND\_OP)  };  #undef COMMAND\_OP  /\* the following prints "Quit\n": \*/  printf("%s\n", commandNames[cmdQuit]()); |

**Tương tự, chúng ta có thể tạo một bảng nhảy (jump table) để gọi các hàm dựa trên giá trị của enum.**

Điều này yêu cầu tất cả các hàm có cùng chữ ký. Nếu chúng không có đối số và trả về một giá trị int, chúng ta sẽ đặt nó vào một tệp tiêu đề với định nghĩa enum.

|  |
| --- |
| /\* declare all functions as extern \*/  #define EXTERN\_FUNC(name) extern int doCmd##name(void);  COMMANDS(EXTERN\_FUNC)  #undef EXTERN\_FUNC  /\* declare the function pointer type and the jump table \*/  typedef int (\*CommandFunc)(void);  extern CommandFunc commandJumpTable[]; |

Tất cả những điều sau đây có thể nằm trong các đơn vị biên dịch khác nhau giả sử phần trên được bao gồm dưới dạng tiêu đề:

|  |
| --- |
| /\* generate the jump table \*/  #define FUNC\_NAME(name) doCmd##name,  CommandFunc commandJumpTable[] = {  COMMANDS(FUNC\_NAME)  };  #undef FUNC\_NAME  /\* call the save command like this: \*/  int result = commandJumpTable[cmdSave]();  /\* somewhere else, we need the implementations of the commands \*/  int doCmdOpen(void) {/\* code performing open command \*/}  int doCmdClose(void) {/\* code performing close command \*/}  int doCmdSave(void) {/\* code performing save command \*/}  int doCmdQuit(void) {/\* code performing quit command \*/} |

Một ví dụ về kỹ thuật này đang được sử dụng trong mã thực là GPU command dispatching in Chromium.

# ALIASING VÀ EFFECTIVE TYPE

## EFFECTIVE TYPE

Effective type của một đối tượng dữ liệu là loại thông tin cuối cùng được liên kết với nó, nếu có.

|  |
| --- |
| // a normal variable, effective type uint32\_t, and this type never changes  uint32\_t a = 0.0;  // effective type of \*pa is uint32\_t, too, simply  // because \*pa is the object a  uint32\_t\* pa = &a;  // the object pointed to by q has no effective type, yet  void\* q = malloc(sizeof uint32\_t);  // the object pointed to by q still has no effective type,  // because nobody has written to it  uint32\_t\* qb = q;  // \*qb now has effective type uint32\_t because a uint32\_t value was written  \*qb = 37;  // the object pointed to by r has no effective type, yet, although  // it is initialized  void\* r = calloc(1, sizeof uint32\_t);  // the object pointed to by r still has no effective type,  // because nobody has written to or read from it  uint32\_t\* rc = r;  // \*rc now has effective type uint32\_t because a value is read  // from it with that type. The read operation is valid because we used calloc.  // Now the object pointed to by r (which is the same as \*rc) has  // gained an effective type, although we didn't change its value.  uint32\_t c = \*rc;  // the object pointed to by s has no effective type, yet.  void\* s = malloc(sizeof uint32\_t);  // the object pointed to by s now has effective type uint32\_t  // because an uint32\_t value is copied into it.  memcpy(s, r, sizeof uint32\_t); |

Lưu ý rằng với ví dụ cuối cùng, không cần thiết phải có một con trỏ uint32\_t\* đến đối tượng đó. Việc sao chép một đối tượng uint32\_t khác là đủ để có effective type

## RESTRICT QUALIFICATION

Nếu chúng ta có hai đối số con trỏ cùng kiểu dữ liệu, trình biên dịch không thể đưa ra bất kỳ giả định nào và sẽ luôn phải giả định rằng thay đổi \*e có thể thay đổi \*f:

|  |
| --- |
| void fun(float\* e, float\* f) {  float a = \*f;  \*e = 22;  float b = \*f;  printf("is %g equal to %g?\n", a, b);  }  float fval = 4;  float eval = 77;  fun(&eval, &fval); |

tất cả diễn ra tốt đẹp và một cái gì đó giống như

|  |
| --- |
| is 4 equal to 4? |

được in. Nếu chúng ta truyền cùng một con trỏ, chương trình vẫn sẽ làm đúng và in

|  |
| --- |
| is 4 equal to 22? |

Điều này có thể trở nên không hiệu quả, nếu chúng ta biết bởi một số thông tin bên ngoài e và f sẽ không bao giờ trỏ đến cùng một đối tượng dữ liệu. Chúng ta có thể phản ánh kiến ​​thức đó bằng cách thêm từ khóa **restrict** vào các tham số con trỏ:

|  |
| --- |
| void fan(float\* restrict e, float\* restrict f) {  float a = \*f;  \*e = 22;  float b = \*f;  printf("is %g equal to %g?\n", a, b);  } |

Sau đó, trình biên dịch có thể luôn cho rằng điều đó e và f trỏ đến các đối tượng khác nhau.

## THAY ĐỔI CÁC BYTE

Sau khi một đối tượng có một effective type, bạn không nên cố gắng sửa đổi nó thông qua một con trỏ thuộc kiểu khác, trừ khi kiểu khác đó là một kiểu ký tự, **char, signed char hoặc unsigned char.**

|  |
| --- |
| #include <inttypes.h>  #include <stdio.h>  int main(void) {  uint32\_t a = 57;  // Ép kiểu cho việc chuyển đổi từ các kiểu không tương thích!  unsigned char\* ap = (unsigned char\*)&a;  for (size\_t i = 0; i < sizeof a; ++i) {  /\* set each byte of a to 42 \*/  ap[i] = 42;  }  printf("a now has value %" PRIu32 "\n", a);  } |

Đây là một chương trình hợp lệ in ra:

|  |
| --- |
| a now has value 707406378 |

Cách hoạt động của mã code này:

* Truy cập được thực hiện vào các byte riêng lẻ của biến a được xem bằng kiểu unsigned char, vì vậy mỗi việc sửa đổi là xác định.
* Hai cách xem đối tượng, thông qua a và thông qua \*ap, gọi là alias, nhưng vì ap là con trỏ trỏ tới kiểu ký tự, quy tắc aliasing nghiêm ngặt không áp dụng. Do đó, trình biên dịch phải giả định rằng giá trị của a có thể đã thay đổi trong vòng lặp for. Giá trị đã sửa đổi của a phải được xây dựng từ các byte đã được thay đổi.
* Kiểu của a, uint32\_t không có các bit dư thừa (padding bits). Tất cả các bit của biểu diễn đều tính cho giá trị, ở đây là 707406378, và không thể có giá trị trap(bẫy).

## CÁC KIỂU KÍ TỰ KHÔNG THỂ TRUY CẬP THÔNG QUA CÁC KIỂU KHÔNG PHẢI KIỂU KÍ TỰ

Nếu một đối tượng được định nghĩa với kiểu lưu trữ tĩnh (static), thread (thread\_local), hoặc tự động (automatic), và nó có kiểu ký tự, bao gồm: char, unsigned char hoặc signed char, thì nó không được truy cập bằng một kiểu không phải ký tự. Trong ví dụ dưới đây, một mảng char được diễn giải dạng kiểu int, và hành vi của con trỏ int b không xác định khi truy cập các giá trị của mảng a.

|  |
| --- |
| #include <stdio.h>  int main(void) {  char a[100];  int\* b = (int\*)&a;  \*b = 1;  static char c[100];  b = (int\*)&c;  \*b = 2;  \_Thread\_local char d[100];  b = (int\*)&d;  \*b = 3;  } |

Đoạn mã này không xác định vì nó vi phạm quy tắc "kiểu hiệu quả" (effective type), không có đối tượng dữ liệu nào có kiểu hiệu quả có thể được truy cập thông qua một kiểu khác không phải ký tự. Vì kiểu khác ở đây là int, điều này không được phép.

Thậm chí nếu việc căn chỉnh (alignment) và kích thước con trỏ có thể phù hợp, điều này cũng không miễn trừ khỏi quy tắc này, hành vi vẫn là không xác định.

Điều này có nghĩa đặc biệt là không có cách nào trong C tiêu chuẩn để dành một đối tượng buffer với kiểu ký tự mà có thể sử dụng thông qua các con trỏ với các kiểu khác nhau, như bạn sử dụng một buffer được nhận từ malloc hoặc các hàm tương tự.

Một cách đúng để đạt được cùng mục tiêu như trong ví dụ trên là sử dụng union.

|  |
| --- |
| #include <stdio.h>  typedef union bufType bufType;  union bufType {  char c[sizeof(int[25])];  int i[25];  };  int main(int argc, char \*argv[]) {  bufType a = { .c = { 0 } }; // dành một buffer và khởi tạo  int\* b = a.i; // không cần ép kiểu  \*b = 1;  static bufType c = { .c = { 0 } };  b = c.i;  \*b = 2;  \_Thread\_local bufType d = { .c = { 0 } };  b = d.i;  \*b = 3;  } |

Ở đây, union đảm bảo rằng trình biên dịch biết từ đầu rằng buffer có thể được truy cập thông qua các view khác nhau. Điều này cũng có lợi thế là giờ đây buffer có một "view" là a.i và đã là kiểu int nên không cần phải chuyển đổi con trỏ.

## VI PHẠM QUY TẮC “STRICT ALIASING”

Trong đoạn mã sau, giả định đơn giản là kiểu dữ liệu float và uint32\_t có cùng kích thước.

|  |
| --- |
| #include <stdio.h>  #include <stdint.h>  void fun(uint32\_t\* u, float\* f) {  float a = \*f;  \*u = 22;  float b = \*f;  printf("%g should equal %g\n", a, b);  } |

Biến u và f có kiểu cơ sở khác nhau, do đó trình biên dịch có thể giả định rằng chúng trỏ đến các đối tượng khác nhau. Không có khả năng là \*f có thể thay đổi giữa hai lần khởi tạo a và b, và do đó trình biên dịch có thể tối ưu mã thành một cái gì đó tương đương với:

|  |
| --- |
| void fun(uint32\_t\* u, float\* f) {  float a = \*f;  \*u = 22;  printf("%g should equal %g\n", a, a);  } |

Tức là, thao tác tải thứ hai của \*f có thể được tối ưu hoàn toàn.

Nếu chúng ta gọi hàm này "bình thường":

|  |
| --- |
| float fval = 4;  uint32\_t uval = 77;  fun(&uval, &fval); |

Tất cả đều ổn và một cái gì đó tương tự như:

|  |
| --- |
| 4 should equal 4 |

được in ra. Nhưng nếu chúng ta "lừa đảo" và truyền cùng một con trỏ, sau khi chuyển đổi nó:

|  |
| --- |
| float fval = 4;  uint32\_t\* up = (uint32\_t\*)&fval;  fun(up, &fval); |

Chúng ta vi phạm quy tắc aliasing nghiêm ngặt. Sau đó, hành vi trở thành không xác định. Kết quả có thể như trên, nếu trình biên dịch tối ưu hóa thao tác truy cập thứ hai, hoặc hoàn toàn khác nhau, và do đó chương trình của bạn sẽ trở thành một trạng thái không đáng tin cậy hoàn toàn.

# BIÊN DỊCH

Ngôn ngữ C truyền thống là một ngôn ngữ được biên dịch (trái ngược phiên dịch). Tiêu chuẩn C xác định các giai đoạn dịch, và kết quả của việc áp dụng chúng là một hình ảnh chương trình (hoặc chương trình đã được biên dịch). Trong phiên bản C11, các giai đoạn được liệt kê trong §5.1.1.2.

## TRÌNH BIÊN DỊCH (THE COMPILER)

Sau khi trình **tiền xử lý** C đã bao gồm tất cả các **tệp tiêu đề** và mở rộng tất cả **các macro**, trình biên dịch có thể biên dịch chương trình. Nó làm điều này bằng cách chuyển mã nguồn C thành một tệp **mã đối tượng**, đó là một tệp có đuôi .o chứa phiên bản nhị phân của mã nguồn. Tuy nhiên mã đối tượng không thể thực thi trực tiếp. Để tạo một chương trình thực thi, bạn cũng phải thêm mã cho tất cả các hàm thư viện đã được #include vào tệp (điều này không giống như việc bao gồm các khai báo, đó là điều mà #include làm). Điều này là công việc của trình liên kết (**Linker**).

Nói chung, trình tự chính xác cách gọi trình biên dịch C phụ thuộc nhiều vào hệ thống mà bạn đang sử dụng. Ở đây chúng tôi đang sử dụng trình biên dịch GCC, mặc dù cần lưu ý rằng còn tồn tại nhiều trình biên dịch khác:

|  |
| --- |
| % gcc -Wall -c foo.c |

‘%’ là dấu nhắc lệnh của hệ điều hành. Dòng lệnh này yêu cầu trình biên dịch chạy trước tiên trình tiền xử lý trên tệp **foo.c** sau đó biên dịch nó thành tệp mã đối tượng **foo.o**. Tùy chọn **-c** có nghĩa là biên dịch tệp mã nguồn thành tệp đối tượng nhưng không gọi trình liên kết. Tùy chọn **-c** này có sẵn trên các hệ thống POSIX, như Linux hoặc macOS; các hệ thống khác có thể sử dụng cú pháp khác nhau.

Nếu toàn bộ chương trình của bạn nằm trong một tệp mã nguồn, bạn có thể làm như sau:

|  |
| --- |
| % gcc -Wall foo.c -o foo |

Dòng lệnh này yêu cầu trình biên dịch chạy trước tiên trình tiền xử lý trên foo.c, sau đó biên dịch nó và liên kết nó để tạo một chương trình có tên là foo. Tùy chọn -o cho biết rằng từ tiếp theo trên dòng là tên tệp thực thi nhị phân (chương trình). Nếu bạn không chỉ định -o (nếu bạn chỉ gõ gcc foo.c), chương trình thực thi sẽ có tên a.out vì lý do lịch sử.

Nói chung, trình biên dịch thực hiện bốn bước khi chuyển đổi một tệp .c thành một chương trình thực thi:

1. **Tiền xử lý (pre-processing)** - mở rộng văn bản các chỉ thị #include và macro #define trong tệp .c của bạn.
2. **Biên dịch (compilation)** - chuyển đổi chương trình thành mã hợp ngữ (bạn có thể dừng trình biên dịch ở bước này bằng cách thêm tùy chọn -S).
3. **Hợp ngữ (assembly)** - chuyển đổi mã hợp ngữ thành mã máy.
4. **Liên kết (linkage**) - liên kết mã đối tượng với các thư viện bên ngoài để tạo một chương trình thực thi.

Lưu ý rằng tên trình biên dịch chúng ta đang sử dụng là GCC, viết tắt của cả "GNU C compiler" và "GNU compiler collection", phụ thuộc vào ngữ cảnh. Có các trình biên dịch C khác tồn tại. Đối với các hệ điều hành giống Unix, nhiều trong số chúng có tên cc, viết tắt của "C compiler", thường là liên kết mục tiêu cho một số trình biên dịch khác. Trên các hệ thống Linux, cc thường là một bí danh của GCC. Trên macOS hoặc OS-X, nó trỏ đến clang.

Tiêu chuẩn POSIX hiện tại mục đích c99 là tên của trình biên dịch C — nó hỗ trợ tiêu chuẩn C99 theo mặc định. Trước đó, phiên bản POSIX mục đích là trình biên dịch c89. POSIX cũng mục đích rằng trình biên dịch này hiểu các tùy chọn -c và -o mà chúng ta đã sử dụng ở trên.

Lưu ý: Tùy chọn -Wall xuất hiện trong cả hai ví dụ gcc trên để yêu cầu trình biên dịch in ra cảnh báo về các cấu trúc đáng ngờ, điều này được đề xuất mạnh mẽ. Nên thêm các tùy chọn cảnh báo khác, ví dụ: -Wextra.

## FILE TYPES

Biên dịch chương trình C yêu cầu bạn làm việc với năm loại tệp (files):

1. **Tệp nguồn (Source files):** Đây là các tệp chứa định nghĩa hàm và thường có đuôi .c theo quy ước. Lưu ý rằng .cc và .cpp là tệp C++, không phải tệp C.

Ví dụ: foo.c

1. **Tệp tiêu đề (Header files):** Đây là các tệp chứa các khai báo hàm và các câu lệnh tiền xử lý khác (xem bên dưới). Chúng được sử dụng để cho phép các tệp mã nguồn truy cập vào các hàm được định nghĩa bên ngoài. Tệp tiêu đề kết thúc bằng .h theo quy ước.

Ví dụ: foo.h

1. **Tệp đối tượng (Object files):** Đây là các tệp được tạo ra như kết quả của trình biên dịch. Chúng bao gồm định nghĩa hàm dưới dạng nhị phân, nhưng chúng không thể thực thi một mình. Tệp đối tượng kết thúc bằng .o theo quy ước, tuy nhiên trên một số hệ điều hành (ví dụ: Windows, MS-DOS), chúng thường kết thúc bằng .obj.

Ví dụ: foo.o foo.obj

1. **Tệp thực thi nhị phân (Binary executables):** Đây là kết quả của trình liên kết. Trình liên kết liên kết một số tệp đối tượng để tạo ra một tệp nhị phân có thể thực thi trực tiếp. Trên hệ điều hành Unix, tệp thực thi nhị phân không có đuôi đặc biệt, tuy nhiên chúng thường kết thúc bằng .exe trên Windows.

Ví dụ: foo foo.exe

1. **Thư viện (Libraries):** Một thư viện là một tệp nhị phân được biên dịch, nhưng không phải là một tệp thực thi (tức không có hàm main() trong thư viện). Một thư viện chứa các hàm có thể được sử dụng bởi nhiều chương trình. Thư viện nên đi kèm với các tệp tiêu đề chứa các khai báo cho tất cả các hàm trong thư viện; các tệp tiêu đề này nên được tham chiếu (ví dụ: #include <library.h>) trong bất kỳ tệp mã nguồn nào sử dụng thư viện. Sau đó, trình liên kết cần tham chiếu đến thư viện để chương trình có thể được biên dịch thành công. Có hai loại thư viện: static và dynamic.

* **Thư viện tĩnh (Static library):** Một thư viện tĩnh (.a trên các hệ thống POSIX và .lib trên Windows — không nên nhầm lẫn với các tệp thư viện nhập khẩu DLL, cũng sử dụng phần mở rộng .lib) được xây dựng tĩnh vào chương trình. Thư viện tĩnh có lợi thế là chương trình biết chính xác phiên bản của thư viện nào được sử dụng. Tuy nhiên, kích thước của các tệp thực thi sẽ lớn hơn vì tất cả các hàm thư viện được sử dụng đều được bao gồm.

Ví dụ: libfoo.a foo.lib

* **Thư viện động (Dynamic library):** Một thư viện động (.so trên hầu hết các hệ thống POSIX, .dylib trên OSX và .dll trên Windows) được liên kết động vào thời gian chạy bởi chương trình. Đôi khi còn được gọi là thư viện chia sẻ vì một hình ảnh thư viện có thể được chia sẻ bởi nhiều chương trình. Thư viện động có lợi thế tiết kiệm không gian đĩa nếu nhiều ứng dụng sử dụng cùng một thư viện. Hơn nữa, chúng cho phép cập nhật thư viện (sửa lỗi) mà không cần phải xây dựng lại các tệp thực thi.

Ví dụ: foo.so foo.dylib foo.dll

## THE LINKER (TRÌNH LIÊN KẾT)

Trình liên kết có nhiệm vụ liên kết các tệp đối tượng (.o files) thành một chương trình thực thi nhị phân. Quá trình liên kết chủ yếu bao gồm giải quyết các địa chỉ tượng trưng thành các địa chỉ số. Kết quả của quá trình liên kết thường là một chương trình thực thi.

Trong quá trình liên kết, trình liên kết sẽ thu thập tất cả các mô-đun đối tượng được chỉ định trên dòng lệnh, thêm vào mã khởi đầu cụ thể của hệ thống và cố gắng giải quyết tất cả các tham chiếu ngoài trong các mô-đun đối tượng với các định nghĩa bên ngoài trong các tệp đối tượng khác (các tệp đối tượng có thể được chỉ định trực tiếp trên dòng lệnh hoặc được thêm vào ngầm qua thư viện). Sau đó, trình liên kết sẽ gán địa chỉ nạp cho các tệp đối tượng, tức là, nó chỉ định nơi mã và dữ liệu sẽ xuất hiện trong không gian địa chỉ của chương trình hoàn chỉnh. Khi đã có địa chỉ nạp, nó có thể thay thế tất cả các địa chỉ tượng trưng trong mã đối tượng bằng địa chỉ "thực", số học trong không gian địa chỉ của mục tiêu. Chương trình đã sẵn sàng để thực thi.

Quá trình này bao gồm cả các tệp đối tượng được trình biên dịch từ các tệp mã nguồn của bạn cũng như các tệp đối tượng đã được biên dịch trước cho bạn và được tập hợp thành các tệp thư viện. Các tệp này có tên kết thúc bằng .a hoặc .so, và bạn thường không cần phải biết về chúng, vì trình liên kết biết vị trí của hầu hết chúng và sẽ liên kết chúng vào tự động khi cần thiết.

### Liên kết ngầm định của trình liên kết

Giống như trình tiền xử lý, trình liên kết là một chương trình riêng biệt, thường được gọi là ld (nhưng Linux sử dụng collect2, ví dụ). Giống như trình tiền xử lý, trình liên kết được gọi tự động cho bạn khi bạn sử dụng trình biên dịch. Do đó, cách thông thường để sử dụng trình liên kết là như sau:

|  |
| --- |
| % gcc foo.o bar.o baz.o -o myprog |

Dòng lệnh này yêu cầu trình biên dịch liên kết ba tệp đối tượng (foo.o, bar.o và baz.o) thành một tệp nhị phân thực thi có tên myprog. Bây giờ bạn có một tệp có tên myprog mà bạn có thể chạy và có hy vọng sẽ thực hiện điều gì đó thú vị và/hoặc hữu ích.

### Yêu cầu rõ ràng của trình liên kết

Có thể gọi trình liên kết trực tiếp, nhưng điều này thường không được khuyến khích và thường đặc thù cho mỗi nền tảng. Điều này có nghĩa là các tùy chọn hoạt động trên Linux không nhất thiết phải hoạt động trên Solaris, AIX, macOS, Windows và tương tự đối với bất kỳ nền tảng nào khác. Nếu bạn làm việc với GCC, bạn có thể sử dụng gcc -v để xem những gì được thực thi thay cho bạn.

### Tùy chọn cho trình liên kết

Trình liên kết cũng có một số đối số để thay đổi hành vi của nó. Lệnh sau sẽ yêu cầu gcc liên kết foo.o và bar.o, nhưng cũng bao gồm thư viện ncurses.

|  |
| --- |
| % gcc foo.o bar.o -o foo -lncurses |

Điều này thực sự (hơn hoặc ít hơn) tương đương với

|  |
| --- |
| % gcc foo.o bar.o /usr/lib/libncurses.so -o foo |

(tuy nhiên libncurses.so có thể là libncurses.a, đó chỉ là một lưu trữ được tạo ra với ar). Lưu ý rằng bạn nên liệt kê các thư viện (hoặc thông qua các tùy chọn -lname) sau các tệp đối tượng. Với thư viện tĩnh, thứ tự mà chúng được chỉ định quan trọng; thường, với các thư viện chia sẻ, thứ tự không quan trọng.

Lưu ý rằng trên nhiều hệ thống, nếu bạn sử dụng các hàm toán học (từ <math.h>), bạn cần chỉ định -lm để tải thư viện toán học - nhưng macOS và macOS Sierra không yêu cầu điều này. Có các thư viện khác là các thư viện riêng biệt trên Linux và các hệ thống Unix khác, nhưng không phải trên macOS - các tiểu chuẩn POSIX, và POSIX realtime, và các thư viện mạng là ví dụ. Do đó, quá trình liên kết thay đổi giữa các nền tảng.

### Những tùy chọn biên dịch khác

Điều này là tất cả những gì bạn cần biết để bắt đầu biên dịch chương trình C của riêng bạn. Nói chung, chúng tôi cũng khuyến nghị bạn sử dụng tùy chọn dòng lệnh -Wall:

|  |
| --- |
| % gcc -Wall -c foo.c |

Tùy chọn -Wall cho phép trình biên dịch cảnh báo về các cấu trúc mã hợp pháp nhưng đáng nghi và sẽ giúp bạn phát hiện rất nhiều lỗi sớm.

Nếu bạn muốn trình biên dịch ném nhiều cảnh báo hơn (bao gồm các biến được khai báo nhưng không được sử dụng, quên trả về một giá trị v.v.), bạn có thể sử dụng tập lệnh này, vì -Wall, bất kể tên gọi, không bật tất cả các cảnh báo có thể có:

|  |
| --- |
| % gcc -Wall -Wextra -Wfloat-equal -Wundef -Wcast-align -Wwrite-strings -Wlogical-op \  > -Wmissing-declarations -Wredundant-decls -Wshadow … |

Lưu ý rằng clang có tùy chọn -Weverything thực sự bật tất cả các cảnh báo trong clang.

## TIỀN XỬ LÝ (THE PREPROCESSOR)

Trước khi trình biên dịch C bắt đầu biên dịch một tệp mã nguồn, tệp đó được xử lý trong một giai đoạn tiền xử lý. Giai đoạn này có thể được thực hiện bởi một chương trình riêng biệt hoặc hoàn toàn tích hợp trong một tệp thực thi duy nhất. Dù cho trường hợp nào, nó được tự động gọi bởi trình biên dịch trước khi bắt đầu quá trình biên dịch chính thức. Giai đoạn tiền xử lý chuyển đổi mã nguồn của bạn thành mã nguồn hoặc đơn vị dịch thuật khác bằng cách áp dụng các thay thế văn bản. Bạn có thể xem nó như là một mã nguồn "sửa đổi" hoặc "mở rộng". Mã nguồn được mở rộng này có thể tồn tại dưới dạng tệp thực tế trong hệ thống tệp hoặc nó có thể chỉ được lưu trữ trong bộ nhớ trong một khoảng thời gian ngắn trước khi được xử lý tiếp.

Các lệnh tiền xử lý bắt đầu bằng dấu thăng (#). Có một số lệnh tiền xử lý; hai lệnh quan trọng nhất là:

1. **Defines:**

#define chủ yếu được sử dụng để định nghĩa các hằng số. Ví dụ:

|  |
| --- |
| #define BIGNUM 1000000  int a = BIGNUM; |

Sẽ trở thành

|  |
| --- |
| int a = 1000000; |

#define được sử dụng theo cách này để tránh việc phải viết rõ ràng một số hằng số trong nhiều vị trí khác nhau trong tệp mã nguồn. Điều này quan trọng trong trường hợp bạn cần thay đổi giá trị hằng số sau này; việc thay đổi nó một lần trong #define ít gặp lỗi hơn việc phải thay đổi nó ở nhiều vị trí khác nhau trên mã.

Vì #define chỉ thực hiện tìm kiếm và thay thế nâng cao, bạn cũng có thể khai báo các macros. Ví dụ:

|  |
| --- |
| #define ISTRUE(stm) do{stm = stm ? 1 : 0;}while(0)  // trong hàm:  a = x;  ISTRUE(a); |

Sẽ trở thành

|  |
| --- |
| // trong hàm:  a = x;  do {  a = a ? 1 : 0;  } while(0); |

Ở phép tính gần đúng đầu tiên, hiệu ứng này là tương tự như với hàm inline, nhưng trình tiền xử lý không cung cấp kiểm tra kiểu cho các macros #define. Điều này đã được biết đến là dễ gây lỗi và việc sử dụng chúng đòi hỏi cẩn thận.

Lưu ý ở đây, trình tiền xử lý cũng sẽ thay thế các chú thích bằng khoảng trắng như đã được giải thích ở dưới.

**Includes:**

|  |
| --- |
| #include được sử dụng để truy cập các định nghĩa hàm được xác định bên ngoài một tệp mã nguồn. Ví dụ:  #include <stdio.h> |

sẽ khiến trình tiền xử lý dán nội dung của <stdio.h> vào tệp mã nguồn tại vị trí câu lệnh #include trước khi được biên dịch. #include hầu như luôn được sử dụng để bao gồm các tệp tiêu đề, các tệp chứa chủ yếu các khai báo hàm và các câu lệnh #define. Trong trường hợp này, chúng ta sử dụng #include để có thể sử dụng các hàm như printf và scanf, các khai báo của chúng được đặt trong tệp stdio.h. Trình biên dịch C không cho phép bạn sử dụng một hàm trừ khi nó đã được khai báo hoặc định nghĩa trước trong tệp đó; câu lệnh #include là cách để tái sử dụng mã đã được viết trước đó trong chương trình C của bạn.

**Các phép toán Logic**

|  |
| --- |
| #if defined A || defined B  variable = another\_variable + 1;  #else  variable = another\_variable \* 2;  #endif |

Sẽ được thay đổi thành:

|  |
| --- |
| variable = another\_variable + 1; |

nếu A hoặc B được xác định ở đâu đó trong dự án trước đó. Nếu điều này không đúng, tất nhiên trình tiền xử lý sẽ thực hiện điều này:

|  |
| --- |
| variable = another\_variable \* 2; |

Điều này thường được sử dụng cho mã chạy trên các hệ thống khác nhau hoặc biên dịch trên các trình biên dịch khác nhau. Vì có các định nghĩa toàn cầu, được xác định cho trình biên dịch / hệ thống cụ thể, bạn có thể thử nghiệm trên những định nghĩa đó và luôn để trình biên dịch chỉ sử dụng mã mà anh ta sẽ biên dịch chắc chắn.

1. **Comments**

Trình tiền xử lý thay thế tất cả các chú thích trong tệp nguồn bằng một khoảng trắng duy nhất. Chú thích được chỉ định bằng // cho đến cuối dòng hoặc một kết hợp dấu mở /\* và dấu đóng \*/.

## THE TRANSLATION PHASES (CÁC GIAI ĐOẠN DỊCH)

Tiêu chuẩn C 2011 xác định quá trình dịch mã nguồn thành hình ảnh chương trình (ví dụ, tệp thực thi) trong 8 bước theo thứ tự, được liệt kê trong §5.1.1.2 Translation Phases:

1. Dữ liệu đầu vào của tệp nguồn được ánh xạ vào tập ký tự nguồn (nếu cần). Trigraphs, là các chuỗi ba ký tự được sử dụng để biểu diễn các ký tự đặc biệt trong các bộ ký tự không có các ký tự này, sẽ được thay thế ở bước này.
2. Các dòng tiếp tục, là các dòng kết thúc bằng dấu gạch chéo ngược (), được ghép với dòng kế tiếp, hiệu quả tạo ra một dòng logic từ nhiều dòng vật lý.
3. Mã nguồn được phân tích thành các token trắng và tiền xử lý. Các khoảng trống, bình luận và các ký tự phân tách khác được sử dụng để tách mã nguồn thành các token riêng lẻ, chẳng hạn như từ khóa, nhận dạng, toán tử và hằng số.
4. Tiền xử lý được áp dụng, thực thi các chỉ thị, mở rộng các macro và áp dụng các pragma. Tiền xử lý xử lý mã nguồn, thay thế các macro bằng các định nghĩa tương ứng, xử lý các chỉ thị #include để bao gồm các tệp khác và thực thi các chỉ thị tiền xử lý khác. Mỗi tệp nguồn được đưa vào bằng #include trải qua các bước dịch 1 đến 4 đệ quy nếu cần. Tất cả các chỉ thị liên quan đến tiền xử lý sau đó được xóa khỏi mã đã xử lý.
5. Giá trị tập ký tự nguồn trong các hằng số ký tự và chuỗi được ánh xạ vào tập ký tự thực thi. Bước này đảm bảo rằng các hằng số ký tự và chuỗi được biểu diễn bằng cách sử dụng mã hóa phù hợp cho môi trường đích.
6. Chuỗi ký tự liền kề với nhau sẽ được nối lại. Nếu có nhiều chuỗi ký tự được đặt liên tiếp nhau, trình biên dịch kết hợp chúng thành một chuỗi ký tự duy nhất.
7. Mã nguồn được phân tích thành các token, tạo thành đơn vị dịch. Các token được phân loại tiếp theo dựa trên loại của chúng, chẳng hạn như nhận dạng, từ khóa, toán tử và hằng số.
8. Các tham chiếu bên ngoài được giải quyết và hình ảnh chương trình được hình thành. Trong bước này, trình biên dịch liên kết các đơn vị dịch khác nhau, giải quyết các tham chiếu bên ngoài đến các hàm và biến và tạo ra hình ảnh chương trình cuối cùng hoặc tệp thực thi.

Lưu ý rằng một triển khai của trình biên dịch C có thể kết hợp nhiều bước lại với nhau để tăng hiệu suất, miễn là chương trình kết quả vẫn hoạt động như nếu các bước trên đã xảy ra riêng biệt và theo thứ tự đã chỉ định.

# INLINE ASSEMBLY

## GCC INLINE ASSEMBLY IN MACROS

Chúng ta có thể đặt các lệnh hợp ngữ bên trong macro và sử dụng macro giống như cách bạn gọi một hàm.

|  |
| --- |
| #define mov(x,y) \  { \  \_\_asm\_\_ ("l.cmov %0,%1,%2" : "=r" (x) : "r" (y), "r" (0x0000000F)); \  }  // một số định nghĩa và gán  unsigned char sbox[size][size];  unsigned char sbox[size][size];  // Sử dụng  mov(state[0][1], sbox[si][sj]); |

Sử dụng các lệnh trình biên dịch hợp ngữ nhúng trong mã C có thể cải thiện thời gian chạy của chương trình. Điều này rất hữu ích trong các tình huống thời gian quan trọng như các thuật toán mật mã như AES. Ví dụ: đối với một thao tác dịch đơn giản cần thiết trong thuật toán AES, chúng ta có thể thay thế một lệnh trình biên dịch hợp ngữ xoay phải trực tiếp bằng toán tử dịch C >>.

Trong một triển khai của 'AES256', trong hàm 'AddRoundKey()', chúng ta có một số câu như sau:

|  |
| --- |
| unsigned int w; // 32-bit  unsigned char subkey[4]; // 8-bit, 4\*8 = 32  subkey[0] = w >> 24; // giữ 8 bit, MSB, nhóm 8 bit ngoài cùng bên trái  subkey[1] = w >> 16; // giữ 8 bit, nhóm 8 bit thứ hai từ bên trái  subkey[2] = w >> 8; // giữ 8 bit, nhóm 8 bit thứ hai từ bên phải  subkey[3] = w; // giữ 8 bit, LSB, nhóm 8 bit ngoài cùng bên phải  /// subkey <- w |

Chúng đơn giản gán giá trị bit của w cho mảng subkey.

Chúng ta có thể thay đổi ba biểu thức C dịch + gán và một biểu thức gán với chỉ một thao tác xoay phải trình biên dịch hợp ngữ.

|  |
| --- |
| \_\_asm\_\_ ("l.ror %0,%1,%2" : "=r" (\* (unsigned int \*) subkey) : "r" (w), "r" (0x10)); |

Kết quả cuối cùng chính xác như nhau.

## HỖ TRỢ ASM CƠ BẢN TRONG GCC

Hỗ trợ asm cơ bản với gcc có cú pháp như sau:

|  |
| --- |
| asm [ volatile ] ( AssemblerInstructions ) |

Trong đó, AssemblerInstructions là mã assembly trực tiếp cho bộ xử lý được sử dụng. Từ khóa volatile là tùy chọn và không có tác dụng gì vì gcc không tối ưu hóa mã trong một câu lệnh asm cơ bản. AssemblerInstructions có thể chứa nhiều lệnh assembly. Một câu lệnh asm cơ bản được sử dụng khi bạn có một hàm asm phải tồn tại bên ngoài một hàm C. Ví dụ sau đây được lấy từ tài liệu GCC:

|  |
| --- |
| /\* Lưu ý rằng đoạn mã này sẽ không biên dịch được với tùy chọn -masm=intel \*/  #define DebugBreak() asm("int $3") |

Trong ví dụ này, bạn có thể sử dụng DebugBreak() ở nơi khác trong mã của bạn và nó sẽ thực thi lệnh assembly "int $3". Lưu ý rằng mặc dù gcc sẽ không thay đổi bất kỳ mã nào trong một câu lệnh asm cơ bản, trình tối ưu hóa vẫn có thể di chuyển các câu lệnh asm liền nhau. Nếu bạn có nhiều lệnh assembly phải xảy ra theo một thứ tự cụ thể, hãy bao gồm chúng trong một câu lệnh asm duy nhất.

## HỖ TRỢ ASM MỞ RỘNG TRONG GCC

Hỗ trợ asm mở rộng trong gcc có cú pháp như sau:

|  |
| --- |
| asm [volatile] ( AssemblerTemplate  : OutputOperands  [ : InputOperands  [ : Clobbers ] ])  asm [volatile] goto ( AssemblerTemplate  :  : InputOperands  : Clobbers  : GotoLabels) |

Trong đó, AssemblerTemplate là mẫu cho lệnh assembler, OutputOperands là bất kỳ biến C nào có thể bị thay đổi bởi mã assembly, InputOperands là bất kỳ biến C nào được sử dụng như tham số đầu vào, Clobbers là danh sách các thanh ghi bị thay đổi bởi mã assembly, và GotoLabels là bất kỳ nhãn lệnh goto nào có thể được sử dụng trong mã assembly.

Định dạng mở rộng này được sử dụng trong các hàm C và là cách sử dụng inline assembly phổ biến hơn. Dưới đây là một ví dụ từ nhân kernel Linux để hoán đổi byte của các số 16-bit và 32-bit trên bộ xử lý ARM:

|  |
| --- |
| /\* Từ arch/arm/include/asm/swab.h trong phiên bản kernel Linux 4.6.4 \*/  #if \_\_LINUX\_ARM\_ARCH\_\_ >= 6  static inline \_\_attribute\_const\_\_ \_\_u32 \_\_arch\_swahb32(\_\_u32 x)  {  \_\_asm\_\_ ("rev16 %0, %1" : "=r" (x) : "r" (x));  return x;  }  #define \_\_arch\_swahb32 \_\_arch\_swahb32  #define \_\_arch\_swab16(x) ((\_\_u16)\_\_arch\_swahb32(x))  static inline \_\_attribute\_const\_\_ \_\_u32 \_\_arch\_swab32(\_\_u32 x)  {  \_\_asm\_\_ ("rev %0, %1" : "=r" (x) : "r" (x));  return x;  }  #define \_\_arch\_swab32 \_\_arch\_swab32  #endif |

Mỗi phần asm sử dụng biến x làm tham số đầu vào và đầu ra của nó. Hàm C sau đó trả về kết quả được thay đổi.

Với định dạng asm mở rộng, gcc có thể tối ưu các lệnh assembly trong khối asm theo các quy tắc tối ưu hóa mã C. Nếu bạn muốn khối asm của bạn không bị tác động, hãy sử dụng từ khóa volatile cho khối asm.

# PHẠM VI CỦA ĐỊNH DANH

## Phạm vi của khai báo hàm

Khi chúng ta khai báo một hàm trong ngôn ngữ C, tên của các tham số trong khai báo hàm đó chỉ có phạm vi tồn tại bên trong khai báo hàm và không có ý nghĩa bên ngoài nó. Dưới đây là một ví dụ minh họa:

|  |
| --- |
| #include <stdio.h>  int test\_function(int apple); // Khai báo hàm test\_function với tham số apple  int main(int argc, char \*argv[]) {  int orange = 5; // Khởi tạo biến orange với giá trị 5  orange = test\_function(orange); // Gọi hàm test\_function và truyền biến orange vào làm tham số  printf("%d\r\n", orange); // In giá trị của biến orange sau khi gọi hàm (orange = 6)  return 0;  }  int test\_function(int fruit) // Định nghĩa hàm test\_function với tham số fruit  {  fruit += 1; // Tăng giá trị của tham số fruit lên 1  return fruit; // Trả về giá trị của tham số fruit  } |

Lưu ý rằng bạn sẽ nhận được thông báo lỗi khó hiểu nếu bạn giới thiệu tên một kiểu trong một khai báo hàm

|  |
| --- |
| int function(struct whatever \*arg);  struct whatever  {  int a;  // ...  };  int function(struct whatever \*arg)  {  return arg->a;  } |

Với GCC 6.3.0, mã nguồn này (tệp dc11.c) sẽ tạo ra thông báo lỗi:

|  |
| --- |
| $ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -c dc11.c  dc11.c:1:25: error: ‘struct whatever’ declared inside parameter list will not be visible outside of this definition or declaration [-Werror]  int function(struct whatever \*arg);  ^~~~~~~~  dc11.c:9:9: error: conflicting types for ‘function’  int function(struct whatever \*arg)  ^~~~~~~~  dc11.c:1:9: note: previous declaration of ‘function’ was here  int function(struct whatever \*arg);  ^~~~~~~~  cc1: all warnings being treated as errors  $ |

Đặt định nghĩa structure trước khai báo hàm hoặc thêm "struct whatever;" trước khai báo hàm, và không có vấn đề gì xảy ra. Bạn không nên giới thiệu các tên kiểu mới trong khai báo hàm vì không có cách nào sử dụng kiểu đó và do đó không thể định nghĩa hoặc sử dụng hàm đó.

## Phạm vi khối

Một identifier có phạm vi khối nếu khai báo tương ứng của nó xuất hiện bên trong một khối (khai báo tham số trong định nghĩa hàm áp dụng). Phạm vi kết thúc ở cuối khối tương ứng.

Không có các thực thể khác nhau với cùng một identifier có thể có cùng phạm vi, nhưng phạm vi có thể trùng lặp. Trong trường hợp trùng lặp phạm vi, chỉ thực thể được khai báo trong phạm vi bên trong nhất mới hiển thị.

|  |
| --- |
| #include <stdio.h>  void test(int bar) // bar có phạm vi khối hàm test  {  int foo = 5; // foo có phạm vi khối hàm test  {  int bar = 10; // bar có phạm vi khối bên trong, điều này trùng lặp với khai báo test:bar trước đó và nó che khuất test:bar  printf("%d %d\n", foo, bar); // 5 10  } // kết thúc phạm vi cho thanh bên trong  printf("%d %d\n", foo, bar); // 5 5, ở đây bar là test:bar  } // kết thúc phạm vi cho test:foo và test:bar  int main(void)  {  int foo = 3; // foo có phạm vi khối hàm chính  printf("%d\n", foo); // 3  test(5);  printf("%d\n", foo); // 3  return 0;  } // kết thúc phạm vi cho main:fo |

## Phạm vi của tệp

|  |
| --- |
| #include <stdio.h>  /\* Định danh, foo, được khai báo bên ngoài tất cả các khối.  Nó có thể được sử dụng ở bất cứ đâu sau khi khai báo cho đến khi kết thúc bộ dịch.\*/  static int foo;  void test\_function(void)  {  foo += 2;  }  int main(void)  {  foo = 1;  test\_function();  printf("%d\r\n", foo); //foo = 3;  return 0;  } |

## Phạm vi hàm

Phạm vi hàm là phạm vi đặc biệt dành cho các nhãn (labels) trong ngôn ngữ lập trình C. Điều này do tính chất đặc biệt của nhãn. Một nhãn có thể được nhìn thấy trong toàn bộ hàm mà nó được định nghĩa, và ta có thể nhảy đến nó (sử dụng câu lệnh goto với tên nhãn) từ bất kỳ điểm nào trong cùng một hàm. Mặc dù không có ích lợi nhiều, ví dụ dưới đây minh họa điều này:

|  |
| --- |
| #include <stdio.h>  int main(int argc, char\* argv[]) {  int a = 0;  goto INSIDE;  OUTSIDE:  if (a != 0) {  int i = 0;  INSIDE:  printf("a=%d\n", a);  goto OUTSIDE;  }  } |

Nhãn INSIDE có vẻ như được định nghĩa trong khối if, giống như biến i mà phạm vi của nó chỉ trong khối. Tuy nhiên, thực tế là nhãn INSIDE có thể nhìn thấy trong toàn bộ hàm, như ví dụ về câu lệnh goto INSIDE; minh họa. Do đó, không thể có hai nhãn cùng tên trong một hàm duy nhất.

Một cách sử dụng tiềm năng của việc sử dụng nhãn như CLEANUP1 và CLEANUP2 trong ví dụ dưới đây là thực hiện các công việc dọn dẹp phức tạp cho các tài nguyên đã cấp phát:

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  void a\_function(void) {  double\* a = malloc(sizeof(double[34]));  if (!a) {  fprintf(stderr, "can't allocate\n");  return; /\* Không cần giải phóng a nếu nó là null \*/  }  FILE\* b = fopen("some\_file", "r");  if (!b) {  fprintf(stderr, "can't open\n");  goto CLEANUP1; /\* Giải phóng a; không cần đóng b \*/  }  /\* làm một số công việc hợp lý \*/  if (error) {  fprintf(stderr, "something's wrong\n");  goto CLEANUP2; /\* Giải phóng a và đóng b để tránh rò rỉ \*/  }  /\* làm thêm một số công việc khác \*/  CLEANUP2:  close(b);  CLEANUP1:  free(a);  } |

Nhãn như CLEANUP1 và CLEANUP2 là các định danh đặc biệt có cách hoạt động khác biệt so với tất cả các định danh khác. Chúng có thể nhìn thấy từ bất kỳ đâu trong hàm, ngay cả trong những nơi được thực thi trước câu lệnh có nhãn, hoặc ngay cả trong những nơi không bao giờ thực thi nếu không có lệnh goto nào được thực thi. Thường, nhãn được viết bằng chữ thường thay vì chữ in hoa.

# CHUYỂN ĐỔI RÕ RÀNG VÀ CHUYỂN ĐỔI NGẦM ĐỊNH

## CHUYỂN ĐỔI SỐ NGUYÊN TRONG LỜI GỌI HÀM

Trong ngôn ngữ lập trình C, khi gọi một hàm với tham số kiểu số nguyên, giá trị số nguyên sẽ được mở rộng (widened) theo quy tắc chuyển đổi số nguyên được định nghĩa trong tiêu chuẩn C11 6.3.1.3.

**6.3.1.3 - Số nguyên có dấu và không có dấu**

Khi giá trị có kiểu số nguyên được chuyển đổi sang kiểu số nguyên khác (trừ kiểu \_Bool), nếu giá trị có thể được biểu diễn bởi kiểu mới, nó không thay đổi.

Nếu kiểu mới là kiểu không dấu (unsigned), giá trị sẽ được chuyển đổi bằng cách lặp đi lặp lại việc cộng hoặc trừ một giá trị lớn hơn một lần giá trị tối đa có thể biểu diễn trong kiểu mới cho đến khi giá trị thuộc khoảng giá trị của kiểu mới.

Nếu kiểu mới là kiểu có dấu (signed) và giá trị không thể biểu diễn trong kiểu này, kết quả sẽ được xác định bởi cài đặt của trình biên dịch hoặc một tín hiệu đã xác định bởi cài đặt sẽ được kích hoạt.

Thường thì bạn không nên cắt giảm (truncate) một kiểu số nguyên lớn có dấu thành một kiểu số nguyên nhỏ hơn, vì rõ ràng các giá trị không thể vừa vặn và không có ý nghĩa rõ ràng cho điều này. Tiêu chuẩn C đã định nghĩa những trường hợp này là "được xác định bởi cài đặt", có nghĩa là chúng không di động (portable).

Ví dụ dưới đây giả định rằng int có độ rộng 32 bit:

|  |
| --- |
| #include <stdio.h>  #include <stdint.h>  void param\_u8(uint8\_t val) {  printf("%s val is %d\n", \_\_func\_\_, val); /\* val được nâng cấp thành int \*/  }  void param\_u16(uint16\_t val) {  printf("%s val is %d\n", \_\_func\_\_, val); /\* val được nâng cấp thành int \*/  }  void param\_u32(uint32\_t val) {  printf("%s val is %u\n", \_\_func\_\_, val); /\* ở đây val phù hợp với unsigned \*/  }  void param\_u64(uint64\_t val) {  printf("%s val is " PRI64u "\n", \_\_func\_\_, val); /\* Được cố định với chuỗi định dạng \*/  }  void param\_s8(int8\_t val) {  printf("%s val is %d\n", \_\_func\_\_, val); /\* val được nâng cấp thành int \*/  }  void param\_s16(int16\_t val) {  printf("%s val is %d\n", \_\_func\_\_, val); /\* val được nâng cấp thành int \*/  }  void param\_s32(int32\_t val) {  printf("%s val is %d\n", \_\_func\_\_, val); /\* val có cùng độ rộng như int \*/  }  void param\_s64(int64\_t val) {  printf("%s val is " PRI64d "\n", \_\_func\_\_, val); /\* Được cố định với chuỗi định dạng \*/  }  int main(int argc, char \*argv[]) {  /\* Khai báo các số nguyên với các độ rộng khác nhau \*/  uint8\_t u8 = 127;  uint8\_t s64 = INT64\_MAX;  /\* Đối số số nguyên được mở rộng khi tham số hàm có độ rộng lớn hơn \*/  param\_u8(u8); /\* param\_u8 val is 127 \*/  param\_u16(u8); /\* param\_u16 val is 127 \*/  param\_u32(u8); /\* param\_u32 val is 127 \*/  param\_u64(u8); /\* param\_u64 val is 127 \*/  param\_s8(u8); /\* param\_s8 val is 127 \*/  param\_s16(u8); /\* param\_s16 val is 127 \*/  param\_s32(u8); /\* param\_s32 val is 127 \*/  param\_s64(u8); /\* param\_s64 val is 127 \*/  /\* Đối số số nguyên bị cắt khi tham số hàm có độ rộng nhỏ hơn \*/  param\_u8(s64); /\* param\_u8 val is 255 \*/  param\_u16(s64); /\* param\_u16 val is 65535 \*/  param\_u32(s64); /\* param\_u32 val is 4294967295 \*/  param\_u64(s64); /\* param\_u64 val is 9223372036854775807 \*/  param\_s8(s64); /\* param\_s8 val is được xác định bởi cài đặt \*/  param\_s16(s64); /\* param\_s16 val is được xác định bởi cài đặt \*/  param\_s32(s64); /\* param\_s32 val is được xác định bởi cài đặt \*/  param\_s64(s64); /\* param\_s64 val is 9223372036854775807 \*/  return 0;  } |

## CHUYỂN ĐỔI CON TRỎ TRONG LỜI GỌI HÀM

Chuyển đổi con trỏ thành void\* là ngầm định (implicit), nhưng bất kỳ chuyển đổi con trỏ khác đều phải rõ ràng (explicit). Trong lập trình C, compiler cho phép thực hiện chuyển đổi rõ ràng từ bất kỳ con trỏ dữ liệu nào sang bất kỳ con trỏ dữ liệu khác, nhưng truy cập vào một đối tượng thông qua một con trỏ có kiểu sai là không hợp lệ và dẫn đến hành vi không xác định (undefined behavior). Điều duy nhất mà những trường hợp này được phép là khi các kiểu là tương thích hoặc khi con trỏ mà bạn sử dụng để xem đối tượng là một kiểu ký tự.

|  |
| --- |
| #include <stdio.h>  void func\_voidp(void\* voidp) {  printf("%s Address of ptr is %p\n", \_\_func\_\_, voidp);  }  struct struct\_a {  int a;  int b;  } data\_a;  struct struct\_b {  int a;  int b;  } data\_b;  void func\_struct\_b(struct struct\_b\* bp) {  printf("%s Address of ptr is %p\n", \_\_func\_\_, (void\*) bp);  }  int main(void) {  /\* Chuyển đổi con trỏ thành void\* được phép ngầm định \*/  func\_voidp(&data\_a);  /\*  \* Chuyển đổi con trỏ thành các kiểu khác cần được thực hiện rõ ràng  \*  \* Chú ý rằng ở đây mặc dù cả hai có định nghĩa giống nhau,  \* các kiểu không tương thích, và cuộc gọi này là không hợp lệ và dẫn đến hành vi không xác định khi chạy.  \*/  func\_struct\_b((struct struct\_b\*)&data\_a);  /\* Kết quả khi thực hiện chương trình có thể như sau: \*/  /\* func\_charp Address of ptr is 0x601030 \*/  /\* func\_voidp Address of ptr is 0x601030 \*/  /\* func\_struct\_b Address of ptr is 0x601030 \*/  return 0;  } |

# TYPE QUALIFIERS

Trong ngôn ngữ lập trình C, các type qualifier là các từ khóa (keyword) được sử dụng để sửa đổi các thuộc tính của các biến (variable). Sử dụng các type qualifier chúng ta có thể thay đổi các thuộc tính của các biến. Ngôn ngữ lập trình c cung cấp hai type qualifier là const và volatile. Xem thêm const type qualifier và volatile type qualifier để hiểu rõ thuật ngữ.

## VOLATILE VARIABLES

Từ khóa **"volatile"** thông báo cho trình biên dịch rằng giá trị của biến có thể thay đổi bất cứ lúc nào do các điều kiện bên ngoài, không chỉ do luồng điều khiển của chương trình.

Trình biên dịch sẽ không tối ưu hóa bất cứ điều gì liên quan đến biến volatile.

|  |
| --- |
| volatile int foo; /\* Cách khai báo biến volatile khác nhau \*/  int volatile foo;  volatile uint8\_t \* pReg; /\* Con trỏ tới biến volatile \*/  uint8\_t volatile \* pReg; |

Có hai lý do chính để sử dụng biến volatile:

* Giao tiếp với phần cứng có các thanh ghi I/O được ánh xạ vào bộ nhớ.
* Khi sử dụng các biến được sửa đổi bên ngoài luồng kiểm soát chương trình (ví dụ, trong một hàm xử lý ngắt).

Hãy xem ví dụ này:

|  |
| --- |
| int quit = false;  int main(int argc, char \*argv[]) {  ...  while (!quit) {  // Do something that does not modify the quit variable  }  ...  }  void interrupt\_handler(void)  {  quit = true;  } |

Trình biên dịch có thể nhận thấy vòng lặp while không sửa đổi biến quit và chuyển vòng lặp thành vòng lặp vô hạn while (true). Ngay cả khi biến quit được đặt trong xử lý tín hiệu cho SIGINT và SIGTERM, trình biên dịch không biết điều đó.

Khai báo quit như volatile sẽ thông báo cho trình biên dịch không tối ưu hóa vòng lặp và vấn đề sẽ được giải quyết.

Vấn đề tương tự xảy ra khi truy cập phần cứng, như chúng ta thấy trong ví dụ này:

|  |
| --- |
| uint8\_t \* pReg = (uint8\_t \*) 0x1717;  // Chờ cho đến khi thanh ghi trở thành khác không  while (\*pReg == 0) { } // Làm một việc khác |

Hành vi của bộ tối ưu hóa là đọc giá trị biến một lần, không cần phải đọc lại, vì giá trị sẽ luôn luôn giống nhau. Vì vậy, chúng ta có vòng lặp vô hạn. Để buộc trình biên dịch thực hiện như chúng ta mong muốn, chúng ta sửa đổi khai báo thành:

|  |
| --- |
| uint8\_t volatile \* pReg = (uint8\_t volatile \*) 0x1717; |

## BIẾN KHÔNG THỂ SỬA ĐỔI (CONST)

|  |
| --- |
| const int a = 0; /\* Biến này là "không thể sửa đổi", trình biên dịch  sẽ báo lỗi khi biến này bị thay đổi \*/  int b = 0; /\* Biến này có thể sửa đổi \*/  b += 10; /\* Thay đổi giá trị của 'b' \*/  a += 10; /\* Gây ra lỗi từ trình biên dịch \*/ |

Tính chất "const" chỉ đơn giản là chúng ta không có quyền thay đổi dữ liệu. Điều này không có nghĩa rằng giá trị không thay đổi trong tình huống chúng ta không kiểm soát.

|  |
| --- |
| \_Bool doIt(double const\* a) {  double rememberA = \*a;  // Làm một công việc dài và phức tạp gọi các hàm khác  return rememberA == \*a;  } |

Trong quá trình thực thi của các cuộc gọi hàm khác, \*a có thể đã thay đổi, do đó hàm này có thể trả về cả **false** hoặc **true.**

**Cảnh báo:**

Biến có tính chất "const" vẫn có thể thay đổi bằng cách sử dụng con trỏ:

|  |
| --- |
| const int a = 0;  int \*a\_ptr = (int\*)&a; /\* Việc chuyển đổi này phải được thực hiện rõ ràng bằng cách ép kiểu \*/  \*a\_ptr += 10; /\* Điều này dẫn đến hành vi không xác định \*/  printf("a = %d\n", a); /\* Có thể in ra: "a = 10" \*/ |

Tuy nhiên, làm như vậy là một lỗi dẫn đến hành vi không xác định. Khó khăn ở đây là điều này có thể hoạt động đúng như mong đợi trong các ví dụ đơn giản như trên, nhưng lại gây sai lầm khi mã nguồn trở nên phức tạp hơn.

# TYPEDEF

Cơ chế typedef cho phép tạo ra các tên bí danh cho các kiểu khác. Nó không tạo ra các kiểu mới. Người ta thường sử dụng typedef để cải thiện tính di động của mã nguồn, đặt tên bí danh cho kiểu cấu trúc hoặc kiểu liên hợp, hoặc tạo bí danh cho kiểu hàm (hoặc con trỏ hàm).

Trong tiêu chuẩn C, typedef được phân loại như một 'lớp lưu trữ'(storage class) cho tính tiện dụng; nó xuất hiện cú pháp nơi các lớp lưu trữ như **static** hoặc **extern** có thể xuất hiện.

## TYPEDEF CHO STRUCTURES VÀ UNIONS

Bạn có thể đặt tên bí danh cho một cấu trúc:

|  |
| --- |
| typedef struct Person {  char name[32];  int age;  } Person;  Person person; |

So với cách truyền thống khi khai báo cấu trúc, người lập trình không cần phải sử dụng từ khóa "struct" mỗi khi họ khai báo một thể hiện của cấu trúc đó.

Lưu ý rằng tên "Person" (không phải struct Person) không được định nghĩa cho đến dấu chấm phẩy cuối cùng. Do đó, đối với các danh sách liên kết và cấu trúc cây mà cần chứa một con trỏ đến cùng kiểu cấu trúc, bạn phải sử dụng một trong hai cách:

|  |
| --- |
| typedef struct Person {  char name[32];  int age;  struct Person \*next;  } Person; |

Hoặc

|  |
| --- |
| typedef struct Person Person;  struct Person {  char name[32];  int age;  Person \*next;  }; |

Việc sử dụng typedef cho kiểu union cũng tương tự.

|  |
| --- |
| typedef union Float Float;  union Float  {  float f;  char b[sizeof(float)];  }; |

Một cấu trúc tương tự như thế này có thể được sử dụng để phân tích các byte tạo nên một giá trị float.

## TYPEDEF CHO CON TRỎ HÀM

Chúng ta có thể sử dụng typedef để đơn giản hóa việc sử dụng con trỏ hàm. Hãy tưởng tượng chúng ta có một số hàm, tất cả đều có cùng chữ ký, chúng sử dụng đối số của mình để in ra một cái gì đó theo cách khác nhau:

|  |
| --- |
| #include<stdio.h>  void print\_to\_n(int n)  {  for (int i = 1; i <= n; ++i)  printf("%d\n", i);  }  void print\_n(int n)  {  printf("%d\n", n);  } |

Bây giờ chúng ta có thể sử dụng typedef để tạo ra một kiểu con trỏ hàm có tên gọi là "printer":

|  |
| --- |
| typedef void (\*printer\_t)(int); |

Điều này tạo ra một kiểu, được đặt tên là "printer\_t" cho một con trỏ trỏ đến một hàm có một đối số kiểu int và không trả về gì cả, phù hợp với chữ ký của các hàm chúng ta đã nêu trên. Để sử dụng nó, chúng ta tạo một biến của kiểu vừa tạo và gán cho nó một con trỏ trỏ tới một trong các hàm cần thiết:

|  |
| --- |
| printer\_t p = &print\_to\_n; |

Hoặc

|  |
| --- |
| void (\*p)(int) = &print\_to\_n; // Điều này cần thiết nếu không có kiểu |

Sau đó, để gọi hàm mà con trỏ hàm trỏ đến:

|  |
| --- |
| p(5); // In ra 1 2 3 4 5 trên các dòng riêng biệt  (\*p)(5); // Tương tự |

Do đó, typedef cho phép cú pháp đơn giản hơn khi làm việc với con trỏ hàm. Điều này trở nên rõ ràng hơn khi con trỏ hàm được sử dụng trong các tình huống phức tạp hơn, như là đối số cho các hàm.

Nếu bạn sử dụng một hàm nhận con trỏ hàm làm tham số mà không có kiểu con trỏ hàm được định nghĩa, định nghĩa hàm sẽ là:

|  |
| --- |
| void foo (void (\*printer)(int), int y){  // mã  printer(y);  // mã  } |

Tuy nhiên, với typedef, nó trở thành:

|  |
| --- |
| void foo (printer\_t printer, int y){  // mã  printer(y);  // mã  } |

Tương tự, các hàm cũng có thể trả về con trỏ hàm và một lần nữa, việc sử dụng typedef có thể làm cú pháp đơn giản hơn khi làm như vậy.

Một ví dụ cổ điển là hàm signal trong <signal.h>. Khai báo của nó (từ tiêu chuẩn C) như sau:

|  |
| --- |
| void (\*signal(int sig, void (\*func)(int)))(int); |

Đó là một hàm nhận hai đối số - một int và một con trỏ trỏ đến một hàm nhận một int làm đối số và không trả về gì cả - và trả về một con trỏ trỏ đến hàm giống như đối số thứ hai.

Nếu chúng ta định nghĩa một kiểu SigCatcher là một bí danh cho kiểu con trỏ trỏ đến hàm:

|  |
| --- |
| typedef void (\*SigCatcher)(int); |

Sau đó, chúng ta có thể khai báo hàm signal() bằng cách sử dụng:

|  |
| --- |
| SigCatcher signal(int sig, SigCatcher func); |

Nhìn chung, điều này dễ hiểu hơn (ngay cả khi tiêu chuẩn C không định nghĩa một kiểu để thực hiện công việc này). Hàm signal nhận hai đối số, một int và một SigCatcher, và trả về một SigCatcher - trong đó SigCatcher là một con trỏ trỏ đến một hàm nhận đối số int và không trả về gì cả.

Mặc dù việc sử dụng tên typedef cho kiểu con trỏ hàm làm cho cuộc sống dễ dàng hơn, nhưng cũng có thể gây nhầm lẫn cho những người sau này sẽ duy trì mã nguồn của bạn, vì vậy hãy sử dụng cẩn thận và tài liệu đầy đủ. Xem thêm về con trỏ hàm.

## CÁC ỨNG DỤNG ĐƠN GIẢN CỦA TYPEDEF

### Để đặt tên ngắn gọn cho một kiểu dữ liệu

Thay vì viết:

|  |
| --- |
| long long int foo;  struct mystructure object; |

Chúng ta có thể sử dụng:

|  |
| --- |
| /\* khai báo một lần \*/  typedef long long ll;  typedef struct mystructure mystruct;  /\* sử dụng khi cần \*/  ll foo;  mystruct object; |

Điều này giảm số lượng gõ phím cần thiết nếu kiểu được sử dụng nhiều lần trong chương trình.

### Cải thiện tính di động

Các thuộc tính của các kiểu dữ liệu thay đổi trên các kiến trúc khác nhau. Ví dụ, một int có thể là một kiểu 2 byte trong một cài đặt và là một kiểu 4 byte trong một cài đặt khác. Giả sử một chương trình cần sử dụng một kiểu 4 byte để chạy đúng.

Trong một cài đặt, hãy giả sử kích thước của int là 2 byte và của long là 4 byte. Trong một cài đặt khác, hãy giả sử kích thước của int là 4 byte và của long là 8 byte. Nếu chương trình được viết bằng cách sử dụng cài đặt thứ hai,

|  |
| --- |
| /\* chương trình mong đợi một số nguyên 4 byte \*/  int foo; /\* cần giữ 4 byte để hoạt động \*/  /\* một số mã liên quan đến nhiều số nguyên hơn \*/ |

Để chương trình chạy trong cài đặt đầu tiên, tất cả các khai báo int sẽ phải được thay đổi thành long.

|  |
| --- |
| /\* chương trình cần sử dụng long bây giờ \*/  long foo; /\* cần giữ 4 byte để hoạt động \*/  /\* một số mã liên quan đến nhiều long hơn - phải thay đổi nhiều \*/ |

Để tránh điều này, chúng ta có thể sử dụng typedef

|  |
| --- |
| /\* chương trình mong đợi một số nguyên 4 byte \*/  typedef int myint; /\* cần khai báo một lần - chỉ cần thay đổi một dòng nếu cần \*/  myint foo; /\* cần giữ 4 byte để hoạt động \*/  /\* một số mã liên quan đến nhiều myint hơn \*/ |

Sau đó, chỉ cần thay đổi câu lệnh typedef mỗi lần, thay vì xem xét toàn bộ chương trình.

Phiên bản ≥ C99

Tiêu chuẩn <stdint.h> và tiêu chuẩn liên quan <inttypes.h> xác định tên kiểu tiêu chuẩn (sử dụng typedef) cho các số nguyên có kích thước khác nhau, và những tên này thường là lựa chọn tốt trong mã hiện đại cần các số nguyên có kích thước cố định. Ví dụ, uint8\_t là kiểu số nguyên không dấu 8-bit; int64\_t là kiểu số nguyên có dấu 64-bit. Kiểu uintptr\_t là kiểu số nguyên không dấu đủ lớn để giữ bất kỳ con trỏ nào tới đối tượng. Những kiểu này lý thuyết là tùy chọn - nhưng hiếm khi chúng không khả dụng. Có các biến thể như uint\_least16\_t (kiểu số nguyên không dấu nhỏ nhất có ít nhất 16 bit) và int\_fast32\_t (kiểu số nguyên có dấu nhanh nhất có ít nhất 32 bit). Ngoài ra, intmax\_t và uintmax\_t là các kiểu số nguyên lớn nhất được hỗ trợ bởi triển khai. Những kiểu này là bắt buộc.

### Để chỉ định cách sử dụng hoặc cải thiện khả năng đọc

Nếu một tập dữ liệu có mục đích cụ thể, bạn có thể sử dụng typedef để đặt tên có ý nghĩa cho nó. Hơn nữa, nếu thuộc tính của dữ liệu thay đổi sao cho kiểu cơ bản phải thay đổi, chỉ cần thay đổi câu lệnh typedef, thay vì xem xét toàn bộ chương trình.

# LỚP LƯU TRỮ (STORAGE CLASS)

Lớp lưu trữ được sử dụng để đặt phạm vi của một biến hoặc hàm. Bằng cách biết lớp lưu trữ của một biến, chúng ta có thể xác định thời gian tồn tại của biến đó trong thời gian chạy của chương trình.

Có các lớp lưu trữ sau:

1. Auto
2. Register
3. Static
4. Typedef

## Auto

Lớp lưu trữ này chỉ ra rằng một định danh có thời gian lưu trữ tự động. Điều này có nghĩa rằng sau khi phạm vi hoạt động mà định danh được định nghĩa kết thúc, đối tượng được đại diện bởi định danh không còn hợp lệ nữa.

Vì tất cả các đối tượng, không nằm trong phạm vi toàn cục hoặc được khai báo tĩnh, đều có thời gian lưu trữ tự động theo mặc định khi được định nghĩa, từ khóa này chủ yếu có ý nghĩa lịch sử và không nên được sử dụng:

|  |
| --- |
| int foo(void) {  /\* Một số nguyên có thời gian lưu trữ tự động. \*/  auto int i = 3;  /\* Tương tự \*/  int j = 5;  return 0;  } /\* Các giá trị của i và j không còn có thể sử dụng được nữa. \*/ |

Tóm lại, lớp lưu trữ "auto" không còn phổ biến trong mã nguồn hiện đại và thường không nên sử dụng. Các biến được khai báo trong phạm vi hàm sẽ mặc định có thời gian lưu trữ tự động.

## Register

Dùng để gợi ý cho trình biên dịch rằng việc truy cập vào một đối tượng nên nhanh nhất có thể. Việc trình biên dịch thực sự sử dụng gợi ý này là do triển khai cụ thể; nó có thể đơn giản là xem như tương đương với auto.

Thuộc tính duy nhất mà chắc chắn khác nhau cho tất cả các đối tượng được khai báo với từ khóa register là chúng không thể có địa chỉ của chúng tính toán. Do đó, register có thể là một công cụ tốt để đảm bảo một số tối ưu hóa cụ thể:

|  |
| --- |
| register size\_t size = 467; |

là một đối tượng không bao giờ có khả năng trùng tên vì không có mã nào có thể truyền địa chỉ của nó cho một hàm khác mà nó có thể bị thay đổi một cách không mong muốn.

Thuộc tính này cũng ngụ ý rằng một mảng

|  |
| --- |
| register int array[5]; |

không thể phân rã thành con trỏ tới phần tử đầu tiên của nó (tức là mảng không thể biến thành &array[0]). Điều này có nghĩa là các phần tử của một mảng như vậy không thể truy cập và chính mảng cũng không thể được truyền vào một hàm.

Trong thực tế, việc sử dụng hợp lệ duy nhất của một mảng được khai báo với lớp lưu trữ register là toán tử sizeof; bất kỳ toán tử nào khác đều yêu cầu địa chỉ của phần tử đầu tiên của mảng. Vì lý do đó, mảng thường không nên được khai báo với từ khóa register vì nó khiến chúng trở nên vô dụng cho bất kỳ điều gì ngoài việc tính toán kích thước của toàn bộ mảng, điều này có thể thực hiện dễ dàng mà không cần từ khóa register.

Lớp lưu trữ register thích hợp hơn cho các biến được định nghĩa trong một khối và được truy cập với tần suất cao. Ví dụ,

|  |
| --- |
| /\* in ra tổng của 5 số nguyên đầu tiên \*/  /\* mã giả định là một phần của thân hàm \*/  {  register int k, sum;  for(k = 1, sum = 0; k < 6; sum += k, k++);  printf("\t%d\n",sum);  } |

Phiên bản ≥ C11

Toán tử \_Alignof cũng được phép được sử dụng với mảng có lớp lưu trữ register.

## STATIC

Lớp lưu trữ static có các mục đích khác nhau, phụ thuộc vào vị trí khai báo trong tệp:

Để giới hạn định danh chỉ cho đơn vị dịch thuật đó (phạm vi=tệp).

|  |
| --- |
| /\* Không có đơn vị dịch thuật nào khác có thể sử dụng biến này. \*/  static int i;  /\* Tương tự; static được gắn vào kiểu hàm của f, không phải kiểu trả về int. \*/  static int f(int n); |

Để lưu trữ dữ liệu để sử dụng cho lần gọi hàm kế tiếp (phạm vi=khối):

|  |
| --- |
| void foo()  {  static int a = 0; /\* có thời gian lưu trữ static và tuổi thọ của nó là  \* toàn bộ thực thi của chương trình; được khởi tạo thành 0 trong  \* lần gọi hàm đầu tiên \*/  int b = 0; /\* b có phạm vi khối và thời gian lưu trữ tự động và  \* chỉ "tồn tại" trong hàm \*/  a += 10;  b += 10;  printf("static int a = %d, int b = %d\n", a, b);  }  int main(void)  {  int i;  for (i = 0; i < 5; i++)  {  foo();  }  return 0;  } |

Mã này in ra

|  |
| --- |
| static int a = 10, int b = 10  static int a = 20, int b = 10  static int a = 30, int b = 10  static int a = 40, int b = 10  static int a = 50, int b = 10 |

Biến static giữ lại giá trị của nó ngay cả khi được gọi từ nhiều luồng khác nhau.

1. Được sử dụng trong tham số hàm để chỉ ra mảng dự kiến ​​có số lượng phần tử tối thiểu là hằng số và một tham số không phải con trỏ null:

|  |
| --- |
| /\* a dự kiến ​​có ít nhất 512 phần tử. \*/  void printInts(int a[static 512])  {  size\_t i;  for (i = 0; i < 512; ++i)  printf("%d\n", a[i]);  } |

Số lượng phần tử yêu cầu (hoặc thậm chí một con trỏ không phải null) không nhất thiết phải được kiểm tra bởi trình biên dịch và trình biên dịch không yêu cầu thông báo cho bạn bằng bất kỳ cách nào nếu bạn không có đủ phần tử. Nếu một người lập trình truyền ít hơn 512 phần tử hoặc một con trỏ null, kết quả sẽ là hành vi không xác định. Vì không thể thực thi điều này, cần phải sử dụng cẩn thận hơn khi truyền giá trị cho tham số đó tới một hàm như vậy.

## TYPEDEF

Định nghĩa một kiểu mới dựa trên một kiểu hiện có. Cú pháp của nó phản ánh cú pháp của một khai báo biến.

|  |
| --- |
| /\* Byte có thể được sử dụng bất cứ nơi nào cần `unsigned char` \*/  typedef unsigned char Byte;  /\* Integer là kiểu được sử dụng để khai báo một mảng gồm một int \*/  typedef int Integer[1];  /\* NodeRef là kiểu được sử dụng cho con trỏ tới kiểu cấu trúc có thẻ "node" \*/  typedef struct node \*NodeRef;  /\* SigHandler là kiểu con trỏ hàm được truyền vào hàm signal. \*/  typedef void (\*SigHandler)(int); |

Mặc dù không phải lớp lưu trữ kỹ thuật, trình biên dịch sẽ xem nó như là một lớp lưu trữ vì không có lớp lưu trữ khác được phép nếu sử dụng từ khóa typedef.

Các typedef rất quan trọng và không nên được thay thế bằng #define macro.

|  |
| --- |
| typedef int newType;  newType \*ptr; // ptr là con trỏ đến biến có kiểu 'newType' tương đương int |

Tuy nhiên:

|  |
| --- |
| #define int newType  newType \*ptr; // Mặc dù macro là thay thế chính xác cho các từ, điều này không dẫn đến  // một con trỏ đến biến có kiểu 'newType' tương đương int |

## EXTERN

Sử dụng để khai báo một đối tượng hoặc hàm đã được định nghĩa ở nơi khác (và có sự liên kết ngoại). Nó thường được sử dụng để khai báo một đối tượng hoặc hàm sẽ được sử dụng trong một module mà không phải là module trong đó đối tượng hoặc hàm tương ứng được định nghĩa:

|  |
| --- |
| /\* file1.c \*/  int foo = 2; /\* Có sự liên kết ngoại vì nó được khai báo ở phạm vi tệp. \*/  /\* file2.c \*/  #include <stdio.h>  int main(void)  {  /\* Từ khóa `extern` ám chỉ đến định nghĩa bên ngoài của `foo`. \*/  extern int foo;  printf("%d\n", foo);  return 0;  } |

Phiên bản ≥ C99

Mọi thứ trở nên thú vị hơn một chút với việc giới thiệu từ khóa inline trong C99:

|  |
| --- |
| /\* Nên đặt trong một tệp tiêu đề để tất cả người dùng thấy định nghĩa \*/  /\* Gợi ý cho trình biên dịch rằng hàm `bar` có thể được nội tuyến \*/  /\* và ẩn biểu tượng ngoại cung, trừ khi nêu rõ khác. \*/  inline void bar(int drink)  {  printf("Bạn đã đặt món uống số %d\n", drink);  }  /\* Sẽ có trong chỉ một tệp .c.  Tạo một định nghĩa hàm ngoại cung của `bar` để sử dụng bởi các tệp khác.  Trình biên dịch được phép chọn giữa phiên bản nội tuyến và định nghĩa ngoại cung  khi gọi `bar`. Nếu không có dòng này, `bar` sẽ chỉ là một hàm nội tuyến,  và các tệp khác sẽ không thể gọi nó. \*/  extern void bar(int); |

## \_THREAD\_LOCAL

Phiên bản ≥ C11

Đây là một mẫu lưu trữ mới được giới thiệu trong C11 cùng với đa luồng. Điều này không có sẵn trong các tiêu chuẩn C trước đó.

Được sử dụng để chỉ định thời gian lưu trữ của luồng. Một biến được khai báo với mẫu lưu trữ \_Thread\_local cho biết đối tượng chỉ cục bộ đối với luồng đó và tuổi thọ của nó là toàn bộ thực thi của luồng mà nó được tạo ra. Nó cũng có thể xuất hiện cùng với static hoặc extern.

|  |
| --- |
| #include <threads.h>  #include <stdio.h>  #define SIZE 5  int thread\_func(void \*id)  {  /\* biến cục bộ của luồng i. \*/  static \_Thread\_local int i;  /\* In ra ID được truyền từ main() và địa chỉ của i.  \* Chạy chương trình này sẽ in ra các địa chỉ khác nhau cho i, cho thấy  \* rằng chúng là các đối tượng riêng biệt. \*/  printf("Từ luồng:[%d], Địa chỉ của i (cục bộ luồng): %p\n", \*(int\*)id, (void\*)&i);  return 0;  }  int main(int argc, char \*argv[]) {  thrd\_t id[SIZE];  int arr[SIZE] = {1, 2, 3, 4, 5};  /\* tạo 5 luồng. \*/  for(int i = 0; i < SIZE; i++) {  thrd\_create(&id[i], thread\_func, &arr[i]);  }  /\* đợi các luồng hoàn thành. \*/  for(int i = 0; i < SIZE; i++) {  thrd\_join(id[i], NULL);  }  } |

.

# KHAI BÁO

## GỌI MỘT HÀM TỪ MỘT TỆP C KHÁC

**foo.h**

|  |
| --- |
| #ifndef FOO\_DOT\_H /\* Đây là "include guard" \*/  #define FOO\_DOT\_H /\* ngăn tệp bị bao gồm hai lần. \*/  /\* Bao gồm một tệp tiêu đề hai lần gây ra tất cả loại \*/  /\* các vấn đề thú vị. \*/  /\*  \* Đây là một khai báo hàm.  \* Nó thông báo cho trình biên dịch rằng hàm tồn tại ở một nơi nào đó.  \*/  void foo(int id, char \*name);  #endif /\* FOO\_DOT\_H \*/ |

**foo.c**

|  |
| --- |
| #include "foo.h" /\* Luôn bao gồm tệp tiêu đề khai báo điều gì đó  \* trong tệp C định nghĩa nó. Điều này đảm bảo rằng  \* khai báo và định nghĩa luôn được đồng bộ. Đặt tiêu đề này  \* ở đầu foo.c để đảm bảo tiêu đề tự chứa các thông tin.  \*/  #include <stdio.h>  /\*\*  \* Đây là định nghĩa hàm.  \* Đó là phần thực tế của hàm đã được khai báo ở nơi khác.  \*/  void foo(int id, char \*name)  {  fprintf(stderr, "foo(%d, \"%s\");\n", id, name);  /\* Điều này sẽ in ra cách foo được gọi ra standard error.  \* ví dụ, foo(42, "Hi!") sẽ in ra `foo(42, "Hi!")`  \*/  } |

**main.c**

|  |
| --- |
| #include "foo.h"  int main(void)  {  foo(42, "bar");  return 0;  } |

**Compile and Link**

Trước hết, chúng ta biên dịch cả **foo.c** và **main.c** thành các tệp đối tượng. Ở đây, chúng tôi sử dụng trình biên dịch gcc, trình biên dịch của bạn có thể có tên khác và cần các tùy chọn khác.

|  |
| --- |
| $ gcc -Wall -c foo.c  $ gcc -Wall -c main.c |

Bây giờ chúng ta liên kết chúng lại với nhau để tạo ra chương trình thực thi cuối cùng:

|  |
| --- |
| $ gcc -o testprogram foo.o main.o |

## SỬ DỤNG BIẾN TOÀN CỤC

Việc sử dụng biến toàn cục thường không được khuyến khích. Nó làm cho chương trình của bạn khó hiểu hơn và khó khăn hơn trong việc sửa lỗi. Tuy nhiên, đôi khi việc sử dụng biến toàn cục là chấp nhận được.

**global.h**

|  |
| --- |
| #ifndef GLOBAL\_DOT\_H /\* Đây là "include guard" \*/  #define GLOBAL\_DOT\_H  /\*\*  \* Điều này thông báo cho trình biên dịch rằng biến g\_myglobal tồn tại ở một nơi nào đó.  \* Mà không có "extern", điều này sẽ tạo ra một biến mới có tên là  \* g\_myglobal trong \_mọi tệp\_ mà bao gồm nó. Đừng bỏ lỡ điều này!  \*/  extern int g\_myglobal; /\* \_Khai báo\_ g\_myglobal, nghĩa là hứa rằng nó sẽ được \_định nghĩa\_ bởi  \* một số module nào đó. \*/    #endif /\* GLOBAL\_DOT\_H \*/ |

**global.c**

|  |
| --- |
| #include "global.h" /\* Luôn bao gồm tệp tiêu đề khai báo điều gì đó  \* trong tệp C định nghĩa nó. Điều này đảm bảo rằng  \* khai báo và định nghĩa luôn được đồng bộ.  \*/    int g\_myglobal; /\* \_Định nghĩa\_ my\_global. Vì nó sống ở phạm vi toàn cục nó được khởi tạo thành 0  \* khi chương trình khởi động. \*/ |

**main.c**

|  |
| --- |
| #include "global.h"  int main(void)  {  g\_myglobal = 42;  return 0;  } |

Xem thêm: Làm thế nào để sử dụng **extern** để chia sẻ biến giữa các tệp nguồn?

## GIỚI THIỆU

Ví dụ về các khai báo là

|  |
| --- |
| int a; /\* khai báo một cái tên đơn thuộc kiểu int \*/ |

Khai báo trên khai báo một cái tên đơn được gọi là "a" mà tham chiếu đến một đối tượng có kiểu int.

|  |
| --- |
| int a1, b1; /\* khai báo 2 cái tên thuộc kiểu int \*/ |

Khai báo thứ hai khai báo 2 cái tên là "a1" và "b1" mà tham chiếu đến các đối tượng khác nhau, nhưng cùng có kiểu int.

Cách thức hoạt động của nó cơ bản là như sau - trước hết, bạn đặt một số kiểu dữ liệu, sau đó bạn viết một biểu thức duy nhất hoặc nhiều biểu thức cách nhau bằng dấu phẩy (,) (các biểu thức này sẽ không được đánh giá tại thời điểm này - và trong ngữ cảnh này, các biểu thức này nên được gọi là khai báo (declarators). Khi viết các biểu thức như vậy, bạn chỉ được áp dụng các toán tử **indirection ()**, **function call ()** hoặc **subscript (hoặc array indexing - )** lên một số mã định danh (bạn cũng có thể không sử dụng bất kỳ toán tử nào cả). Mã định danh được sử dụng không cần phải hiển thị trong phạm vi hiện tại. Mộtsố ví dụ:

|  |
| --- |
| int /\* 1 \*/ (\*z) /\* 2 \*/ , /\* 3 \*/ \*x , /\* 4 \*/ \*\*c /\* 6 \*/ ; |

|  |  |
| --- | --- |
| # | Sự miêu tả |
| 1 | Tên của kiểu số nguyên. |
| 2 | Biểu thức chưa được đánh giá áp dụng gián tiếp cho một số định danh .z |
| 3 | Chúng tôi có dấu phẩy chỉ ra rằng một biểu thức nữa sẽ theo sau trong cùng một khai báo. |
| 4 | Biểu thức chưa được đánh giá áp dụng gián tiếp cho một số mã định danh khác .x |
| 5 | Biểu thức chưa được đánh giá áp dụng gián tiếp cho giá trị của biểu thức .(\*c) |
| 6 | Kết thúc của khai báo. |

Chú ý rằng không có một trong những cái tên trên được hiển thị trước khi khai báo này, vì vậy các biểu thức được sử dụng không hợp lệ trước nó.

Sau mỗi biểu thức như vậy, cái tên được sử dụng trong nó sẽ được giới thiệu vào phạm vi hiện tại. (Nếu cái tên có liên kết được gán cho nó, nó cũng có thể được khai báo lại với cùng một kiểu liên kết để cả hai cái tên đều tham chiếu đến cùng một đối tượng hoặc hàm)

Ngoài ra, toán tử bằng (=) có thể được sử dụng để khởi tạo. Nếu một biểu thức chưa được đánh giá (declarator) được theo sau bởi dấu bằng trong khai báo - chúng ta nói rằng cái tên đang được giới thiệu cũng đang được khởi tạo. Sau dấu bằng chúng ta có thể đặt một lần nữa một số biểu thức, nhưng lần này nó sẽ được đánh giá và giá trị của nó sẽ được sử dụng làm giá trị ban đầu cho đối tượng được khai báo.

Ví dụ:

|  |
| --- |
| int l = 90; /\* tương đương với: \*/  int l; l = 90; /\* nếu khai báo của l ở trong phạm vi khối \*/  int c = 2, b[c]; /\* ok, tương đương với: \*/  int c = 2; int b[c]; |

Trong phần mã của bạn sau này, bạn được phép viết biểu thức chính xác giống như phần khai báo của cái tên mới được giới thiệu, cho bạn một đối tượng có kiểu được xác định ở đầu khai báo, giả định rằng bạn đã gán giá trị hợp lệ cho tất cả các đối tượng được truy cập theo cách đó. Ví dụ:

|  |
| --- |
| void f()  {  int b2; /\* bạn nên có thể viết sau này trong mã của bạn b2  mà sẽ trực tiếp tham chiếu đến đối tượng số nguyên  mà b2 định danh \*/    b2 = 2; /\* gán giá trị cho b2 \*/    printf("%d", b2); /\*ok - nên in 2\*/  int \*b3; /\* bạn nên có thể viết sau này trong mã của bạn \*b3 \*/  b3 = &b2; /\* gán giá trị con trỏ hợp lệ cho b3 \*/  printf("%d", \*b3); /\* ok - nên in 2 \*/  int \*\*b4; /\* bạn nên có thể viết sau này trong mã của bạn \*\*b4 \*/  b4 = &b3;  printf("%d", \*\*b4); /\* ok - nên in 2 \*/  void (\*p)(); /\* bạn nên có thể viết sau này trong mã của bạn (\*p)() \*/  p = &f; /\* gán một giá trị con trỏ hợp lệ \*/  (\*p)(); /\* ok - gọi hàm f bằng cách lấy giá trị con trỏ bên trong p - p  và giải tham chiếu nó - \*p  kết quả là một hàm  sau đó được gọi - (\*p)() -  nó không phải là \*p() vì nếu không trước hết toán tử () được áp dụng vào p và sau đó đối tượng void kết quả được giải tham chiếu không phải là những gì chúng ta muốn ở đây \*/  } |

Khai báo của b3 xác định rằng bạn có thể tiềm ẩn sử dụng giá trị của b3 như một phương tiện để truy cập vào một số đối tượng số nguyên. Tất nhiên, để áp dụng con trỏ (\*) vào b3, bạn cũng cần có giá trị thích hợp được lưu trữ trong nó (xem thêm về con trỏ để biết thêm thông tin). Bạn cũng nên trước tiên lưu một số giá trị vào một đối tượng trước khi cố gắng truy xuất nó (bạn có thể xem thêm về vấn đề này tại đây). Chúng ta đã thực hiện tất cả các điều này trong các ví dụ trên.

|  |
| --- |
| int a3(); /\* bạn nên có thể gọi a3 \*/ |

Thành phần này cho trình biên dịch biết rằng bạn sẽ cố gắng gọi a3. Trong trường hợp này, a3 tham chiếu đến một hàm thay vì một đối tượng. Một khác biệt giữa đối tượng và hàm là các hàm luôn luôn có một số loại liên kết. Ví dụ:

|  |
| --- |
| void f1()  {  {  int f2(); /\* 1 tham chiếu đến một số hàm f2 \*/  }    {  int f2(); /\* tham chiếu đến cùng một hàm f2 như (1) \*/  }  } |

Trong ví dụ trên, 2 khai báo tham chiếu đến cùng một hàm f2, trong khi nếu chúng đang khai báo đối tượng thì trong ngữ cảnh này (có 2 phạm vi khối khác nhau), chúng sẽ được coi là 2 đối tượng riêng biệt khác nhau.

|  |
| --- |
| int (\*a3)(); /\* bạn nên có thể áp dụng con trỏ cho `a3` và sau đó gọi nó \*/ |

Bây giờ có vẻ như nó đang trở nên phức tạp, nhưng nếu bạn biết về thứ tự ưu tiên của toán tử, bạn sẽ không gặp vấn đề gì khi đọc các khai báo trên. Các dấu ngoặc đơn cần thiết vì toán tử \* có mức ưu tiên thấp hơn toán tử (). Trong trường hợp sử dụng toán tử chỉ số, biểu thức kết quả không thể được xác định thực sự sau phần khai báo vì chỉ số được sử dụng trong nó (giá trị trong [ và ]) sẽ luôn luôn cao hơn giá trị tối đa được phép cho đối tượng/hàm này.

|  |
| --- |
| int a4[5]; /\* ở đây a4 không nên được truy cập bằng chỉ số 5 sau này \*/ |

Nhưng nó nên được truy cập bằng tất cả các chỉ số khác nhau dưới 5. Ví dụ:

|  |
| --- |
| a4[0], a4[1]; a4[4]; |

a4[5] sẽ dẫn đến hành vi không xác định (undefined behavior).

Thông tin thêm về mảng có thể được tìm thấy ở đây.

|  |
| --- |
| int (\*a5)[5](); /\* here a4 could be applied indirection  indexed up to (but not including) 5  and called \* |

Thật không may cho chúng tôi, mặc dù về mặt cú pháp có thể đúng, việc tuyên bố bị cấm theo tiêu chuẩn hiện tại.a5

## TYPEDEF

Khái niệm typedef là một loại khai báo có từ khóa **typedef** đứng trước và trước kiểu dữ liệu. Ví dụ:

|  |
| --- |
| typedef int (\*(\*t0)())[5]; |

(Bạn cũng có thể đặt từ khóa typedef sau kiểu dữ liệu - như int typedef (\*(\*t0)())[5]; nhưng điều này không được khuyến khích)

Phần khai báo trên định nghĩa một định danh cho một tên kiểu dữ liệu typedef. Sau đó, bạn có thể sử dụng nó như sau:

|  |
| --- |
| t0 pf; |

Điều này có tác dụng tương tự như việc viết:

|  |
| --- |
| int (\*(\*pf)())[5]; |

Như bạn có thể thấy, tên kiểu dữ liệu typedef "lưu giữ" khai báo như một kiểu để sử dụng sau đó trong các khai báo khác. Điều này giúp bạn tiết kiệm một số lần nhấn phím. Bên cạnh đó, việc sử dụng typedef để khai báo vẫn giữ tính chất khai báo, do đó bạn không bị giới hạn chỉ bởi ví dụ trên:

|  |
| --- |
| t0 (\*pf1); |

Tương đương với:

|  |
| --- |
| int (\*(\*\*pf1)())[5]; |

Như bạn có thể thấy, khai báo typedef cho phép bạn tạo ra các tên kiểu dữ liệu tùy chỉnh để sử dụng trong mã của bạn, làm cho mã dễ đọc hơn và dễ duy trì hơn, đặc biệt khi làm việc với các kiểu dữ liệu phức tạp.

## SỬ DỤNG HẰNG SỐ TOÀN CỤC

Các tệp tiêu đề có thể được sử dụng để khai báo các tài nguyên chỉ đọc toàn cục, chẳng hạn như bảng chuỗi.

Hãy khai báo chúng trong một tệp tiêu đề riêng biệt, sau đó bao gồm tệp đó vào bất kỳ tệp nào ("Đơn vị Dịch") muốn sử dụng chúng. Thường thì cùng một tiêu đề cũng được sử dụng để khai báo một enumeration liên quan để xác định tất cả các nguồn tài nguyên chuỗi:

**resources.h:**

|  |
| --- |
| #ifndef RESOURCES\_H  #define RESOURCES\_H  typedef enum {  RESOURCE\_UNDEFINED = -1,  RESOURCE\_UNKNOWN = 0,  RESOURCE\_OK,  RESOURCE\_CANCEL,  RESOURCE\_ABORT,  RESOURCE\_MAX  } EnumResourceID;  extern const char \* const resources[RESOURCE\_MAX];  #endif |

Để thực sự định nghĩa các nguồn tài nguyên, tạo một tệp .c liên quan, đó là một đơn vị dịch khác chứa các phiên bản thực tế của những gì đã được khai báo trong tệp tiêu đề (.h):

**resources.c:**

|  |
| --- |
| #include "resources.h"  const char \* const resources[RESOURCE\_MAX] = {  "<unknown>",  "OK",  "Cancel",  "Abort"  }; |

Một chương trình sử dụng này có thể như sau:

**main.c:**

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  #include "resources.h"  int main(void)  {  EnumResourceID resource\_id = RESOURCE\_UNDEFINED;  while ((++resource\_id) < RESOURCE\_MAX)  {  printf("resource ID: %d, resource: '%s'\n", resource\_id, resources[resource\_id]);  }  return EXIT\_SUCCESS; |

Biên dịch ba tệp trên bằng GCC và liên kết chúng để trở thành tệp chương trình main chẳng hạn như sau:

|  |
| --- |
| gcc -Wall -Wextra -pedantic -Wconversion -g main.c resources.c -o main |

Chạy chương trình đã tạo

|  |
| --- |
| $ ./main |

Và kết quả nhận được sẽ là:

|  |
| --- |
| resource ID: 0, resource: ''  resource ID: 1, resource: 'OK'  resource ID: 2, resource: 'Cancel'  resource ID: 3, resource: 'Abort' |

## SỬ DỤNG QUY TẮC PHẢI TRÁI HOẶC XOẮN ỐC ĐỂ GIẢI MÃ CÁC KHAI BÁO TRONG C

Quy tắc "phải-trái" là một quy tắc hoàn toàn thông thường để giải mã các khai báo C. Nó cũng có thể hữu ích trong việc tạo ra chúng.

Đọc các biểu tượng khi bạn gặp chúng trong khai báo...

|  |
| --- |
| \* như "con trỏ tới" - luôn ở phía trái  [] như "mảng của" - luôn ở phía phải  () như "hàm trả về" - luôn ở phía phải |

**Cách áp dụng quy tắc:**

**BƯỚC 1**

Tìm cái định danh. Đây là điểm khởi đầu của bạn. Sau đó, tự nói với bản thân, "cái định danh là." Bạn đã bắt đầu khai báo của mình.

**BƯỚC 2**

Nhìn vào các biểu tượng ở bên phải của cái định danh. Ví dụ, nếu bạn thấy () ở đó, thì bạn biết đó là khai báo cho một hàm. Vì vậy, bạn sẽ có "cái định danh là hàm trả về". Hoặc nếu bạn thấy [] ở đó, bạn sẽ nói "cái định danh là mảng của". Tiếp tục sang phải cho đến khi bạn hết biểu tượng HOẶC gặp dấu ngoặc phải ). (Nếu bạn gặp dấu ngoặc trái (, đó là phần đầu của biểu tượng (), ngay cả khi có gì đó ở giữa các dấu ngoặc. Thêm thông tin về điều này ở dưới.)

**BƯỚC 3**

Nhìn vào các biểu tượng bên trái của cái định danh. Nếu nó không phải là một trong những biểu tượng ở trên (ví dụ, "int"), chỉ cần nói nó. Nếu không, dịch nó sang tiếng Anh bằng bảng ở trên. Tiếp tục sang trái cho đến khi bạn hết biểu tượng HOẶC gặp dấu ngoặc trái (.

Bây giờ lặp lại các bước 2 và 3 cho đến khi bạn đã tạo ra khai báo của mình.

**Dưới đây là một số ví dụ:**

|  |
| --- |
| int \*p[]; |

Trước tiên, tìm cái định danh:

|  |
| --- |
| int \*p[];  ^ |

“là p”

Tiếp theo, di chuyển sang phải cho đến khi hết biểu tượng hoặc gặp dấu ngoặc phải.

|  |
| --- |
| int \*p[];  ^^ |

“p là mảng của”

Không thể di chuyển sang phải nữa (hết biểu tượng), vậy di chuyển sang trái và tìm:

|  |
| --- |
| int \*p[];  ^ |

"p là mảng của con trỏ tới"

Tiếp tục sang trái và tìm:

|  |
| --- |
| int \*p[];  ^^^ |

"p là mảng của con trỏ tới int".

(hoặc "p là một mảng trong đó mỗi phần tử thuộc kiểu con trỏ đến int")

**Hoặc một ví dụ khác:**

|  |
| --- |
| int \*(\*func())(); |

Tìm cái định danh

|  |
| --- |
| int \*(\*func())();  ^^^^ |

“là func”

Di chuyển sang phải

|  |
| --- |
| nt \*(\*func())();  ^^ |

"**func** là hàm trả về"

Không thể di chuyển sang phải nữa vì có dấu ngoặc phải, vậy di chuyển sang trái.

|  |
| --- |
| int \*(\*func())();  ^ |

"**func** là hàm trả về con trỏ trỏ tới"

Không thể di chuyển sang trái nữa vì có dấu ngoặc trái, vậy tiếp tục sang phải.

|  |
| --- |
| int \*(\*func())();  ^^ |

"**func** là hàm trả về con trỏ tới hàm trả về"

Không thể di chuyển sang phải nữa vì chúng ta hết biểu tượng, vậy di chuyển sang trái.

|  |
| --- |
| int \*(\*func())();  ^ |

"**func** là hàm trả về con trỏ tới hàm trả về con trỏ tới"

Và cuối cùng, tiếp tục sang trái, vì không còn gì ở bên phải.

|  |
| --- |
| int \*(\*func())();  ^^^ |

"**func** là hàm trả về con trỏ tới hàm trả về con trỏ trỏ tới int".

Như bạn có thể thấy, quy tắc này có thể rất hữu ích. Bạn cũng có thể sử dụng nó để kiểm tra lý thuyết của mình trong khi bạn tạo các khai báo và để cho bạn một gợi ý về nơi đặt biểu tượng tiếp theo và xem liệu có cần dấu ngoặc đơn hay không.

Một số khai báo trông phức tạp hơn so với thực sự của chúng do kích thước mảng và danh sách đối số trong dạng nguyên mẫu. Nếu bạn thấy **[3]**, đó được đọc như **"mảng (kích thước 3)** của...". Nếu bạn thấy **(char \*, int)**, đó được đọc như **\*"hàm mong đợi (char ,int) và trả về...".**

**Dưới đây là một ví dụ thú vị**:

|  |
| --- |
| int (\*(\*fun\_one)(char \*,double))[9][20]; |

Tôi sẽ không đi qua từng bước để giải mã điều này.

"fun\_one là con trỏ tới hàm mong đợi (char ,double) và trả về con trỏ tới mảng (kích thước 9) của mảng (kích thước 20) của int."

Như bạn có thể thấy, nó không phức tạp nếu bạn loại bỏ kích thước mảng và danh sách đối số:

|  |
| --- |
| int (\*(\*fun\_one)())[][]; |

Bạn có thể giải mã nó như cách đó và sau đó thêm kích thước mảng và danh sách đối số sau.

**Một số lời kết:**

Hoàn toàn có thể tạo các khai báo không hợp lệ bằng cách sử dụng quy tắc này, vì vậy cần có một số kiến thức về điều gì hợp lệ trong C. Ví dụ, nếu cái trên là:

|  |
| --- |
| int \*((\*fun\_one)())[][]; |

nó sẽ đọc "fun\_one là con trỏ tới hàm trả về mảng của mảng của con trỏ tới int". Vì một hàm không thể trả về một mảng, mà chỉ có thể trả về một con trỏ tới mảng, khai báo đó là không hợp lệ.

Các kết hợp không hợp lệ bao gồm:

|  |
| --- |
| []() - không thể có một mảng của hàm  ()() - không thể có một hàm trả về hàm  ()[] - không thể có một hàm trả về mảng |

Trong tất cả các trường hợp trên, bạn sẽ cần một tập hợp dấu ngoặc để kết nối một biểu tượng \* ở bên trái giữa các biểu tượng () và [] ở bên phải để làm cho khai báo hợp lệ.

Dưới đây là một số ví dụ khác:

**Hợp lệ:**

|  |
| --- |
| int i; an int  int \*p; an int pointer (ptr to an int)  int a[]; an array of ints  int f(); a function returning an int  int \*\*pp; a pointer to an int pointer (ptr to a ptr to an int)  int (\*pa)[]; a pointer to an array of ints  int (\*pf)(); a pointer to a function returning an int  int \*ap[]; an array of int pointers (array of ptrs to ints)  int aa[][]; an array of arrays of ints  int \*fp(); a function returning an int pointer  int \*\*\*ppp; a pointer to a pointer to an int pointer  int (\*\*ppa)[]; a pointer to a pointer to an array of ints  int (\*\*ppf)(); a pointer to a pointer to a function returning an int  int \*(\*pap)[]; a pointer to an array of int pointers  int (\*paa)[][]; a pointer to an array of arrays of ints  int \*(\*pfp)(); a pointer to a function returning an int pointer  int \*\*app[]; an array of pointers to int pointers  int (\*apa[])[]; an array of pointers to arrays of ints  int (\*apf[])(); an array of pointers to functions returning an int  int \*aap[][]; an array of arrays of int pointers  int aaa[][][]; an array of arrays of arrays of int  int \*\*fpp(); a function returning a pointer to an int pointer  int (\*fpa())[]; a function returning a pointer to an array of ints  int (\*fpf())(); a function returning a pointer to a function returning an int |

**Không hợp lệ:**

|  |
| --- |
| int af[](); an array of functions returning an int  int fa()[]; a function returning an array of ints  int ff()(); a function returning a function returning an int  int (\*pfa)()[]; a pointer to a function returning an array of ints  int aaf[][](); an array of arrays of functions returning an int  int (\*paf)[](); a pointer to a an array of functions returning an int  int (\*pff)()(); a pointer to a function returning a function returning an int  int \*afp[](); an array of functions returning int pointers  int afa[]()[]; an array of functions returning an array of ints  int aff[]()(); an array of functions returning functions returning an int  int \*fap()[]; a function returning an array of int pointers  int faa()[][]; a function returning an array of arrays of ints  int faf()[](); a function returning an array of functions returning an int  int \*ffp()(); a function returning a function returning an int pointer |

Nguồn: <http://ieng9.ucsd.edu/~cs30x/rt_lt.rule.html>

# STRUCTURE PADDING VÀ PACKING

Mặc định, trình biên dịch C sắp xếp cấu trúc sao cho mỗi thành viên có thể được truy cập nhanh chóng, mà không phải chịu **incurring penalties** cho **'unaligned access’**, một vấn đề với các máy RISC như DEC Alpha và một số CPU ARM.

Tùy thuộc vào kiến trúc CPU và trình biên dịch, một cấu trúc có thể chiếm nhiều không gian trong bộ nhớ hơn tổng kích thước của các thành viên thành phần của nó. Trình biên dịch có thể thêm padding (phần đệm) giữa các **members** hoặc ở cuối cấu trúc, nhưng không phải ở đầu.

**Packing** sẽ ghi đè lớp **padding** mặc định.

## PACKING STRUCTURES

Mặc định, cấu trúc được **padded** trong C. Nếu bạn muốn tránh hành vi này, bạn phải yêu cầu nó một cách rõ ràng. Dưới GCC, đó là **attribute((packed))**. Xem ví dụ này trên máy 64 bit:

|  |
| --- |
| struct foo {  char \*p; /\* 8 byte \*/  char c; /\* 1 byte \*/  long x; /\* 8 byte \*/  }; |

Cấu trúc này sẽ tự động được **padded** để có **8-byte alignment** và sẽ trông như sau:

|  |
| --- |
| struct foo {  char \*p; /\* 8 byte \*/  char c; /\* 1 byte \*/  char pad[7]; /\* 7 byte được thêm vào bởi trình biên dịch \*/  long x; /\* 8 byte \*/  }; |

Vì vậy, **sizeof (struct foo**) sẽ cho chúng ta kết quả là 24 thay vì 17. Điều này xảy ra vì trình biên dịch 64 bit đọc/ghi từ/bộ nhớ trong **8 byte của word** trong mỗi bước và rõ ràng khi cố gắng viết **char c**; một byte vào bộ nhớ thì 8 byte hoàn chỉnh (tức là **word**) được đọc và chỉ tiêu tốn **byte đầu tiên** của nó và **bảy byte tiếp theo** của nó không được sử dụng và không thể truy cập cho bất kỳ hoạt động đọc và ghi nào cho việc **structure padding**.

**Structure packing**

Nhưng nếu bạn thêm thuộc tính **packed**, trình biên dịch sẽ không thêm **padding**:

|  |
| --- |
| struct \_\_attribute\_\_((\_\_packed\_\_)) foo {  char \*p; /\* 8 byte \*/  char c; /\* 1 byte \*/  long x; /\* 8 byte \*/  }; |

Bây giờ **sizeof(struct foo)** sẽ trả về 17.

Thường, **packed structures** được sử dụng để:

* Để tiết kiệm không gian.
* Để định dạng cấu trúc dữ liệu để truyền qua mạng mà không phụ thuộc vào **architecture alignment** của từng nút của mạng.

Cần phải lưu ý rằng một số bộ xử lý như ARM Cortex-M0 không cho phép truy cập bộ nhớ **unaligned memory**; trong những trường hợp như vậy, **structure packing** có thể dẫn đến hành vi không xác định và có thể làm hỏng CPU.

## STRUCTURE PADDING

Giả sử cấu trúc này được định nghĩa và biên dịch bằng trình biên dịch 32 bit:

|  |
| --- |
| struct test\_32 {  int a; // 4 byte  short b; // 2 byte  int c; // 4 byte  } str\_32; |

Chúng ta có thể mong đợi cấu trúc này chiếm 10 byte bộ nhớ, nhưng khi in ra sizeof (str\_32) thì chúng ta thấy nó sử dụng 12 byte.

Điều này xảy ra vì trình biên dịch **aligns variables** để truy cập nhanh. Một mẫu phổ biến là khi **kiểu cơ sở** chiếm N byte (trong đó N là một lũy thừa của 2 như 1, 2, 4, 8, 16 - và hiếm khi có kích thước lớn hơn), biến nên được **aligned** trên ranh giới N-byte (là bội số của N byte).

Cho cấu trúc được hiển thị với **sizeof(int) == 4** và **sizeof(short) == 2**, một bố trí phổ biến là:

|  |
| --- |
| int a; lưu trữ tại vị trí offset 0; kích thước 4.  short b; lưu trữ tại vị trí offset 4; kích thước 2.  khoảng trống không tên tại vị trí offset 6; kích thước 2.  int c; lưu trữ tại vị trí offset 8; kích thước 4. |

|  |  |  |  |
| --- | --- | --- | --- |
| 11 | 10 | 9 | 8 |
| **c** | **c** | **c** | **c** |
| 7 | 6 | 5 | 4 |
| **padding** | **padding** | **b** | **b** |
| 3 | 2 | 1 | 0 |
| **a** | **a** | **a** | **a** |

Do đó, struct test\_32 chiếm 12 byte bộ nhớ. Trong ví dụ này, không có **trailing padding**.

Trình biên dịch sẽ đảm bảo rằng các biến struct test\_32 được lưu trữ bắt đầu từ ranh giới 4 byte, để các thành viên trong cấu trúc sẽ được **properly aligned** cho việc truy cập nhanh. Các hàm phân bổ bộ nhớ như malloc(), calloc() và realloc() phải đảm bảo rằng con trỏ được trả về có **sufficiently well aligned** cho việc sử dụng với bất kỳ kiểu dữ liệu nào, vì vậy các cấu trúc được cấp phát động cũng sẽ được **aligned** đúng.

Bạn có thể gặp tình huống kỳ lạ như trên bộ xử lý Intel x86\_64 64 bit (ví dụ Intel Core i7 - một máy Mac chạy macOS Sierra hoặc Mac OS X), khi biên dịch ở chế độ 32 bit, trình biên dịch đặt **double aligned on a 4-byte boundary**; nhưng trên cùng phần cứng, khi biên dịch ở chế độ 64 bit, trình biên dịch đặt **double aligned on an 8- byte boundary**.

# MEMORY MANAGEMENT (QUẢN LÝ BỘ NHỚ)

|  |  |
| --- | --- |
| **Tên** | **sự miêu tả** |
| size (malloc, realloc và aligned\_alloc) | Kích thước tổng cộng của bộ nhớ theo byte. Đối với aligned\_alloc, kích thước phải là bội số nguyên của alignment. |
| size (calloc) | kích thước của mỗi phần tử |
| nelements | Số lượng các phần tử |
| ptr | Con trỏ trỏ đến bộ nhớ được cấp phát trước đó bởi malloc, calloc, realloc hoặc aligned\_alloc. |
| alignment | Alignment bộ nhớ được cấp phát |

Để quản lý bộ nhớ được cấp phát động, thư viện tiêu chuẩn C cung cấp các hàm malloc(), calloc(), realloc() và free(). Trong C99 và phiên bản sau, còn có aligned\_alloc(). Một số hệ thống cũng cung cấp alloca().

## ALLOCATING MEMORY (CẤP PHÁT BỘ NHỚ)

### Cấp phát tiêu chuẩn

Các hàm cấp phát bộ nhớ động của C được định nghĩa trong tiêu đề <stdlib.h>. Nếu bạn muốn cấp phát không gian bộ nhớ cho một đối tượng một cách động, bạn có thể sử dụng mã sau:

|  |
| --- |
| int \*p = malloc(10 \* sizeof \*p);  if (p == NULL)  {  perror("malloc() failed");  return -1;  } |

Điều này tính toán số byte mà mười **int** chiếm trong bộ nhớ, sau đó yêu cầu nhiều byte đó từ **malloc** và gán kết quả (nghĩa là địa chỉ bắt đầu của đoạn bộ nhớ vừa được tạo bằng **malloc**) cho một con trỏ có tên p

Việc sử dụng sizeof để tính toán lượng bộ nhớ cần yêu cầu là việc tốt vì kết quả của sizeof được định nghĩa bởi người triển khai (trừ các kiểu ký tự, đó là char, signed char và unsigned char, với sizeof được định nghĩa luôn luôn là 1).

**Vì malloc có thể không đáp ứng được yêu cầu nên nó có thể trả về một con null. Điều quan trọng là phải kiểm tra điều này để ngăn chặn các nỗ lực dereference con trỏ null sau này.**

Bộ nhớ được cấp phát động bằng **malloc()** có thể được thay đổi kích thước bằng **realloc()** hoặc, khi không còn cần thiết, được giải phóng bằng **free().**

Hoặc, việc khai báo int array[10]; sẽ cấp phát cùng lượng bộ nhớ. Tuy nhiên, nếu nó được khai báo bên trong một hàm mà không có từ khóa static, nó chỉ có thể sử dụng trong phạm vi của hàm mà nó được khai báo và các hàm mà nó gọi (bởi vì mảng sẽ được cấp phát trên ngăn xếp và không gian sẽ được giải phóng để tái sử dụng khi hàm trả về). Hoặc, nếu nó được định nghĩa với static trong một hàm, hoặc nếu nó được định nghĩa bên ngoài bất kỳ hàm nào, thì tuổi thọ của nó là tuổi thọ của chương trình. Con trỏ cũng có thể được trả về từ một hàm, tuy nhiên một hàm trong C không thể trả về một mảng.

### Zeroed Memory

Bộ nhớ được trả về bởi malloc có thể không được khởi tạo thành giá trị hợp lý, và cần phải chú ý để đặt giá trị 0 vào bộ nhớ với memset hoặc sao chép ngay lập tức một giá trị thích hợp vào nó. Hoặc, calloc trả về một khối bộ nhớ với kích thước mong muốn trong đó tất cả các bit được khởi tạo thành 0. Điều này không cần phải giống như biểu diễn của số 0 dấu phẩy động hoặc hằng số con trỏ null.

|  |
| --- |
| int \*p = calloc(10, sizeof \*p);  if (p == NULL)  {  perror("calloc() failed");  return -1;  } |

Một lưu ý về **calloc**: Hầu hết các triển khai (phổ biến) sẽ tối ưu hóa **calloc()** cho hiệu suất, vì vậy nó sẽ **nhanh hơn** việc gọi **malloc()**, sau đó **memset()**, mặc dù hiệu ứng tông rất giống nhau.

### Aligned Memory

Phiên bản ≥ C11

C11 giới thiệu một hàm mới là **aligned\_alloc()** có khả năng cấp phát không gian với **alignment** đã cho. Nó có thể được sử dụng nếu bộ nhớ cần được cấp phát cần được aligned tại các ranh giới cụ thể không thể được đáp ứng bởi **malloc()** hoặc **calloc().** Các hàm malloc() và calloc() cấp phát bộ nhớ **aligned** cho bất kỳ kiểu đối tượng nào (tức là sự căn chỉnh là alignof(max\_align\_t)). Nhưng với aligned\_alloc(), có thể yêu cầu các **alignments** lớn hơn.

|  |
| --- |
| /\* Allocates 1024 bytes with 256 bytes alignment. \*/  char \*ptr = aligned\_alloc(256, 1024);  if (ptr) {  perror("aligned\_alloc()");  return -1;  }  free(ptr); |

Tiêu chuẩn C11 đặt hai hạn chế:

1) kích thước (đối số thứ hai) yêu cầu phải là bội số nguyên của alignment (đối số đầu tiên) và

2) giá trị của alignment phải là alignment hợp lệ được hỗ trợ bởi triển khai. Vi phạm một trong hai điều này dẫn đến hành vi không xác định.

## GIẢI PHÓNG BỘ NHỚ (FREEING MEMORY)

Có thể giải phóng bộ nhớ được cấp phát động bằng cách gọi hàm free().

|  |
| --- |
| int \*p = malloc(10 \* sizeof \*p); /\* cấp phát bộ nhớ \*/  if (p == NULL)  {  perror("malloc failed");  return -1;  }  free(p); /\* giải phóng bộ nhớ \*/  /\* chú ý rằng sau khi free(p), thậm chí việc sử dụng \*giá trị\* của con trỏ p  có hành vi không xác định, cho đến khi một giá trị mới được lưu trữ vào nó. \*/  /\* tái sử dụng/tái mục đích chính con trỏ \*/  int i = 42;  p = &i; /\* Điều này hợp lệ, có hành vi xác định \*/ |

Bộ nhớ mà con trỏ p trỏ đến sẽ được thu hồi (hoặc bởi triển khai libc hoặc bởi hệ điều hành dưới), do đó **truy cập vào khối bộ nhớ bị giải phóng** thông qua p sẽ dẫn đến hành vi không xác định. Con trỏ trỏ đến các yếu tố bộ nhớ đã được giải phóng thường được gọi là con trỏ treo (dangling pointers), và tạo ra một rủi ro về bảo mật.

Hơn nữa, tiêu chuẩn C nêu rõ rằng thậm chí việc truy cập giá trị của con trỏ treo có hành vi không xác định.

Chú ý rằng con trỏ p có thể được tái sử dụng chính nó như được hiển thị ở trên.

Vui lòng lưu ý rằng bạn chỉ có thể gọi free() trên các con trỏ đã được trả về trực tiếp từ các hàm malloc(), calloc(), realloc() và aligned\_alloc(), hoặc khi tài liệu cho bạn biết bộ nhớ đã được cấp phát theo cách đó (các hàm như strdup () là các ví dụ đáng chú ý). Việc giải phóng một con trỏ được, được đạt được bằng cách sử dụng toán tử & trên một biến, hoặc nằm ở giữa một khối được cấp phát, là bị cấm. Lỗi như vậy thường không được chẩn đoán bởi trình biên dịch của bạn nhưng sẽ dẫn đến trạng thái không xác định của chương trình.

Có hai phương pháp phổ biến để ngăn các trường hợp không xác định như vậy.

**Phương pháp đầu tiên và ưu tiên là đơn giản - khi p không còn cần thiết, p chính nó ngưng tồn tại, ví dụ:**

|  |
| --- |
| if (something\_is\_needed())  {  int \*p = malloc(10 \* sizeof \*p);  if (p == NULL)  {  perror("malloc failed");  return -1;  }  /\* thực hiện bất kỳ điều gì cần với p \*/  free(p);  } |

Bằng cách gọi free() trực tiếp trước khi kết thúc khối chứa (tức là }), p chính nó ngưng tồn tại. Trình biên dịch sẽ hiển thị một lỗi biên dịch trên mọi nỗ lực để sử dụng p sau đó.

**Phương pháp thứ hai là vô hiệu hóa con trỏ chính nó sau khi giải phóng bộ nhớ mà nó trỏ đến:**

|  |
| --- |
| free(p);  p = NULL; // bạn cũng có thể sử dụng 0 thay vì NULL |

Lý do cho phương pháp này:

* Trên nhiều nền tảng, việc cố gắng dereference một con trỏ null sẽ gây ra sự cố ngay lập tức: Segmentation fault. Ở đây, ít nhất chúng ta có một stack trace trỏ đến biến đã được sử dụng sau khi được giải phóng.
* Không đặt con trỏ thành NULL, chúng ta có con trỏ treo. Chương trình có khả năng sẽ vẫn bị lỗi, nhưng sau đó, vì bộ nhớ mà con trỏ trỏ tới sẽ âm thầm bị hỏng. Lỗi như vậy khó để theo dõi vì chúng có thể dẫn đến một stack gọi hoàn toàn không liên quan đến vấn đề ban đầu. Phương pháp này tuân theo nguyên tắc "fail-fast" (sự thất bại nhanh chóng).
* Giải phóng một con trỏ null là an toàn. Tiêu chuẩn C xác định rằng free(NULL) không có tác dụng:
  + Hàm free gây ra việc không gian mà ptr trỏ đến được giải phóng, nghĩa là được thực hiện cho phép cấp phát tiếp theo. Nếu ptr là một con trỏ null, không có hành động nào xảy ra. Nếu
  + đối số không phù hợp với một con trỏ trước đó được trả về bởi hàm calloc, malloc hoặc realloc, hoặc nếu không gian đã được giải phóng bởi cuộc gọi free hoặc realloc, hành vi là không xác định.
* Đôi khi, phương pháp đầu tiên không thể được sử dụng (ví dụ: bộ nhớ được cấp phát trong một hàm và được giải phóng sau đó trong một hàm hoàn toàn khác).

## THAY ĐỔI KÍCH THƯỚC BỘ NHỚ CẤP PHÁT (REALLOCATING MEMORY)

Có thể bạn cần mở rộng hoặc thu hẹp không gian lưu trữ của con trỏ sau khi bạn đã cấp phát bộ nhớ cho nó. Hàm void \*realloc(void \*ptr, size\_t size) giải phóng đối tượng cũ được trỏ bởi ptr và trả về một con trỏ đến một đối tượng có kích thước được xác định bởi size. ptr là con trỏ đến một khối bộ nhớ được cấp phát trước đó bằng malloc, calloc hoặc realloc (hoặc con trỏ null) để được cấp phát lại. Nội dung tối đa có thể của bộ nhớ gốc được bảo tồn. Nếu kích thước mới lớn hơn, bất kỳ bộ nhớ bổ sung vượt quá kích thước cũ sẽ không được khởi tạo. Nếu kích thước mới ngắn hơn, nội dung của phần thu nhỏ bị mất. Nếu ptr là NULL, một khối mới được cấp phát và một con trỏ đến nó được trả về bởi hàm.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(int argc, char \*argv[]) {  int \*p = malloc(10 \* sizeof \*p);  if (NULL == p)  {  perror("malloc() failed");  return EXIT\_FAILURE;  }  p[0] = 42;  p[9] = 15;  /\* Reallocate array to a larger size, storing the result into a  \* temporary pointer in case realloc() fails. \*/  {  int \*temporary = realloc(p, 1000000 \* sizeof \*temporary);  /\* realloc() failed, the original allocation was not free'd yet. \*/  if (NULL == temporary)  {  perror("realloc() failed");  free(p); /\* Clean up. \*/  return EXIT\_FAILURE;  }  p = temporary;  }  /\* From here on, array can be used with the new size it was  \* realloc'ed to, until it is free'd. \*/  /\* The values of p[0] to p[9] are preserved, so this will print:  42 15  \*/  printf("%d %d\n", p[0], p[9]);  free(p);  return EXIT\_SUCCESS;  } |

Đối tượng đã được cấp phát lại có thể hoặc không có địa chỉ giống như \*p. Do đó, quan trọng là bắt giá trị trả về từ realloc chứa địa chỉ mới nếu lời gọi thành công.

Hãy đảm bảo bạn gán giá trị trả về của realloc vào một biến tạm thay vì p ban đầu. realloc sẽ trả về null trong trường hợp bất kỳ sự cố nào, điều này sẽ ghi đè lên con trỏ. Điều này sẽ làm mất dữ liệu của bạn và tạo ra một memory leak.

## REALLOC(PTR, 0) KHÔNG TƯƠNG ĐƯƠNG VỚI FREE(PTR)

realloc cơ bản tương đương với việc malloc + memcpy + free trên con trỏ khác.

Nếu kích thước không gian yêu cầu là 0, hành vi của realloc là xác định bởi cài đặt. Điều này tương tự cho tất cả các hàm cấp phát bộ nhớ nhận tham số kích thước có giá trị 0. Những hàm như vậy thực tế có thể trả về một con trỏ không phải là null, nhưng con trỏ đó không bao giờ được dereferenced.

Do đó, realloc(ptr, 0) không tương đương với free(ptr). Nó có thể:

* Là một cài đặt "lười" và chỉ trả về ptr.
* Giải phóng ptr, cấp phát một phần tử giả và trả về nó.
* Giải phóng ptr và trả về 0.
* Chỉ trả về 0 cho thất bại và không làm gì khác.

Vì vậy, đặc biệt là hai trường hợp cuối cùng không thể phân biệt được bằng mã ứng dụng.

Điều này có nghĩa là realloc(ptr, 0) có thể không thực sự giải phóng/bộ nhớ, và do đó nó không bao giờ nên được sử dụng để thay thế cho free.

## MẢNG ĐA CHIỀU CÓ KÍCH THƯỚC THAY ĐỔI

Phiên bản ≥ C99

Kể từ C99, C hỗ trợ mảng có kích thước biến đổi (VLA), đại diện cho các mảng có giới hạn chỉ được biết vào thời điểm khởi tạo. Mặc dù bạn cần phải cẩn thận để không cấp phát VLA quá lớn (chúng có thể gây sự cố cho stack của bạn), sử dụng con trỏ đến VLA và sử dụng chúng trong biểu thức sizeof là hoàn toàn hợp lệ.

|  |
| --- |
| double sumAll(size\_t n, size\_t m, double A[n][m]) {  double ret = 0.0;  for (size\_t i = 0; i < n; ++i)  for (size\_t j = 0; j < m; ++j)  ret += A[i][j]  return ret;  }  int main(int argc, char \*argv[argc+1]) {  size\_t n = argc\*10;  size\_t m = argc\*8;  double (\*matrix)[m] = malloc(sizeof(double[n][m]));  // khởi tạo ma trận bằng cách nào đó  double res = sumAll(n, m, matrix);  printf("result is %g\n", res);  free(matrix);  } |

Ở đây, matrix là một con trỏ đến các phần tử kiểu double[m], và biểu thức sizeof với double[n][m] đảm bảo rằng nó chứa không gian cho n phần tử như vậy. Tất cả không gian này được cấp phát liên tiếp và do đó có thể bị giải phóng bằng một lời gọi đơn cho hàm free.

Sự xuất hiện của VLA trong ngôn ngữ cũng ảnh hưởng đến các khai báo có thể của mảng và con trỏ trong tiêu đề hàm. Bây giờ, một biểu thức số nguyên tổng quát được phép ở bên trong dấu [] của các tham số mảng. Cho cả hai hàm, các biểu thức trong [] sử dụng các tham số đã được khai báo trước trong danh sách tham số. Đối với hàm sumAll, đó là các độ dài mà mã nguồn của người dùng mong đợi cho ma trận. Như với tất cả các tham số hàm mảng trong C, chiều trong cùng được viết lại thành một kiểu con trỏ, vì vậy điều này tương đương với khai báo:

|  |
| --- |
| double sumAll(size\_t n, size\_t m, double (\*A)[m]); |

Tức là, n thực sự không phải là một phần của giao diện hàm, nhưng thông tin này có thể hữu ích cho tài liệu và nó cũng có thể được sử dụng bởi các trình biên dịch kiểm tra giới hạn để cảnh báo về truy cập vượt giới hạn.

Tương tự, đối với hàm main, biểu thức argc+1 là độ dài tối thiểu mà tiêu chuẩn C quy định cho đối số argv.

Lưu ý rằng chính thức hỗ trợ VLA là tùy chọn trong C11, nhưng chúng ta không biết có trình biên dịch nào thực hiện C11 mà không có chúng. Bạn có thể kiểm tra bằng macro \_\_STDC\_NO\_VLA\_\_ nếu cần.

## ALLOCA: CẤP PHÁT BỘ NHỚ TRÊN STACK

Cảnh báo: alloca chỉ được đề cập ở đây để hoàn thiện. Nó hoàn toàn không có khả năng chuyển đổi (không được bao gồm trong bất kỳ tiêu chuẩn thông thường nào) và có một số tính năng tiềm ẩn nguy hiểm có thể làm cho nó không an toàn cho những người không có kinh nghiệm. Mã C hiện đại nên thay thế nó bằng Variable Length Arrays (VLA).

Manual page

|  |
| --- |
| #include <alloca.h>  // phiên bản glibc của stdlib.h mặc định bao gồm alloca.h  void foo(int size) {  char \*data = alloca(size);  /\*  thân hàm;  \*/  // dữ liệu tự động được giải phóng  } |

Cấp phát bộ nhớ trên stack frame của hàm gọi, không gian được trỏ đến bởi con trỏ trả về sẽ tự động được giải phóng khi hàm gọi kết thúc.

Mặc dù hàm này thuận tiện cho việc quản lý bộ nhớ tự động, hãy nhớ rằng yêu cầu cấp phát lớn có thể gây ra stack overflow và bạn không thể sử dụng free với bộ nhớ được cấp phát bằng alloca (điều này có thể gây ra vấn đề hơn với stack overflow).

Vì những lý do này, không nên sử dụng alloca trong một vòng lặp hoặc trong một hàm đệ quy.

Và vì bộ nhớ được giải phóng khi hàm trả về, bạn không thể trả con trỏ như một kết quả của hàm (hành vi này sẽ không xác định).

**Tóm lại**

* Gọi giống với malloc.
* Tự động được giải phóng khi hàm trả về.
* Không tương thích với các hàm free và realloc (hành vi không xác định).
* Con trỏ không thể được trả lại như kết quả của một hàm (hành vi không xác định).
* Kích thước cấp phát giới hạn bởi không gian stack, mà (trên hầu hết các máy tính) nhỏ hơn nhiều so với không gian heap có sẵn để sử dụng bởi malloc.
* Tránh sử dụng alloca và VLA (mảng kích thước biến đổi) trong một hàm duy nhất.

**Khuyến khích**

* Không nên sử dụng **alloca( )** trong mã mới.

Phiên bản ≥ C99:

Thay thế hiện đại.

|  |
| --- |
| void foo(int size) {  char data[size];  /\*  thân hàm;  \*/  // dữ liệu tự động được giải phóng  } |

Điều này hoạt động tương tự như alloca, và hoạt động ở những nơi mà alloca không hoạt động (trong vòng lặp, ví dụ). Nó giả định một trong hai: hiện thực C99 hoặc hiện thực C11 không xác định \_\_STDC\_NO\_VLA\_\_.

## QUẢN LÝ BỘ NHỚ DO NGƯỜI DÙNG ĐỊNH NGHĨA

Hàm malloc() thường gọi các hàm của hệ thống điều hành để thu được các trang bộ nhớ. Nhưng không có gì đặc biệt về hàm này và nó có thể được triển khai trực tiếp bằng ngôn ngữ C bằng cách khai báo một mảng tĩnh lớn và cấp phát từ nó (có một chút khó khăn trong việc đảm bảo căn chỉnh đúng, trong thực tế căn chỉnh đến 8 byte hầu như luôn luôn là đủ).

Để triển khai một kế hoạch đơn giản, một khối kiểm soát được lưu trữ trong vùng bộ nhớ ngay trước con trỏ sẽ được trả về từ cuộc gọi. Điều này có nghĩa là free() có thể được triển khai bằng cách trừ đi từ con trỏ trả về và đọc thông tin kiểm soát, thông tin này thường là kích thước khối cộng thêm một số thông tin cho phép đặt nó trở lại danh sách không cấp phát - một danh sách liên kết của các khối chưa cấp phát.

Khi người dùng yêu cầu cấp phát, danh sách không cấp phát được tìm kiếm cho đến khi tìm thấy một khối có kích thước giống hoặc lớn hơn số lượng được yêu cầu, sau đó nếu cần, khối sẽ được chia. Điều này có thể dẫn đến sự mảng bộ nhớ nếu người dùng liên tục thực hiện nhiều cấp phát và giải phóng không dự đoán được về kích thước và tại các khoảng thời gian không dự đoán (không phải tất cả các chương trình thực sự hoạt động như vậy, kế hoạch đơn giản thường là đủ cho các chương trình nhỏ).

|  |
| --- |
| /\* khối kiểm soát tiêu biểu \*/  struct block  {  size\_t size; /\* kích thước của khối \*/  struct block \*next; /\* khối kế tiếp trong danh sách không cấp phát \*/  struct block \*prev; /\* con trỏ phía trước đến khối trước trong bộ nhớ \*/  void \*padding; /\* cần 16 byte để làm cho nhiều lần của 8 \*/  }  static struct block arena[10000]; /\* cấp phát từ đây \*/  static struct block \*firstfree; |

Nhiều chương trình yêu cầu số lượng lớn cấp phát các đối tượng nhỏ có cùng kích thước. Điều này rất dễ triển khai. Đơn giản sử dụng một khối với con trỏ `next`. Ví dụ, nếu cần một khối 32 byte:

|  |
| --- |
| union block  {  union block \* next;  unsigned char payload[32];  }  static union block arena[100];  static union block \* head;  void init(void)  {  int i;  for (i = 0; i < 100 - 1; i++)  arena[i].next = &arena[i + 1];  arena[i].next = 0; /\* cuối cùng, null \*/  head = &block[0];  }  void \*block\_alloc()  {  void \*answer = head;  if (answer)  head = head->next;  return answer;  }  void block\_free(void \*ptr)  {  union block \*block = ptr;  block->next = head;  head - block;  } |

Kế hoạch này vô cùng nhanh và hiệu quả và có thể trở nên thông dụng với việc mất một phần sự rõ ràng.

# THỰC HIỆN HÀNH VI ĐƯỢC XÁC ĐỊNH (IMPLEMENTATION – DEFINED BEHAVIOUR)

## DỊCH PHẢI MỘT SỐ NGUYÊN ÂM

|  |
| --- |
| int signed\_integer = -1;  // The right shift operation exhibits implementation-defined behavior:  int result = signed\_integer >> 1; |

## GÁN MỘT GIÁ TRỊ NẰM NGOÀI PHẠM VI CỦA MỘT SỐ NGUYÊN

|  |
| --- |
| // Supposing SCHAR\_MAX, the maximum value that can be represented by a signed char, is  // 127, the behavior of this assignment is implementation-defined:  signed char integer;  integer = 128; |

## CẤP PHÁT SỐ BYTE BẰNG 0

|  |
| --- |
| // The allocation functions have implementation-defined behavior when the requested size  // of the allocation is zero.  void \*p = malloc(0); |

## BIỂU DIỄN SỐ NGUYÊN CÓ DẤU

Mỗi loại số nguyên đã ký có thể được biểu diễn ở bất kỳ một trong ba định dạng nào; nó được xác định theo triển khai cái nào được sử dụng. Việc triển khai được sử dụng cho bất kỳ loại số nguyên đã ký nào có độ rộng tối thiểu intcó thể được xác định trong thời gian chạy từ hai bit có thứ tự thấp nhất của biểu diễn giá trị -1trong loại đó, như vậy:

|  |
| --- |
| enum { sign\_magnitude = 1, ones\_compl = 2, twos\_compl = 3, };  #define SIGN\_REP(T) ((T)-1 & (T)3)  switch (SIGN\_REP(long)) {  case sign\_magnitude: { /\* làm gì đó \*/ break; }  case ones\_compl: { /\* làm khác đi \*/ break; }  case twos\_compl: { /\* làm khác nữa \*/ break; }  case 0: { \_Static\_assert(SIGN\_REP(long), "biểu diễn dấu giả"); }  } |

Mẫu tương tự áp dụng cho việc biểu diễn các loại hẹp hơn, nhưng chúng không thể được kiểm tra bằng kỹ thuật này vì các toán hạng của &phải tuân theo "các phép chuyển đổi số học thông thường" trước khi tính toán kết quả.

# ATOMICS

## ATOMICS VÀ CÁC TOÁN TỬ

Biến atomics có thể được truy cập cùng lúc bởi các luồng khác nhau mà không tạo ra tình trạng cạnh tranh (race conditions).

|  |
| --- |
| /\* a global static variable that is visible by all threads \*/  static unsigned \_Atomic active = ATOMIC\_VAR\_INIT(0);  int myThread(void\* a) {  ++active; // increment active race free  // do something  --active; // decrement active race free  return 0;  } |

Tất cả các hoạt động lvalue (hoạt động sửa đổi đối tượng) cho phép cho kiểu cơ bản đều được cho phép và sẽ không dẫn đến tình trạng cạnh tranh giữa các luồng khác nhau truy cập chúng.

* Các hoạt động trên các đối tượng atomic thường nhanh hơn hàng động so với các hoạt động số học thông thường. Điều này bao gồm cả các hoạt động load hoặc store đơn giản. Vì vậy, bạn chỉ nên sử dụng chúng cho các nhiệm vụ quan trọng.
* Các hoạt động số học thông thường và gán như a = a+1; thực sự là ba hoạt động trên a: trước hết là một hoạt động load, sau đó là phép cộng và cuối cùng là một hoạt động store. Điều này không an toàn đối với tình trạng cạnh tranh. Chỉ có các hoạt động a += 1; và a++; là an toàn.

# CÂU LỆNH NHẢY (JUMP STATEMENTS)

## SỬ DỤNG RETURN

**Trả về một giá trị**

Một trường hợp thường sử dụng: trả về từ hàm main()

|  |
| --- |
| #include <stdlib.h> /\* cho các macro EXIT\_xxx \*/  int main(int argc, char \*\* argv) {  if (2 < argc)  {  return EXIT\_FAILURE; /\* Chương trình yêu cầu một đối số:  ra khỏi ngay lập tức và bỏ qua phần còn lại của mã chương trình. \*/  }  /\* Làm điều gì đó. \*/  return EXIT\_SUCCESS;  } |

Ghi chú bổ sung

* Đối với một hàm có kiểu trả về là void (không bao gồm void \* hoặc các kiểu liên quan), câu lệnh return không nên có bất kỳ biểu thức đi kèm nào; tức là, câu lệnh return duy nhất được phép là return;.
* Đối với một hàm có kiểu trả về không phải là void, câu lệnh return không được xuất hiện mà không có biểu thức.
* Đối với hàm main() (và chỉ đối với main()), một câu lệnh return tường minh không bắt buộc (trong C99 hoặc phiên bản sau đó). Nếu việc thực thi đạt tới dấu }, một giá trị ngầm định là 0 sẽ được trả về. Một số người cho rằng việc bỏ qua câu lệnh return này là tập quán không tốt; những người khác đề xuất để bỏ nó đi.

**Không trả về gì cả**

Trả về từ một hàm void

|  |
| --- |
| void log(const char \* message\_to\_log)  {  if (NULL == message\_to\_log)  {  return; /\* Không có gì để ghi nhật ký, đi về ngay bây giờ, bỏ qua việc ghi nhật ký. \*/  }  fprintf(stderr, "%s:%d %s\n", \_\_FILE\_\_, \_LINE\_\_, message\_to\_log);  return; /\* Tùy chọn, vì hàm này không trả về giá trị gì cả. \*/  } |

## SỬ DỤNG GOTO ĐỂ NHẢY RA KHỎI CÁC VÒNG LẶP LỒNG NHAU

|  |
| --- |
| size\_t i, j;  for (i = 0; i < myValue && !breakout\_condition; ++i) {  for (j = 0; j < mySecondValue && !breakout\_condition; ++j) {  ... /\* Làm một số việc gì đó, có thể thay đổi breakout\_condition \*/  /\* Khi breakout\_condition == true thì vòng lặp kết thúc \*/  }  } |

Nhưng ngôn ngữ C cung cấp câu lệnh goto, có thể hữu ích trong trường hợp này. Bằng cách sử dụng nó với một nhãn được khai báo sau các vòng lặp, chúng ta có thể dễ dàng thoát khỏi các vòng lặp.

|  |
| --- |
| size\_t i, j;  for (i = 0; i < myValue; ++i) {  for (j = 0; j < mySecondValue; ++j) {  ...  if(breakout\_condition)  goto final;  }  }  final: |

Tuy nhiên, thường khi nhu cầu này xuất hiện, việc sử dụng câu lệnh return có thể tốt hơn. Cấu trúc này cũng được coi là "không có cấu trúc" trong lý thuyết lập trình cấu trúc.

Tình huống khác mà goto có thể hữu ích là để chuyển đến một trình xử lý lỗi:

|  |
| --- |
| ptr = malloc(N \* x);  if (!ptr)  goto out\_of\_memory;  /\* xử lý bình thường \*/  free(ptr);  return SUCCESS;  out\_of\_memory:  free(ptr); /\* không gây hại, và cần thiết nếu chúng ta có lỗi tiếp theo \*/  return FAILURE; |

Việc sử dụng gotogiữ luồng lỗi tách biệt với luồng điều khiển chương trình thông thường. Tuy nhiên, nó cũng được coi là "không có cấu trúc" theo nghĩa kỹ thuật.

## SỬ DỤNG BREAK VÀ CONTINUE

Đọc ngay lập tức **continue** trên đầu vào **không hợp lệ** hoặc **break** theo yêu cầu của người dùng hoặc phần cuối của tệp:

|  |
| --- |
| #include <stdlib.h> /\* for EXIT\_xxx macros \*/  #include <stdio.h> /\* for printf() and getchar() \*/  #include <ctype.h> /\* for isdigit() \*/  void flush\_input\_stream(FILE \* fp);  int main(int argc, char \*argv[]) {  int sum = 0;  printf("Enter digits to be summed up or 0 to exit:\n");  do  {  int c = getchar();  if (EOF == c)  {  printf("Read 'end-of-file', exiting!\n");  break;  }  if ('\n' != c)  {  flush\_input\_stream(stdin);  }  if (!isdigit(c))  {  printf("%c is not a digit! Start over!\n", c);  continue;  }  if ('0' == c)  {  printf("Exit requested.\n");  break;  }  sum += c - '0';  printf("The current sum is %d.\n", sum);  } while (1);  return EXIT\_SUCCESS;  }  void flush\_input\_stream(FILE \* fp)  {  size\_t i = 0;  int c;  while ((c = fgetc(fp)) != '\n' && c != EOF) /\* Pull all until and including the next new-line. \*/  {  ++i;  }  if (0 != i)  {  fprintf(stderr, "Flushed %zu characters from input.\n", i);  }  } |

# CREATE AND INCLUDE HEADER FILES

Trong ngôn ngữ C hiện đại, các tệp header là công cụ quan trọng và cần được thiết kế và sử dụng đúng cách. Chúng cho phép trình biên dịch kiểm tra chéo các phần của chương trình đã được biên dịch một cách độc lập.

Các tệp header khai báo các kiểu, hàm, macro v.v. cần thiết bởi các phần khác trong chương trình. Mã code sử dụng bất kỳ trong số các dịch vụ đó đều bao gồm tệp header. Mã code định nghĩa các dịch vụ đó cũng bao gồm tệp header. Điều này cho phép trình biên dịch kiểm tra xem việc **sử dụng** và **định nghĩa** có khớp nhau không.

## Giới thiệu

Có một số hướng dẫn cần tuân thủ khi tạo và sử dụng các tệp header trong một dự án C:

* ***Khả năng tái sử dụng***

Nếu một tệp header được bao gồm nhiều lần trong một đơn vị biên dịch (TU), nó không nên gây ra lỗi biên dịch.

* ***Tính tự động đóng gói***

Nếu bạn cần các dịch vụ được khai báo trong một tệp header, bạn không nên phải bao gồm bất kỳ tệp header nào khác một cách tường minh.

* ***Tối thiểu hóa***

Bạn không nên thể loại bỏ bất kỳ thông tin nào khỏi một tệp header mà không gây ra lỗi biên dịch.

* ***Bao gồm những gì bạn sử dụng (IWYU)***

Quan trọng hơn đối với C++ hơn là C, nhưng vẫn quan trọng trong C. Nếu mã code trong một TU (gọi là code.c) trực tiếp sử dụng các tính năng được khai báo bởi một tệp header (gọi là "headerA.h"), thì code.c nên #include "headerA.h" trực tiếp, ngay cả khi TU bao gồm một tệp header khác (gọi là "headerB.h") mà tạm thời bao gồm "headerA.h".

Đôi khi, có thể có lý do đủ tốt để vi phạm một hoặc nhiều hướng dẫn này, nhưng bạn nên nhận thức rằng bạn đang vi phạm quy tắc và nhận thức về hậu quả của việc làm như vậy trước khi vi phạm.

## SELF – CONTAINMENT

Các tiêu đề hiện đại self-contained, có nghĩa là một chương trình cần sử dụng các phương tiện được xác định bởi header.h có thể include header đó ( #include "header.h") và không phải lo lắng về việc liệu các header khác có cần được đưa vào trước hay không.

**Khuyến nghị: Các tệp header self-contained.**

**Quy luật lịch sử**

Lịch sử, đây đã là một chủ đề đôi chút gây tranh cãi.

Vào một thời đại nào đó, AT&T Indian Hill C Style and Coding Standards đã nêu rõ:

Các tệp header không nên lồng nhau. Phần đầu của một tệp header nên miêu tả những tệp header khác cần phải được #included để tệp header này có thể hoạt động. Trong trường hợp cực đoan, khi có một số lượng lớn các tệp header cần được included trong một số lượng lớn các tệp nguồn khác nhau, có thể đặt tất cả các #include chung trong một tệp include duy nhất.

Điều này hoàn toàn trái ngược với self-containment.

**Quy luật hiện đại**

Tuy nhiên, kể từ đó, dư luận đã có xu hướng ngược lại. Nếu một tệp nguồn cần sử dụng các phương tiện được khai báo bởi tiêu đề header.h, lập trình viên sẽ có thể viết:

|  |
| --- |
| #include "header.h" |

và (dưới điều kiện chỉ cần đặt các đường dẫn tìm kiếm đúng trên command line), bất kỳ tệp header tiên quyết cần thiết nào sẽ được bao gồm bởi header.h mà không cần thêm thêm tệp header nào vào source file.

Điều này cung cấp tính linh hoạt tốt hơn cho mã nguồn. Điều này cũng bảo vệ mã nguồn khỏi vấn đề "guess why this header was added" mà phát sinh sau khi mã nguồn đã được sửa đổi và sử dụng trong một hoặc hai thập kỷ .

NASA Goddard Space Flight Center (GSFC) coding standards for C là một trong những quy chuẩn hiện đại hơn - nhưng hiện đã khá khó để theo dõi. Nó nêu rõ rằng các tệp header nên được self-contained. Nó cũng cung cấp một cách đơn giản để đảm bảo rằng các tệp header được self-contained: tệp nguồn thực thi cho tệp header nên đặt tệp header làm tệp header đầu tiên. Nếu nó không self-contained, mã nguồn sẽ không biên dịch.

Lý do được GSFC đưa ra bao gồm:

§2.1.1 Giải thích về việc header include

Tiêu chuẩn này yêu cầu tệp header của một đơn vị chứa các câu lệnh #include cho tất cả các tệp header khác cần thiết bởi tệp đơn vị. Đặt #include cho tệp header đầu tiên trong phần thân đơn vị cho phép trình biên dịch xác minh rằng tệp header chứa tất cả các câu lệnh #include cần thiết.

Một thiết kế thay thế, không được phép bởi tiêu chuẩn này, không cho phép tệp header chứa bất kỳ câu lệnh #include nào; tất cả các #include được thực hiện trong các tệp nguồn. Sau đó, tệp header đơn vị phải chứa câu lệnh #ifdef kiểm tra xem các tệp header cần thiết đã được bao gồm theo đúng thứ tự hay chưa.

Một ưu điểm của thiết kế thay thế là danh sách #include trong tệp body chính xác là danh sách phụ thuộc cần thiết trong một tệp makefile và danh sách này được kiểm tra bởi trình biên dịch. Với thiết kế tiêu chuẩn, một công cụ phải được sử dụng để tạo danh sách phụ thuộc. Tuy nhiên, tất cả các môi trường phát triển được khuyến nghị của nhánh đều cung cấp một công cụ như vậy.

Một hạn chế lớn của thiết kế thay thế là nếu danh sách tệp cần thiết cho đơn vị thay đổi, mỗi tệp sử dụng đơn vị đó phải được chỉnh sửa để cập nhật danh sách câu lệnh #include. Ngoài ra, danh sách tệp cần thiết cho một thư viện trình biên dịch có thể khác nhau trên các mục tiêu khác nhau.

Một hạn chế khác của thiết kế thay thế là các tệp header của thư viện trình biên dịch và các tệp bên thứ ba khác phải được sửa đổi để thêm các câu lệnh #ifdef cần thiết.

Do đó, self-containment có nghĩa là:

* Nếu một tệp header header.h cần một tệp header lồng vào mới extra.h, bạn không cần kiểm tra tất cả các tệp nguồn sử dụng header.h để xem liệu bạn cần phải thêm extra.h không.
* Nếu một tệp header header.h không còn cần phải bao gồm một tệp header cụ thể nào đó notneeded.h, bạn không cần phải kiểm tra tất cả các tệp nguồn sử dụng header.h để xem liệu bạn có thể an toàn loại bỏ notneeded.h không (nhưng hãy xem phần "Bao gồm những gì bạn sử dụng").
* Bạn không cần thiết lập chuỗi đúng cho việc bao gồm các tệp header tiên quyết (điều này yêu cầu một sắp xếp thứ tự đỉnh để thực hiện công việc một cách chính xác).

**Kiểm tra self-containment**

Xem "Linking against a static library" để biết về tập lệnh **chkhdr** có thể được sử dụng để kiểm tra khả năng tái sử dụng và self-containment của tệp header.

## TỐI GIẢN

Headers là một cơ chế kiểm tra tính nhất quán quan trọng, nhưng chúng phải càng nhỏ càng tốt. Cụ thể, điều đó có nghĩa là một tiêu đề không nên bao gồm các tiêu đề khác chỉ vì tệp triển khai sẽ cần các tiêu đề khác. Tiêu đề chỉ nên chứa những tiêu đề cần thiết cho người tiêu dùng dịch vụ được mô tả.

Ví dụ: không nên bao gồm tiêu đề dự án <stdio.h>trừ khi một trong các giao diện chức năng sử dụng loại FILE \*(hoặc một trong các loại khác chỉ được xác định trong <stdio.h>). Nếu một giao diện sử dụng size\_t, tiêu đề nhỏ nhất đủ là <stddef.h>. Rõ ràng, nếu một tiêu đề xác định khác size\_tđược bao gồm, thì không cần phải bao gồm <stddef.h>nữa.

Nếu các headers là tối thiểu, thì nó cũng giữ thời gian biên dịch ở mức tối thiểu.

Có thể tạo ra các tiêu đề với mục đích duy nhất là bao gồm rất nhiều tiêu đề khác. Những điều này hiếm khi trở thành một ý tưởng hay về lâu dài vì một số tệp nguồn sẽ thực sự cần tất cả các phương tiện được mô tả bởi tất cả các tiêu đề. Ví dụ: a <standard-c.h>có thể được tạo ra bao gồm tất cả các tiêu đề C tiêu chuẩn - hãy cẩn thận vì một số tiêu đề không phải lúc nào cũng có mặt. Tuy nhiên, rất ít chương trình thực sự sử dụng các tiện ích của <locale.h> hoặc <tgmath.h>.

Xem thêm How to link multiple implementation files in C?

## NOTATION VÀ MISCELLANY

Tiêu chuẩn C nói rằng không có sự khác biệt lớn giữa ký hiệu **#include <header.h>** và **#include "header.h"**.

* [#include <header.h>] tìm kiếm một chuỗi các vị trí được xác định bởi cài đặt để tìm một tệp header được xác định duy nhất bởi chuỗi cụ thể giữa các dấu < và > và gây ra việc thay thế chỉ thị đó bằng toàn bộ nội dung của tệp header. Cách xác định các vị trí hoặc xác định tệp header là do cài đặt quy định.
* [#include "header.h"] gây ra việc thay thế chỉ thị đó bằng toàn bộ nội dung của tệp nguồn được xác định bởi chuỗi cụ thể giữa các dấu "…". Tệp nguồn có tên được tìm kiếm theo cách do cài đặt quy định. Nếu tìm kiếm này không được hỗ trợ hoặc nếu tìm kiếm thất bại, chỉ thị sẽ được xử lý lại như nếu nó đã đọc [#include <header.h>]...

Vì vậy, dạng bằng ngoặc kép có thể tìm kiếm nhiều nơi hơn so với dạng bằng ngoặc nhọn. Tiêu chuẩn chỉ ra ví dụ rằng các tệp tiêu chuẩn nên được bao gồm trong ngoặc nhọn, ngay cả khi việc biên dịch hoạt động nếu bạn sử dụng dấu ngoặc kép thay vì đó. Tương tự, các tiêu chuẩn như POSIX sử dụng định dạng bằng ngoặc nhọn - và bạn cũng nên vậy. Dành tiêu chuẩn bằng ngoặc kép cho các tiêu chuẩn được xác định bởi dự án. Đối với các tiêu chuẩn được xác định bên ngoài (bao gồm cả các tiêu chuẩn từ các dự án khác mà dự án của bạn phụ thuộc vào), định dạng bằng ngoặc nhọn là phù hợp nhất.

Chú ý rằng phải có một khoảng trống giữa #include và tiêu đề, ngay cả khi các trình biên dịch sẽ chấp nhận không có khoảng trống ở đó. Khoảng trống rất hiệu quả.

Nhiều dự án sử dụng biểu thức như:

|  |
| --- |
| #include <openssl/ssl.h>  #include <sys/stat.h>  #include <linux/kernel.h> |

Bạn nên cân nhắc xem có nên sử dụng điều khiển không gian tên đó trong dự án của mình hay không (có lẽ đây là một ý tưởng hay). Bạn nên tránh xa các tên được sử dụng bởi các dự án hiện có (đặc biệt, cả hai sysvà linuxsẽ là những lựa chọn tồi).

Nếu bạn sử dụng điều này, mã của bạn phải cẩn thận và nhất quán trong việc sử dụng ký hiệu.

Không sử dụng ký hiệu **#include "../include/header.h**".

Tệp header hiếm khi định nghĩa biến. Mặc dù bạn sẽ giữ các biến toàn cầu ở mức tối thiểu, nếu bạn cần một biến toàn cầu, bạn sẽ khai báo nó trong tệp header và định nghĩa nó trong một tệp nguồn phù hợp, và tệp nguồn đó sẽ bao gồm tệp header để kiểm tra lại khai báo và định nghĩa, và tất cả các tệp nguồn sử dụng biến đó sẽ sử dụng tệp header để khai báo nó.

Hậu quả: bạn sẽ không khai báo biến toàn cầu trong một tệp nguồn - tệp nguồn chỉ chứa định nghĩa.

Các tệp tiêu đề hiếm khi khai báo staticcác hàm, với ngoại lệ đáng chú ý là static inlinecác hàm sẽ được xác định trong các tiêu đề nếu chức năng đó cần thiết trong nhiều tệp nguồn.

* Các tệp nguồn xác định các biến toàn cục và các hàm toàn cục.
* Các tệp nguồn không khai báo sự tồn tại của các biến hoặc hàm toàn cục; chúng bao gồm tiêu đề khai báo biến hoặc hàm.
* Các tệp tiêu đề khai báo các hàm và biến toàn cục (và các loại và tài liệu hỗ trợ khác).
* Các tệp tiêu đề không xác định các biến hoặc bất kỳ hàm nào ngoại trừ các hàm (static) inline.

**Cross-references**

* Where to document functions in C?
* List of standard header files in C and C++
* Is inline without static or extern ever useful in C99?
* How do I use extern to share variables between source files?
* What are the benefits of a relative path such as "../include/header.h" for a header?
* Header inclusion optimization
* Should I include every header?

## IDEMPOTENCE

Nếu một header file cụ thể được included nhiều lần trong một đơn vị dịch thuật (TU), sẽ không có bất kỳ sự cố biên dịch nào. Điều này được gọi là ''idempotence'; headers của bạn phải idempotent. Hãy nghĩ xem cuộc sống sẽ khó khăn như thế nào nếu bạn phải đảm bảo điều đó #include <stdio.h>chỉ được included vào một lần.

Có hai cách để đạt được idempotence: **header guards** và chỉ thị **#pragma once**.

**Header guards**

Header guardsđơn giản, đáng tin cậy và phù hợp với tiêu chuẩn C. Các dòng không chú thích đầu tiên trong header file phải có dạng:

|  |
| --- |
| #ifndef UNIQUE\_ID\_FOR\_HEADER  #define UNIQUE\_ID\_FOR\_HEADER |

Dòng không chú thích cuối cùng phải là #endif, tùy chọn có chú thích sau đó:

|  |
| --- |
| #endif /\* UNIQUE\_ID\_FOR\_HEADER \*/ |

Tất cả mã hoạt động, bao gồm các chỉ thị #include khác, nên nằm giữa những dòng này.

Mỗi tên phải là duy nhất. Thường, một hệ thống tên như HEADER\_H\_INCLUDED được sử dụng. Một số mã cũ hơn sử dụng một ký hiệu được xác định là Header guards(ví dụ: #ifndef BUFSIZ trong <stdio.h>), nhưng nó không đáng tin cậy như một tên duy nhất. Một tùy chọn có thể là sử dụng tên bảo vệ tiêu chuẩn được tạo ra từ mã MD5 (hoặc mã khác). Bạn nên tránh mô phỏng các kế hoạch được sử dụng bởi các tiêu chuẩn hệ thống mà thường sử dụng các tên đã được dành cho cài đặt - các tên bắt đầu bằng một gạch dưới theo sau bởi một gạch dưới khác hoặc một chữ cái viết hoa.

**The #pragma once Directive**

Một cách khác, một số trình biên dịch hỗ trợ chỉ thị #pragma once có tác dụng tương tự như ba dòng được hiển thị cho header guards.

|  |
| --- |
| #pragma once |

Các trình biên dịch hỗ trợ #pragma once bao gồm MS Visual Studio và GCC và Clang. Tuy nhiên, nếu tính di động là một vấn đề, nên sử dụng cơ chế bảo vệ tiêu chuẩn hoặc sử dụng cả hai. Trình biên dịch hiện đại (các trình biên dịch hỗ trợ C89 hoặc mới hơn) được yêu cầu bỏ qua, mà không có bình luận, các chỉ thị pragma mà họ không nhận ra ('Mọi chỉ thị pragma như vậy không được nhận dạng bởi cài đặt sẽ bị bỏ qua'), nhưng các phiên bản cũ của GCC không hoàn toàn như vậy.

## INCLUDE WHAT YOU USE (IWYU)

Dự án Include What You Use của Google, hoặc IWYU, đảm bảo rằng các tệp nguồn bao gồm tất cả các tiêu chuẩn được sử dụng trong mã code.

Giả sử một tệp nguồn source.c bao gồm một tiêu chuẩn arbitrary.h, tiêu chuẩn này trong lúc đó tình cờ bao gồm tiêu chuẩn freeloader.h, nhưng tệp nguồn cũng một cách rõ ràng và độc lập sử dụng các chức năng từ freeloader.h. Mọi thứ bắt đầu tốt đẹp.

Sau đó, một ngày nào đó, arbitrary.h được thay đổi để người dùng của nó không còn cần các chức năng của freeloader.h nữa. Đột nhiên, source.c ngừng biên dịch - vì nó không đáp ứng được tiêu chí IWYU. Vì mã code trong source.c một cách rõ ràng sử dụng các chức năng của freeloader.h, nó nên đã bao gồm những gì nó sử dụng - nên có một chỉ thị #include "freeloader.h" một cách rõ ràng trong nguồn nguồn. (Tính đơn vị đã đảm bảo không có vấn đề nào.)

Triết lý của IWYU tối đa hóa khả năng rằng mã code tiếp tục biên dịch ngay cả khi có những thay đổi hợp lý được thực hiện đối với các giao diện. Rõ ràng, nếu mã code của bạn gọi một chức năng sau đó bị loại bỏ khỏi giao diện đã công bố, không có lượng chuẩn bị nào có thể ngăn ngừa sự cần thiết của các thay đổi. Đây là lý do tại sao các thay đổi đối với APIs được tránh nếu có thể, và tại sao có chu kỳ bỏ phiên bản qua nhiều phiên bản phát hành, v.v.

Điều này là một vấn đề đặc biệt trong C++ vì các tiêu chuẩn tiêu chuẩn được phép bao gồm lẫn nhau. Tệp nguồn file.cpp có thể bao gồm một tiêu chuẩn header1.h mà trên một nền tảng nào đó bao gồm một tiêu chuẩn tiêu chuẩn khác header2.h. file.cpp có thể sử dụng cả chức năng của header2.h nữa. Ban đầu điều này không phải là một vấn đề - mã code sẽ được biên dịch vì header1.h bao gồm header2.h. Trên nền tảng khác hoặc khi nâng cấp nền tảng hiện tại, header1.h có thể được điều chỉnh để không còn bao gồm header2.h nữa, và sau đó file.cpp sẽ ngừng biên dịch như kết quả. IWYU sẽ phát hiện vấn đề và đề xuất rằng header2.h nên được bao gồm trực tiếp trong file.cpp. Điều này sẽ đảm bảo nó tiếp tục biên dịch. Các yếu tố tương tự cũng áp dụng cho mã code C.

# <CTYPE.H> - CHARACTER CLASSFICATION & CONVERTER

## GIỚI THIỆU

Tiêu chuẩn ctype.h là một phần của thư viện chuẩn C. Nó cung cấp các chức năng để phân loại và chuyển đổi các ký tự.

Tất cả các hàm này đều nhận một tham số, một int mà phải là EOF hoặc có thể biểu diễn như một unsigned char.

Tên của các hàm phân loại có tiền tố là 'is'. Mỗi giá trị trả về một giá trị số nguyên khác không (TRUE) nếu ký tự được truyền cho nó thỏa mãn điều kiện liên quan. Nếu điều kiện không được thỏa mãn thì hàm trả về giá trị 0 (FALSE).

Các hàm phân loại này hoạt động như được hiển thị, giả sử ngôn ngữ C mặc định:

|  |
| --- |
| int a;  int c = 'A';  a = isalpha(c); /\* Kiểm tra xem c có phải là chữ cái (A-Z, a-z), trả về giá trị khác không ở đây. \*/  a = isalnum(c); /\* Kiểm tra xem c có phải là chữ cái hoặc chữ số (A-Z, a-z, 0-9), trả về giá trị khác không ở đây. \*/  a = iscntrl(c); /\* Kiểm tra xem c có phải là ký tự điều khiển (0x00-0x1F, 0x7F), trả về giá trị zero ở đây. \*/  a = isdigit(c); /\* Kiểm tra xem c có phải là số (0-9), trả về giá trị zero ở đây. \*/  a = isgraph(c); /\* Kiểm tra xem c có biểu diễn đồ họa (bất kỳ ký tự in nào trừ dấu cách),trả về giá trị khác không ở đây. \*/  a = islower(c); /\* Kiểm tra xem c có phải là chữ thường (a-z), trả về giá trị zero ở đây. \*/  a = isprint(c); /\* Kiểm tra xem c có phải là bất kỳ ký tự có thể in (bao gồm cả dấu cách), trả về giá trị khác không ở đây. \*/  a = isupper(c); /\* Kiểm tra xem c có phải là chữ in hoa (a-z), trả về giá trị zero ở đây. \*/  a = ispunct(c); /\* Kiểm tra xem c có phải là ký tự dấu câu, trả về giá trị zero ở đây. \*/  a = isspace(c); /\* Kiểm tra xem c có phải là ký tự khoảng trắng, trả về giá trị zero ở đây. \*/  a = isupper(c); /\* Kiểm tra xem c có phải là chữ in hoa (A-Z), trả về giá trị khác không ở đây. \*/  a = isxdigit(c); /\* Kiểm tra xem c có phải là chữ số thập lục phân (A-F, a-f, 0-9), trả về giá trị khác không ở đây. \*/ |

Phiên bản ≥ C99

|  |
| --- |
| a = isblank(c); /\* Kiểm tra xem c có phải là ký tự trống (khoảng trắng hoặc tab), trả về giá trị khác không ở đây. \*/ |

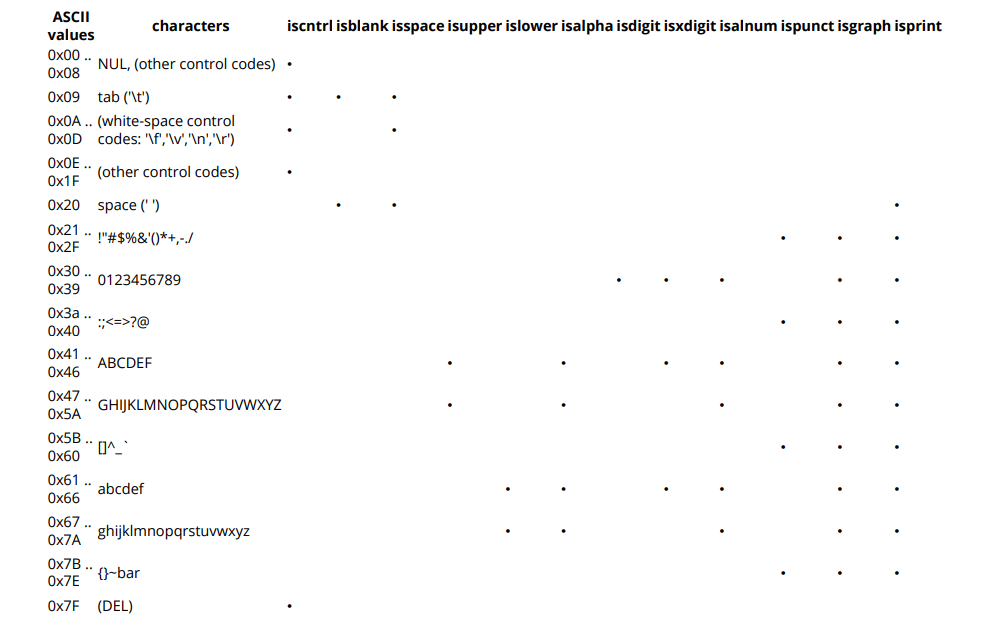
Có hai hàm chuyển đổi. Chúng được đặt tên bằng tiền tố 'to'. Các hàm này nhận cùng đối số như các hàm ở trên. Tuy nhiên, giá trị trả về không phải là giá trị zero hoặc không phải giá trị không mà là đối số được truyền vào đã được thay đổi theo một cách nào đó.

Các hàm chuyển đổi này hoạt động như sau, giả sử với ngữ cảnh C locale mặc định:

|  |
| --- |
| int a;  int c = 'A';  /\* Chuyển đổi c thành chữ thường (a-z).  \* Nếu không thể chuyển đổi, giá trị không thay đổi được trả về.  \* Trả về 'a' ở đây.  \*/  a = tolower(c);  /\* Chuyển đổi c thành chữ in hoa (A-Z).  \* Nếu không thể chuyển đổi, giá trị không thay đổi được trả về.  \* Trả về 'A' ở đây.  \*/  a = toupper(c); |

Thông tin dưới đây được trích từ trang web cplusplus.com, mô tả cách 127 ký tự trong bộ mã ASCII ban đầu được xem xét bởi mỗi hàm phân loại (một dấu • chỉ ra rằng hàm trả về khác không cho ký tự đó):

Giá trị ASCII của các ký tự



## PHÂN LOẠI CÁC KÍ TỰ ĐỌC TỪ MỘT LUỒNG

|  |
| --- |
| #include <ctype.h>  #include <stdio.h>  typedef struct {  size\_t space;  size\_t alnum;  size\_t punct;  } chartypes;  chartypes classify(FILE \*f) {  chartypes types = { 0, 0, 0 };  int ch;    while ((ch = fgetc(f)) != EOF) {  types.space += !!isspace(ch);  types.alnum += !!isalnum(ch);  types.punct += !!ispunct(ch);  }    return types;  } |

Hàm classify đọc các ký tự từ một luồng (stream) và đếm số lượng dấu cách, ký tự chữ và ký tự dấu câu. Nó tránh một số lỗi phổ biến.

* Khi đọc một ký tự từ luồng, kết quả được lưu dưới dạng int, vì không thì sẽ có một sự mơ hồ giữa việc đọc EOF (ký hiệu cuối tập tin) và một ký tự có cùng mẫu bit.
* Các hàm phân loại ký tự (ví dụ isspace) mong đợi đối số của chúng có thể biểu diễn dưới dạng unsigned char hoặc giá trị của macro EOF. Vì vậy, điều này chính xác là điều mà fgetc trả về, không cần phải chuyển đổi ở đây.
* Giá trị trả về của các hàm phân loại ký tự chỉ phân biệt giữa zero (nghĩa là sai) và không phải zero (nghĩa là đúng). Đối với việc đếm số lần xuất hiện, giá trị này cần được chuyển đổi thành 1 hoặc 0, điều này được thực hiện bằng cách phủ định kép, !!.

## PHÂN LOẠI CÁC KÍ TỰ ĐỌC TỪ MỘT CHUỖI

|  |
| --- |
| #include <ctype.h>  #include <stddef.h>  typedef struct {  size\_t space;  size\_t alnum;  size\_t punct;  } chartypes;  chartypes classify(const char \*s) {  chartypes types = { 0, 0, 0 };  const char \*p;  for (p = s; \*p != '\0'; p++) {  types.space += !!isspace((unsigned char)\*p);  types.alnum += !!isalnum((unsigned char)\*p);  types.punct += !!ispunct((unsigned char)\*p);  }  return types;  } |

Hàm classify kiểm tra tất cả các ký tự từ một chuỗi và đếm số lượng ký tự dấu cách, ký tự chữ và ký tự dấu câu. Nó tránh một số rủi ro.

* Các hàm phân loại ký tự (ví dụ isspace) mong đợi đối số của chúng có thể được biểu diễn dưới dạng unsigned char hoặc giá trị của macro EOF.
* Biểu thức \*p có kiểu char và do đó phải được chuyển đổi để phù hợp với văn bản ở trên.
* Kiểu char được xác định tương đương với hoặc signed char hoặc unsigned char.
* Khi char tương đương với unsigned char, không có vấn đề gì, vì mọi giá trị có thể có của kiểu char có thể được biểu diễn dưới dạng unsigned char.
* Khi char tương đương với signed char, nó phải được chuyển đổi thành unsigned char trước khi được truyền vào các hàm phân loại ký tự. Và mặc dù giá trị của ký tự có thể thay đổi do việc chuyển đổi này, đây chính xác là điều các hàm này mong đợi.
* Giá trị trả về của các hàm phân loại ký tự chỉ phân biệt giữa số không (nghĩa là sai) và số khác không (nghĩa là đúng). Để đếm số lần xuất hiện, giá trị này cần được chuyển đổi thành 1 hoặc 0, điều này được thực hiện bằng cách phủ định kép, !!.

# SIDE EFFECTS

## TOÁN TỬ TĂNG/GIẢM TRƯỚC/SAU

Trong C, có hai toán tử một ngôi - '++' và '--' - là nguồn gây nhầm lẫn rất phổ biến. Toán tử ++ được gọi là toán tử tăng và toán tử -- được gọi là toán tử giảm. Cả hai đều có thể được sử dụng ở dạng **tiền tố** hoặc **hậu tố**. Cú pháp cho dạng tiền tố của toán tử ++ là **++toán\_hạng** và cú pháp cho dạng hậu tố là **toán\_hạng++**. Khi được sử dụng ở dạng tiền tố, toán\_hạng được tăng trước 1 đơn vị và giá trị kết quả của toán\_hạng được sử dụng trong việc đánh giá biểu thức. Xem xét ví dụ sau:

|  |
| --- |
| int n, x = 5;  n = ++x; /\* x được tăng lên 1 (x=6), và kết quả được gán cho n (n=6) \*/  /\* Đây là một dạng ngắn gọn cho hai câu lệnh: \*/  /\* x = x + 1; \*/  /\* n = x; \*/ |

Khi được sử dụng ở dạng hậu tố, giá trị hiện tại của toán hạng được sử dụng trong biểu thức và sau đó giá trị của toán hạng được tăng thêm 1. Hãy xem xét ví dụ sau:

|  |
| --- |
| int n, x = 5;  n = x++; /\* giá trị của x (5) được gán trước cho n (5), sau đó x được tăng lên 1; x (6) \*/  /\* Đây là một dạng ngắn gọn cho hai câu lệnh: \*/  /\* n = x; \*/  /\* x = x + 1; \*/ |

Cách làm việc của toán tử giảm -- có thể được hiểu tương tự.

Đoạn mã sau đây mô tả những gì mỗi toán tử làm:

|  |
| --- |
| int main(int argc, char\* argv[]) {  int a, b, x = 42;  a = ++x; /\* a và x là 43 \*/  b = x++; /\* b là 43, x là 44 \*/  a = x--; /\* a là 44, x là 43 \*/  b = --x; /\* b và x là 42 \*/  return 0;  } |

Từ trên, rõ ràng rằng các toán tử hậu trả về giá trị hiện tại của một biến và sau đó thay đổi nó, nhưng các toán tử tiền trả về giá trị đã được sửa đổi của biến.

Trong tất cả các phiên bản của C, thứ tự đánh giá của các toán tử tiền và hậu không được xác định, do đó đoạn mã sau có thể trả về kết quả không mong đợi:

|  |
| --- |
| int main(int argc, char\* argv[]) {  int a, x = 42;  a = x++ + x; /\* sai \*/  a = x + x; /\* đúng \*/  ++x;  int ar[10];  x = 0;  ar[x] = x++; /\* sai \*/  ar[x++] = x; /\* sai \*/  ar[x] = x; /\* đúng \*/  ++x;  return 0;  } |

Chú ý rằng cũng là thực hành tốt khi sử dụng toán tử tiền thay vì hậu khi được sử dụng một mình trong một câu lệnh. Xem xét mã trên cho điều này.

Lưu ý rằng khi một hàm được gọi, tất cả các hiệu ứng phụ lên đối số phải diễn ra trước khi hàm chạy.

|  |
| --- |
| int foo(int x) {  return x;  }  int main(int argc, char\* argv[]) {`  int a = 42;  int b = foo(a++); /\* Điều này trả về 43, ngay cả khi có vẻ như nó nên trả về 42 \*/  return 0;  } |

# MULTI – CHARACTER CHARACTER SEQUENCE

## TRIGRAPHS

Các ký hiệu [ ] { } ^ \ | ~ # thường được sử dụng trong các chương trình C. Nhưng vào cuối thập kỷ 1980, có các bộ mã đang được sử dụng (ví dụ như các biến thể ISO 646 tại các quốc gia Scandinavia), trong đó các vị trí ký tự ASCII này được sử dụng cho các ký tự biến thể của ngôn ngữ quốc gia (ví dụ: £ thay cho # ở Vương quốc Anh; Æ Å æ å ø Ø thay cho { } { } | \ ở Đan Mạch; không có ~ trong EBCDIC). Điều này làm cho việc viết mã C trên các máy sử dụng các bộ mã này trở nên khó khăn.

Để giải quyết vấn đề này, chuẩn C đề xuất sử dụng các kết hợp ba ký tự để tạo ra một ký tự đơn gọi là trigraph. Một trigraph là một dãy gồm ba ký tự, trong đó hai ký tự đầu tiên là dấu hỏi ??. Mỗi trigraph tương ứng với một ký tự thay thế. Ví dụ, trigraph ??= tương ứng với ký tự #, trigraph ??< tương ứng với ký tự {.

Phần sau đây là một ví dụ đơn giản sử dụng trigraph thay vì #, { và }:

|  |
| --- |
| ??=include <stdio.h>  int main()  ??<  printf("Hello World!\n");  ??> |

Đoạn mã trên sẽ được xử lý bởi bộ tiền xử lý C bằng cách thay thế trigraph bằng các ký tự đơn tương ứng như khi mã đã được viết:

|  |
| --- |
| #include <stdio.h>  int main()  {  printf("Hello World!\n");  } |

|  |  |
| --- | --- |
| **Trigraph** | **Equivalent** |
| ??= | # |
| ??/ | \ |
| ??' | ^ |
| ??( | [ |
| ??) | ] |
| ??! | | |
| ??< | { |
| ??> | } |
| ??- | ~ |

Lưu ý rằng trigraphs có thể gây rối vì, ví dụ, ??/ là một dấu gạch chéo ngược và có thể ảnh hưởng đến ý nghĩa của các dòng tiếp tục trong các chú thích, và chúng phải được nhận dạng trong chuỗi và ký tự literal (ví dụ: '??/??/' là một ký tự duy nhất, là một dấu gạch chéo ngược).

## DIGRAPHS

Phiên bản ≥ C99

Năm 1994, các thay thế dễ đọc hơn cho năm trigraph phổ biến đã được cung cấp. Chúng sử dụng chỉ hai ký tự và được gọi là digraph. Khác với trigraph, digraph là các token. Nếu một digraph xuất hiện trong một token khác (ví dụ: chuỗi hoặc hằng ký tự) thì nó sẽ không được xử lý như một digraph mà vẫn giữ nguyên như cũ.

Dưới đây là sự khác biệt trước và sau khi xử lý digraphs.

|  |
| --- |
| #include <stdio.h>  int main(int argc, char\* argv[]) {  <%  printf("Hello %> World!\n"); /\* Note that the string contains a digraph \*/  %> |

Sẽ được xử lý tương tự như:

|  |
| --- |
| #include <stdio.h>  int main(int argc, char\* argv[]) {  printf("Hello %> World!\n"); /\* Note the unchanged digraph within the string. \*/  } |

|  |  |
| --- | --- |
| **Digraph** | **Equivalent** |
| <: | [ |
| :> | ] |
| <% | { |
| %> | } |
| %: | # |

# RÀNG BUỘC (CONSTRAINT)

## TÊN BIẾN TRÙNG LẶP TRONG CÙNG PHẠM VI

Một ví dụ về ràng buộc được thể hiện trong tiêu chuẩn C là có hai biến cùng tên được khai báo trong cùng một phạm vi1), ví dụ:

|  |
| --- |
| void foo(int bar) {  int var;  double var;  } |

Mã này vi phạm ràng buộc và phải tạo ra một thông báo chẩn đoán trong quá trình biên dịch. Điều này rất hữu ích so với hành vi không xác định vì nhà phát triển sẽ được thông báo về vấn đề trước khi chương trình được chạy, tránh nguy cơ xảy ra bất cứ điều gì.

Do đó, các ràng buộc thường là các lỗi dễ dàng phát hiện trong quá trình biên dịch như ví dụ trên, các vấn đề dẫn đến hành vi không xác định nhưng khó hoặc không thể phát hiện trong quá trình biên dịch không phải là các ràng buộc.

Đoạn văn chính xác:

Phiên bản = C99

Nếu một định danh không có liên kết, thì không nên có nhiều hơn một khai báo của định danh (trong một bộ khai báo hoặc chỉ mục kiểu) cùng một phạm vi và trong cùng không gian tên, trừ trường hợp của các thẻ như đã quy định trong 6.7.2.3.

## TOÁN TỬ SỐ HỌC MỘT NGÔI

Các toán tử số học một ngôi + và - chỉ có thể sử dụng trên các kiểu số học, do đó nếu ví dụ cố gắng sử dụng chúng trên một cấu trúc thì chương trình sẽ tạo ra một thông báo chẩn đoán, ví dụ:

|  |
| --- |
| struct foo  {  bool bar;  };  void baz(void)  {  struct foo testStruct;  -testStruct; /\* Điều này vi phạm ràng buộc nên phải tạo ra một thông báo chẩn đoán \*/  } |

# INLINING

## HÀM INLINING ĐƯỢC SỬ DỤNG TRONG HƠN MỘT TỆP NGUỒN

Đối với các hàm nhỏ mà thường được gọi thường xuyên, chi phí liên quan đến việc gọi hàm có thể chiếm một phần đáng kể của tổng thời gian thực thi của hàm đó. Một cách để cải thiện hiệu suất là loại bỏ chi phí gọi hàm.

Trong ví dụ này, chúng ta sử dụng bốn hàm (cộng với main()) trong ba tệp nguồn. Hai trong số đó (plusfive() và timestwo()) mỗi cái được gọi bởi hai cái khác trong "source1.c" và "source2.c". main() được bao gồm để chúng ta có một ví dụ hoạt động.

**main.c:**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include "headerfile.h"  int main(void) {  int start = 3;  int intermediate = complicated1(start);  printf("First result is %d\n", intermediate);  intermediate = complicated2(start);  printf("Second result is %d\n", intermediate);  return 0;  } |

**source1.c:**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include "headerfile.h"  int complicated1(int input) {  int tmp = timestwo(input);  tmp = plusfive(tmp);  return tmp;  } |

**source2.c:**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include "headerfile.h"  int complicated2(int input) {  int tmp = plusfive(input);  tmp = timestwo(tmp);  return tmp;  } |

**headerfile.h:**

|  |
| --- |
| #ifndef HEADERFILE\_H  #define HEADERFILE\_H  int complicated1(int input);  int complicated2(int input);  inline int timestwo(int input) {  return input \* 2;  }  inline int plusfive(int input) {  return input + 5;  }  #endif |

Hàm timestwo và plusfive được gọi bởi cả complicated1 và complicated2, chúng nằm trong các "đơn vị dịch" hoặc các tệp nguồn khác nhau. Để sử dụng chúng theo cách này, chúng ta phải định nghĩa chúng trong tệp header.

Biên dịch như sau, giả sử gcc:

|  |
| --- |
| cc -O2 -std=c99 -c -o main.o main.c  cc -O2 -std=c99 -c -o source1.o source1.c  cc -O2 -std=c99 -c -o source2.o source2.c  cc main.o source1.o source2.o -o main |

Chúng ta sử dụng tùy chọn tối ưu hóa -O2 vì một số trình biên dịch không tích hợp mà không có tùy chọn tối ưu hóa được bật.

Tác động của từ khóa inline là biểu tượng hàm cụ thể không được phát ra vào tệp đối tượng. Nếu không có inline, một lỗi sẽ xảy ra ở dòng cuối cùng, nơi chúng ta đang liên kết các tệp đối tượng để tạo thành chương trình thực thi cuối cùng. Nếu không có inline, biểu tượng cùng tên sẽ được định nghĩa trong cả hai tệp .o và lỗi "biểu tượng được định nghĩa nhiều lần" sẽ xảy ra.

Trong những tình huống mà biểu tượng thực sự cần thiết, có hai khả năng để xử lý vấn đề đó. Khả năng đầu tiên là thêm một khai báo extern bổ sung của các hàm được tích hợp trong chính xác một trong các tệp .c. Vì vậy, thêm dòng sau vào source1.c:

|  |
| --- |
| extern int timestwo(int input);  extern int plusfive(int input); |

Khả năng thứ hai là định nghĩa hàm với static inline thay vì inline. Phương pháp này có điểm bất lợi là cuối cùng một bản sao của hàm cụ thể có thể được tạo ra trong mọi tệp đối tượng được tạo ra bằng tệp header này.

# UNIONS

## SỬ DỤNG UNIONS ĐỂ TÁI TẠO CÁC GIÁ TRỊ

Một số phiên bản C cho phép mã viết vào một thành viên của kiểu Union sau đó đọc từ một thành viên khác để thực hiện một loại ép kiểu diễn giải lại (phân tích cú pháp kiểu mới như biểu diễn bit của kiểu cũ).

Tuy nhiên, điều quan trọng cần lưu ý là điều này không được phép theo tiêu chuẩn C hiện tại hoặc quá khứ và sẽ dẫn đến hành vi không xác định, không hơn không kém, đây là một tiện ích mở rộng rất phổ biến được cung cấp bởi trình biên dịch (vì vậy hãy kiểm tra tài liệu trình biên dịch của bạn nếu bạn định làm điều này) .

Một ví dụ thực tế về kỹ thuật này là thuật toán "Fast Inverse Square Root," mà dựa vào các chi tiết về số dấu phẩy động IEEE 754 để thực hiện tính căn bình phương nghịch đảo nhanh hơn so với việc sử dụng phép tính số dấu phẩy động. Thuật toán này có thể được thực hiện thông qua việc chuyển kiểu con trỏ (rất nguy hiểm và vi phạm quy tắc đa loại nghiêm ngặt) hoặc thông qua union (vẫn là hành vi không xác định nhưng hoạt động trên nhiều trình biên dịch):

|  |
| --- |
| union floatToInt  {  int32\_t intMember;  float floatMember; /\* Float phải là 32 bit IEEE 754 để tính năng này hoạt động \*/  };  float inverseSquareRoot(float input)  {  union floatToInt x;  int32\_t i;  float f;  x.floatMember = input; /\* Gán vào thành viên kiểu float \*/  i = x.intMember; /\* Đọc trở lại từ thành viên kiểu integer \*/  i = 0x5f3759df - (i >> 1);  x.intMember = i; /\* Gán vào thành viên kiểu integer \*/  f = x.floatMember; /\* Đọc trở lại từ thành viên kiểu float \*/  f = f \* (1.5f - input \* 0.5f \* f \* f);  return f \* (1.5f - input \* 0.5f \* f \* f);  } |

Kỹ thuật này đã được sử dụng rộng rãi trong đồ họa máy tính và trò chơi trước đây do tốc độ nhanh hơn so với sử dụng các phép toán dấu phẩy động và rất thỏa hiệp, mất đi một số độ chính xác và rất khó di chuyển để đổi lấy tốc độ.

## VIẾT VÀO MỘT PHẦN TỬ UNION VÀ ĐỌC TỪ PHẦN TỬ KHÁC

Các thành viên của một liên hợp chia sẻ cùng một vùng bộ nhớ. Điều này có nghĩa rằng việc ghi vào một thành viên sẽ ghi đè lên dữ liệu trong tất cả các thành viên khác và việc đọc từ một thành viên sẽ dẫn đến việc đọc cùng dữ liệu như việc đọc từ tất cả các thành viên khác. Tuy nhiên, vì các thành viên của liên hợp có thể có các loại và kích thước khác nhau, dữ liệu được đọc có thể được diễn giải khác nhau, xem thêm

|  |
| --- |
| http://stackoverflow.com/documentation/c/1119/structs-and-unions/9399/using-unions-to-reinterpret-values |

Ví dụ đơn giản dưới đây minh họa một union có hai phần tử, cả hai đều có cùng kiểu dữ liệu. Nó cho thấy việc ghi vào phần tử m\_1 dẫn đến giá trị được ghi được đọc từ phần tử m\_2 và việc ghi vào phần tử m\_2 dẫn đến việc giá trị được ghi được đọc từ phần tử m\_1.

|  |
| --- |
| #include <stdio.h>  union my\_union /\* Định nghĩa liên hợp \*/  {  int m\_1;  int m\_2;  };  int main (void)  {  union my\_union u; /\* Khai báo union \*/  u.m\_1 = 1; /\* Ghi vào m\_1 \*/  printf("u.m\_2: %i\n", u.m\_2); /\* Đọc từ m\_2 \*/  u.m\_2 = 2; /\* Ghi vào m\_2 \*/  printf("u.m\_1: %i\n", u.m\_1); /\* Đọc từ m\_1 \*/  return 0;  } |

## SỰ KHÁC BIỆT GIỮA STRUCT VÀ UNION

Đoạn mã sau minh họa rằng các phần tử của union chia sẻ bộ nhớ trong khi các phần tử của struct không chia sẻ bộ nhớ.

|  |
| --- |
| #include <stdio.h>  #include <string.h>  union My\_Union  {  int variable\_1;  int variable\_2;  };  struct My\_Struct  {  int variable\_1;  int variable\_2;  };  int main (void)  {  union My\_Union u;  struct My\_Struct s;  u.variable\_1 = 1;  u.variable\_2 = 2;  s.variable\_1 = 1;  s.variable\_2 = 2;  printf ("u.variable\_1: %i\n", u.variable\_1);  printf ("u.variable\_2: %i\n", u.variable\_2);  printf ("s.variable\_1: %i\n", s.variable\_1);  printf ("s.variable\_2: %i\n", s.variable\_2);  printf ("sizeof (union My\_Union): %i\n", sizeof (union My\_Union));  printf ("sizeof (struct My\_Struct): %i\n", sizeof (struct My\_Struct));  return 0;  } |

# THREADS (NATIVE)

## Khởi tạo một thread

Trong hầu hết các trường hợp, tất cả dữ liệu mà nhiều luồng truy cập nên được khởi tạo trước khi các luồng được tạo ra. Điều này đảm bảo rằng tất cả các luồng bắt đầu với một trạng thái rõ ràng và không xảy ra tình huống cạnh tranh.

Nếu điều này không khả thi, ta có thể sử dụng once\_flag và call\_once.

|  |
| --- |
| #include <threads.h>  #include <stdlib.h>  double const\* Big = 0;  static once\_flag onceBig = ONCE\_INIT;  void destroyBig(void) {  free((void\*)Big);  }  void initBig(void) {  double\* b = malloc(largeNum);  if (!b) {  perror("allocation failed for Big");  exit(EXIT\_FAILURE);  }  initializeBigWithSophisticatedValues(largeNum, b);  Big = b;  atexit(destroyBig);  at\_quick\_exit(destroyBig);  }  int myThreadFunc(void\* a) {  call\_once(&onceBig, initBig);  ...  return 0;  } |

once\_flag được sử dụng để điều phối các luồng khác nhau có thể muốn khởi tạo cùng dữ liệu Big. Cuộc gọi đến call\_once đảm bảo rằng:

* initBig được gọi chính xác một lần.
* call\_once chặn cho đến khi cuộc gọi initBig đã được thực hiện, bởi cùng một luồng hoặc một luồng khác.

Ngoài việc cấp phát, một việc điển hình để thực hiện trong một hàm được gọi một lần như vậy là khởi tạo động của các cấu trúc dữ liệu kiểm soát luồng như mtx\_t hoặc cnd\_t không thể được khởi tạo tĩnh, bằng cách sử dụng mtx\_init hoặc cnd\_init tương ứng.

## Bắt đầu nhiều luồng

|  |
| --- |
| #include <stdio.h>  #include <threads.h>  #include <stdlib.h>  struct my\_thread\_data {  double factor;  };  int my\_thread\_func(void\* a) {  struct my\_thread\_data\* d = a;  // thực hiện điều gì đó với d  printf("Chúng tôi tìm thấy %g\n", d->factor);  // trả về mã thành công hoặc mã lỗi  return d->factor > 1.0;  }  int main(int argc, char\* argv[argc+1]) {  unsigned n = 4;  if (argc > 1) n = strtoull(argv[1], 0, 0);  // dự trữ không gian cho các đối số cho các luồng  struct my\_thread\_data D[n]; // không thể khởi tạo  for (unsigned i = 0; i < n; ++i) {  D[i] = (struct my\_thread\_data){ .factor = 0.5\*i, };  }  // dự trữ không gian cho ID của các luồng  thrd\_t id[4];  // khởi chạy các luồng  for (unsigned i = 0; i < n; ++i) {  thrd\_create(&id[i], my\_thread\_func, &D[i]);  }  // Chờ tất cả các luồng kết thúc, nhưng bỏ đi các giá trị trả về của chúng  for (unsigned i = 0; i < n; ++i) {  thrd\_join(id[i], 0);  }  return EXIT\_SUCCESS;  } |

# ĐA LUỒNG (MULTITHREADING)

**Đa luồng (Multithreading)** là một form chuyên dụng của đa nhiệm (multitasking) là tính năng cho phép máy tính của bạn chạy hai hoặc nhiều chương trình đồng thời. Nói chung, có hai kiểu đa nhiệm là: process-based (dựa trên tiến trình) và thread-based (dựa trên luồng).

Đa nhiệm dựa trên tiến trình xử lý việc thực thi đồng thời của các chương trình. Đa nhiệm dựa trên luồng xử lý việc thực thi đồng thời các phần của cùng một chương trình.

Một chương trình đa luồng chứa hai hoặc nhiều phần mà có thể chạy đồng thời. Mỗi phần của chương trình đó được gọi là một thread, và mỗi thread định nghĩa một path riêng biệt của sự thực thi.

C++ không chứa bất kỳ hỗ trợ có sẵn nào cho các ứng dụng đa luồng. Thay vào đó, nó dựa hoàn toàn vào Hệ điều hành để cung cấp tính năng này.

Chương này giả sử bạn đang làm việc trên Hệ điều hành Linux và chúng tôi đang chuẩn bị viết chương trình đa luồng trong C++ bởi sử dụng POSIX. POSIX Threads hoặc Pthreads cung cấp API mà có sẵn trên nhiều hệ thống như FreeBSD, NetBSD, GNU/Linux, Mac OS X và Solaris.

## Tạo Thread trong C++

Đây là chương trình chúng ta sử dụng để tạo một POSIX thread:

|  |
| --- |
| #include <pthread.h>  pthread\_create (thread, attr, start\_routine, arg) |

Ở đây, **pthread\_create** tạo một thread mới và làm nó có thể thực thi. Chương trình này có thể được gọi bất cứ thời điểm nào ở bất cứ đâu trong code của bạn. Dưới đây là miêu tả các tham số:

|  |  |
| --- | --- |
| **Tham số** | **Miêu tả** |
| thread | Một định danh duy nhất cho thread mới được trả về bởi chương trình con |
| attr | Một thuộc tính mà có thể được sử dụng để thiết lập các thuộc tính của thread. Bạn có thể xác định một đối tượng thuộc tính thread, hoặc NULL cho các giá trị mặc định |
| start\_routine | Chương trình C++ mà thread này sẽ thực thi một khi nó được tạo |
| arg | Một tham số đơn mà có thể được truyền tới start\_routine. Nó phải được truyền bởi tham chiếu dạng một con trỏ của kiểu void. NULL có thể được sử dụng nếu không có tham số nào được truyền |

Số thread tối đa có thể được tạo bởi một tiến trình là phụ thuộc vào trình triển khai (Implementation). Một khi được tạo, các thread là ngang hàng, và có thể tạo các thread khác. Không có sự phụ thuộc giữa các thread trong C++.

## Kết thúc Thread trong C++

Chương trình sau được sử dụng để kết thúc một POSIX thread trong C++:

#include <pthread.h>

pthread\_exit (status)

Ở đây **pthread\_exit** được sử dụng để kết thúc một thread. Chương trình pthread\_exit() được gọi sau khi một thread đã hoàn thành công việc của nó và không cần thiết phải tồn tại nữa.

Nếu main() kết thúc trước các thread nó đã tạo, và kết thúc chương trình pthread\_create(), thì các thread khác sẽ tiếp tục thực thi. Nếu không thì, chúng sẽ tự động được kết thúc khi main() hoàn thành.

**Ví dụ**

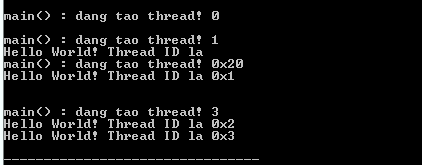
Ví dụ đơn giản sau tạo 5 thread với chương trình pthread\_create(). Mỗi thread in một thông báo "Hello World!", và sau đó kết thúc với một lời gọi tới pthread\_exit() trong C++:

|  |
| --- |
| #include <iostream>  #include <cstdlib>  #include <pthread.h>using namespace std;#define SO\_THREAD     4void \*InLoiChao(void \*threadid)  {     cout << "Hello World! Thread ID la " << threadid << endl;     pthread\_exit(NULL);  }  int main(int argc, char\* argv[]) {     pthread\_t threads[SO\_THREAD];     int rc;     int i;     for( i=0; i < SO\_THREAD; i++ ){        cout << "\nmain() : dang tao thread! " << i << endl;        rc = pthread\_create(&threads[i], NULL,                            InLoiChao, (int \*)i);        if (rc){           cout << "\nError: Khong the tao thread!" << rc << endl;           exit(-1);        }     }     pthread\_exit(NULL);  } |

Nếu bạn đang sử dụng command promt để biên dịch chương trình, thì bạn sử dụng thư viện **–lpthread** như sau:

|  |
| --- |
| $gcc test.cpp -lpthread |

Nếu không, biên dịch và chạy chương trình C++ trên sẽ cho kết quả sau:



## Truyền tham số tới Thread trong C++

Ví dụ này minh họa cách truyền nhiều tham số thông qua một cấu trúc. Bạn có thể truyền bất kỳ kiểu dữ liệu nào trong một Thread callback, bởi vì nó trỏ tới void như được giải thích trong ví dụ sau:

|  |
| --- |
| #include <iostream>  #include <cstdlib>  #include <pthread.h>using namespace std;#define SO\_THREAD     5struct du\_lieu{     int  id;     char \*thongdiep;  };void \*InLoiChao(void \*thamso)  {     struct du\_lieu \*data;   data = (struct du\_lieu \*) thamso;   cout << "\nThread ID la: " << data->id ;     cout << "\nThong diep: " << data->thongdiep << endl;   pthread\_exit(NULL);  }  int main(int argc, char\* argv[]) {     pthread\_t threads[SO\_THREAD];     struct du\_lieu td[SO\_THREAD];     int rc;     int i;   for( i=0; i < SO\_THREAD; i++ ){        cout <<"\nmain() : dang tao thread! " << i << endl;        td[i].id = i;        td[i].thongdiep = "\nDay la thong diep";        rc = pthread\_create(&threads[i], NULL,                            InLoiChao, (void \*)&td[i]);        if (rc){           cout << "\nError: Khong the tao thread! " << rc << endl;           exit(-1);        }     }     pthread\_exit(NULL);  } |

Biên dịch và chạy chương trình C++ trên sẽ cho kết quả sau:



## Kết hợp và Tháo gỡ các Thread trong C++

Hai cú pháp sau được sử dụng để kết hợp (joining) hoặc tháo gỡ (detaching) các Thread trong C++:

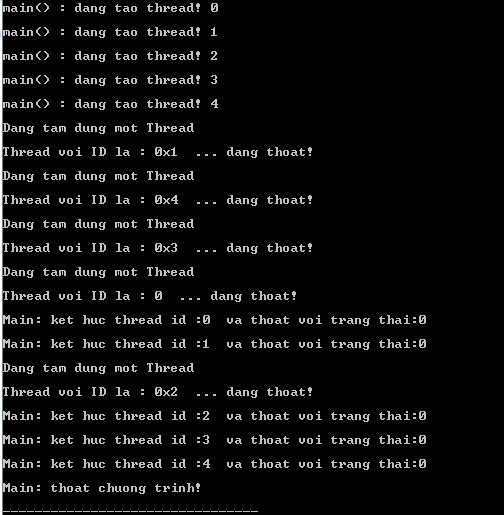
|  |
| --- |
| pthread\_join (threadid, status)  pthread\_detach (threadid) |

Chương trình con pthread\_join() đóng khối thread đang gọi tới khi threadid kết thúc. Khi một thread được tạo, một trong các thuộc tính định nghĩa nó là joinable hoặc detached. Chỉ các thread được tạo với dạng joinable có thể được kết hợp. Nếu một thread được tạo với dạng detached, nó không bao giờ được kết hợp.

Ví dụ sau minh họa cách đợi cho các thread kết thúc bởi sử dụng chương trình kết hợp Pthread trong C++:

|  |
| --- |
| #include <iostream>  #include <cstdlib>  #include <pthread.h>  #include <unistd.h>using namespace std;#define SO\_THREAD     5void \*wait(void \*tid)  {     int i;       sleep(1);     cout << "\nDang tam dung mot Thread " << endl;     cout << "\nThread voi ID la : " << tid << "  ... dang thoat! " << endl;     pthread\_exit(NULL);  }  int main(int argc, char\* argv[]) {     int rc;     int i;     pthread\_t threads[SO\_THREAD];     pthread\_attr\_t attr;     void \*status;   // khoi tao va thiet lap mot thread co the ket hop     pthread\_attr\_init(&attr);     pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_JOINABLE);   for( i=0; i < SO\_THREAD; i++ ){        cout << "\nmain() : dang tao thread! " << i << endl;        rc = pthread\_create(&threads[i], NULL, wait, (void \*)i );        if (rc){           cout << "\nError: Khong the tao thread! " << rc << endl;           exit(-1);        }     }   // giai phong thuoc tinh va doi thread khac     pthread\_attr\_destroy(&attr);     for( i=0; i < SO\_THREAD; i++ ){        rc = pthread\_join(threads[i], &status);        if (rc){           cout << "\nError: khong the ket hop! " << rc << endl;           exit(-1);        }        cout << "\nMain: ket huc thread id :" << i ;        cout << "  va thoat voi trang thai:" << status << endl;     }   cout << "\nMain: thoat chuong trinh!" << endl;     pthread\_exit(NULL);  } |

Biên dịch và chạy chương trình C++ trên sẽ cho kết quả sau:



Trong C11, có một thư viện chuẩn về Luồng Hoạt Động, <threads.h>, nhưng hiện chưa có trình biên dịch nào đã triển khai nó. Do đó, để sử dụng đa luồng trong C, bạn phải sử dụng các triển khai cụ thể cho từng nền tảng như thư viện luồng POSIX (thường được gọi là pthreads) bằng cách sử dụng tiêu đề pthread.h.

## VÍ DỤ ĐƠN GIẢN VỀ LUỒNG C11

|  |
| --- |
| #include <threads.h>  #include <stdio.h>  int run(void \*arg)  {  printf("Hello world of C11 threads.");  return 0;  }  int main(int argc, const char \*argv[]) {  thrd\_t thread;  int result;  thrd\_create(&thread, run, NULL);  thrd\_join(&thread, &result);    printf("Thread return %d at the end\n", result);  } |

# GIAO TIẾP GIỮA CÁC TIẾN TRÌNH (INTERPROCESS COMMUNICATION - IPC)

Cơ chế giao tiếp giữa các tiến trình (IPC) cho phép các tiến trình độc lập khác nhau tương tác với nhau. Ngôn ngữ C chuẩn không cung cấp bất kỳ cơ chế IPC nào. Do đó, tất cả các cơ chế như vậy được xác định bởi hệ điều hành mẹ. POSIX định nghĩa một tập hợp phức tạp các cơ chế IPC; Windows định nghĩa một tập hợp khác; và các hệ thống khác định nghĩa các biến thể riêng của chúng.

## Semaphores

Semaphore được sử dụng để đồng bộ hoạt động giữa hai hoặc nhiều tiến trình. POSIX định nghĩa hai tập hàm khác nhau cho semaphore:

* 'System V IPC' — semctl(), semop(), semget().
* 'POSIX Semaphores' — sem\_close(), sem\_destroy(), sem\_getvalue(), sem\_init(), sem\_open(), sem\_post(),sem\_trywait(), sem\_unlink().

Phần này mô tả semaphore theo kiểu System V IPC, được gọi là như vậy vì chúng bắt nguồn từ Unix System V.

Đầu tiên, bạn cần bao gồm các tiêu đề cần thiết. Phiên bản cũ của POSIX yêu cầu #include <sys/types.h>; nhưng POSIX hiện đại và hầu hết các hệ thống không yêu cầu điều này.

|  |
| --- |
| #include <sys/sem.h> |

Tiếp theo, bạn cần định nghĩa một khóa (key) trong cả tiến trình cha và con.

|  |
| --- |
| #define KEY 0x1111 |

Khóa này cần phải giống nhau trong cả hai chương trình hoặc chúng sẽ không tham chiếu đến cùng một cấu trúc IPC. Có cách để tạo ra một khóa được đồng ý mà không cần gắn cứng giá trị.

Sau đó, tùy theo trình biên dịch của bạn, bạn có thể cần làm bước này: khai báo một union cho mục đích thực hiện các thao tác semaphore.

|  |
| --- |
| union semun {  int val;  struct semid\_ds \*buf;  unsigned short \*array;  }; |

Sau đó, định nghĩa cấu trúc của bạn cho thử (semwait) và nâng (semsignal). Tên P và V bắt nguồn từ tiếng Hà Lan.

|  |
| --- |
| struct sembuf p = { 0, -1, SEM\_UNDO}; # semwait  struct sembuf v = { 0, +1, SEM\_UNDO}; # semsignal |

Bây giờ, bắt đầu bằng cách lấy ID cho semaphore IPC của bạn.

|  |
| --- |
| int id;  // Tham số thứ hai là số lượng semaphore  // Tham số thứ ba là chế độ (IPC\_CREAT tạo bộ semaphore nếu cần)  if ((id = semget(KEY, 1, 0666 | IPC\_CREAT)) < 0) {  /\* Xử lý lỗi \*/  } |

Trong tiến trình cha, khởi tạo semaphore để có một bộ đếm bằng 1.

|  |
| --- |
| union semun u;  u.val = 1;  if (semctl(id, 0, SETVAL, u) < 0) { // SETVAL là một macro chỉ định bạn đang đặt giá trị của semaphore  vào giá trị được xác định bởi union u  /\* Xử lý lỗi \*/  } |

Sau đó, bạn có thể giảm hoặc tăng semaphore theo nhu cầu. Ở đầu phần quan trọng của bạn, bạn giảm bộ đếm sử dụng hàm semop():

|  |
| --- |
| if (semop(id, &p, 1) < 0) {  /\* Xử lý lỗi \*/  } |

Để tăng semaphore, bạn sử dụng &v thay vì &p:

|  |
| --- |
| if (semop(id, &v, 1) < 0) {  /\* Xử lý lỗi \*/  } |

Lưu ý rằng mọi hàm trả về 0 khi thành công và -1 khi xảy ra lỗi. Không kiểm tra các trạng thái trả về này có thể gây ra các vấn đề nghiêm trọng.

**Ví dụ 1.1: Racing with Threads**

Chương trình dưới đây sẽ cho một tiến trình tạo một tiến trình con và cả hai tiến trình cha và con đều cố gắng in các ký tự lên terminal mà không có bất kỳ đồng bộ hóa nào.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <string.h>  int main(int argc, char\* argv[]) {  int pid;  pid = fork();  srand(pid);  if(pid < 0)  {  perror("fork"); exit(1);  }  else if(pid)  {  char \*s = "abcdefgh";  int l = strlen(s);  for(int i = 0; i < l; ++i)  {  putchar(s[i]);  fflush(stdout);  sleep(rand() % 2);  putchar(s[i]);  fflush(stdout);  sleep(rand() % 2);  }  }  else  {  char \*s = "ABCDEFGH";  int l = strlen(s);  for(int i = 0; i < l; ++i)  {  putchar(s[i]);  fflush(stdout);  sleep(rand() % 2);  putchar(s[i]);  fflush(stdout);  sleep(rand() % 2);  }  }  } |

Kết quả (lần chạy đầu tiên):

|  |
| --- |
| aAABaBCbCbDDcEEcddeFFGGHHeffgghh |

(Lần chạy thứ hai):

|  |
| --- |
| aabbccAABddBCeeCffgDDghEEhFFGGHH |

Biên dịch và chạy chương trình này sẽ cho bạn kết quả khác nhau mỗi lần chạy.

Ví dụ 1.2: Avoid Racing with Semaphores

Sửa đổi Ví dụ 1.1 để sử dụng semaphore, chúng ta có:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <string.h>  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h>  #define KEY 0x1111  union semun {  int val;  struct semid\_ds \*buf;  unsigned short \*array;  };  struct sembuf p = { 0, -1, SEM\_UNDO};  struct sembuf v = { 0, +1, SEM\_UNDO};  int main(int argc, char\* argv[]) {  int id = semget(KEY, 1, 0666 | IPC\_CREAT);  if(id < 0)  {  perror("semget"); exit(11);  }  union semun u;  u.val = 1;  if(semctl(id, 0, SETVAL, u) < 0)  {  perror("semctl"); exit(12);  }  int pid;  pid = fork();  srand(pid);  if(pid < 0)  {  perror("fork"); exit(1);  }  else if(pid)  {  char \*s = "abcdefgh";  int l = strlen(s);  for(int i = 0; i < l; ++i)  {  if(semop(id, &p, 1) < 0)  {  perror("semop p"); exit(13);  }  putchar(s[i]);  fflush(stdout);  sleep(rand() % 2);  putchar(s[i]);  fflush(stdout);  if(semop(id, &v, 1) < 0)  {  perror("semop p"); exit(14);  }  sleep(rand() % 2);  }  }  else  {  char \*s = "ABCDEFGH";  int l = strlen(s);  for(int i = 0; i < l; ++i)  {  if(semop(id, &p, 1) < 0)  {  perror("semop p"); exit(15);  }  putchar(s[i]);  fflush(stdout);  sleep(rand() % 2);  putchar(s[i]);  fflush(stdout);  if(semop(id, &v, 1) < 0)  {  perror("semop p"); exit(16);  }  sleep(rand() % 2);  }  }  } |

Kết quả

|  |
| --- |
| aabbAABBCCccddeeDDffEEFFGGHHgghh |

Biên dịch và chạy chương trình này sẽ cho bạn cùng một kết quả mỗi lần chạy.

# TESTING FRAMEWORKS

Nhiều nhà phát triển sử dụng các bài kiểm tra đơn vị (Unit Test) để kiểm tra xem phần mềm của họ hoạt động như mong đợi hay không. Các bài kiểm tra đơn vị (Unit Test) kiểm tra các đơn vị nhỏ của các phần mềm và đảm bảo rằng các đầu ra khớp với kỳ vọng.

Các framework giúp việc kiểm tra đơn vị dễ dàng hơn bằng cách cung cấp dịch vụ thiết lập/loại bỏ và phối hợp các bài kiểm tra.

Có nhiều framework kiểm thử đơn vị có sẵn cho C. Ví dụ, Unity là một framework hoàn toàn bằng ngôn ngữ C. Người ta thường sử dụng các framework kiểm thử C++ để kiểm thử mã C; cũng có nhiều framework kiểm thử C++.

## UNITY TEST FRAMEWORK

Unity là một framework kiểm thử theo kiểu xUnit dành cho việc kiểm thử đơn vị ứng dụng C. Nó được viết hoàn toàn bằng ngôn ngữ C và có khả năng chuyển giao, nhanh chóng, đơn giản, linh hoạt và có khả năng mở rộng. Nó được thiết kế đặc biệt để hữu ích cho việc kiểm thử đơn vị trong các hệ thống nhúng.

Một bài kiểm tra đơn giản kiểm tra giá trị trả về của một hàm có thể trông như sau:

|  |
| --- |
| void test\_FunctionUnderTest\_should\_ReturnFive(void)  {  TEST\_ASSERT\_EQUAL\_INT( 5, FunctionUnderTest() );  } |

Một tập tin kiểm thử đầy đủ có thể trông như sau:

|  |
| --- |
| #include "unity.h"  #include "UnitUnderTest.h" /\* Các đơn vị cần được kiểm thử. \*/  /\* Được chạy trước mỗi bài kiểm tra, đặt cuộc gọi khởi tạo đơn vị ở đây. \*/  void setUp (void) {}  /\* Được chạy sau mỗi bài kiểm tra, đặt cuộc gọi dọn dẹp đơn vị ở đây. \*/  void tearDown (void) {}  void test\_TheFirst(void) {  /\* Bỏ qua bài kiểm tra này nhưng in một thông báo. \*/  TEST\_IGNORE\_MESSAGE("Hello world!");  }  int main (void) {  UNITY\_BEGIN();  RUN\_TEST(test\_TheFirst); /\* Chạy bài kiểm tra. \*/  return UNITY\_END();  } |

Unity đi kèm với một số dự án mẫu, các tập tin makefile và một số kịch bản rake Ruby giúp việc tạo các tập tin kiểm tra dài hơn trở nên dễ dàng hơn một chút.

## Cmocka

CMocka là một framework kiểm thử đơn vị tinh tế dành cho ngôn ngữ C với hỗ trợ cho các đối tượng giả lập (mock objects). Nó chỉ yêu cầu thư viện tiêu chuẩn của C, hoạt động trên nhiều nền tảng tính toán khác nhau (bao gồm cả nhúng) và với các trình biên dịch khác nhau. Nó đi kèm với hướng dẫn kiểm thử với các đối tượng giả lập, tài liệu API và nhiều ví dụ khác nhau.

|  |
| --- |
| #include <stdarg.h>  #include <stddef.h>  #include <setjmp.h>  #include <cmocka.h>  void null\_test\_success (void \*\* state) {}  void null\_test\_fail (void \*\* state) {  assert\_true (0);  }  /\* Những hàm này sẽ được sử dụng để khởi tạo  và dọn dẹp tài nguyên sau mỗi lần chạy kiểm thử \*/  int setup (void \*\* state) {  return 0;  }  int teardown (void \*\* state) {  return 0;  }  int main (void) {  const struct CMUnitTest tests [] =  {  cmocka\_unit\_test (null\_test\_success),  cmocka\_unit\_test (null\_test\_fail),  };  /\* Nếu không cần đến các hàm setup và teardown,  ta có thể truyền NULL thay vào \*/  int count\_fail\_tests =  cmocka\_run\_group\_tests (tests, setup, teardown);    return count\_fail\_tests;  } |

Hãy nhớ rằng, CMocka là một framework kiểm thử đơn vị tinh tế dành cho C, và bạn có thể sử dụng nó để kiểm thử các đơn vị của mã C một cách hiệu quả và dễ dàng.

## Cpputests

CppUTest là một framework kiểm thử đơn vị theo kiểu xUnit dành cho C và C++. Nó được viết bằng C++ và mục tiêu là tính di động và đơn giản trong thiết kế. Nó hỗ trợ phát hiện rò rỉ bộ nhớ, tạo đối tượng giả lập và chạy các kiểm thử cùng với Google Test. CppUTest đi kèm với các tập lệnh hỗ trợ và các dự án mẫu cho Visual Studio và Eclipse CDT.

|  |
| --- |
| #include <CppUTest/CommandLineTestRunner.h>  #include <CppUTest/TestHarness.h>  TEST\_GROUP(Foo\_Group) {}  TEST(Foo\_Group, Foo\_TestOne) {}  /\* Bộ kiểm tra có thể được cung cấp các tùy chọn,  như kích hoạt đầu ra màu sắc, chạy chỉ một kiểm thử  cụ thể hoặc một nhóm kiểm thử, v.v. Điều này sẽ trả  về số lượng kiểm thử thất bại. \*/  int main(int argc, char \*\* argv) {  RUN\_ALL\_TESTS(argc, argv);  } |

Một nhóm kiểm thử có thể có các phương thức setup() và teardown(). Phương thức setup() được gọi trước mỗi kiểm thử và phương thức teardown() được gọi sau mỗi kiểm thử. Cả hai đều tùy chọn và có thể bị bỏ qua độc lập. Các phương thức và biến khác cũng có thể được khai báo trong một nhóm và sẽ có sẵn cho tất cả các kiểm thử của nhóm đó.

|  |
| --- |
| TEST\_GROUP(Foo\_Group) {  size\_t data\_bytes = 128;  void \* data;  void setup() {  data = malloc(data\_bytes);  }  void teardown() {  free(data);  }    void clear() {  memset(data, 0, data\_bytes);  }  } |

# VALGRIND

## MẤT DỮ LIỆU BYTE – QUÊN GIẢI PHÓNG BỘ NHỚ

Dưới đây là một chương trình sử dụng malloc nhưng không sử dụng free:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(int argc, char \*\*argv) {  char \*s;  s = malloc(26); // nguyên nhân gây ra lỗi  return 0;  } |

Nếu không có đối số bổ sung, valgrind sẽ không tìm kiếm lỗi này.

Nhưng nếu chúng ta bật ***--leak-check=yes*** hoặc ***--tool=memcheck***, valgrind sẽ cảnh báo và hiển thị những dòng chịu trách nhiệm cho memory leaks nếu chương trình được biên dịch ở chế độ gỡ lỗi:

|  |
| --- |
| $ valgrind -q --leak-check=yes ./missing\_free  ==4776== 26 bytes in 1 blocks are definitely lost in loss record 1 of 1  ==4776== at 0x4024F20: malloc (vg\_replace\_malloc.c:236)  ==4776== by 0x80483F8: main (missing\_free.c:9)  ==4776== |

Nếu chương trình không được biên dịch ở chế độ gỡ lỗi (ví dụ: không có cờ -g trong GCC), valgrind vẫn sẽ chỉ ra chỗ xảy ra lỗi dựa trên hàm liên quan, nhưng không hiển thị dòng lỗi cụ thể.

Điều này giúp chúng ta quay trở lại và xem xét khối dữ liệu được cấp phát trong dòng đó và cố gắng theo dõi để xem tại sao nó không được giải phóng.

## CÁC LỖI PHỔ BIẾN THƯỜNG GẶP KHI SỬ DỤNG VALGRIND

Valgrind cung cấp cho bạn các dòng mà lỗi xảy ra ở cuối mỗi dòng trong định dạng (file.c:line\_no). Các lỗi trong valgrind được tóm tắt như sau:

|  |
| --- |
| ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0) |

Các lỗi phổ biến bao gồm:

1. **Lỗi đọc/ghi không hợp lệ**

|  |
| --- |
| ==8451== Invalid read of size 2  ==8451== at 0x4E7381D: getenv (getenv.c:84)  ==8451== by 0x4EB1559: \_\_libc\_message (libc\_fatal.c:80)  ==8451== by 0x4F5256B: \_\_fortify\_fail (fortify\_fail.c:37)  ==8451== by 0x4F5250F: \_\_stack\_chk\_fail (stack\_chk\_fail.c:28)  ==8451== by 0x40059C: main (valg.c:10)  ==8451== Address 0x700000007 is not stack'd, malloc'd or (recently) free'd |

Điều này xảy ra khi mã bắt đầu truy cập vào bộ nhớ không thuộc chương trình. Kích thước của bộ nhớ được truy cập cũng cho bạn một gợi ý về biến nào đã được sử dụng.

1. **Sử dụng biến chưa khởi tạo**

|  |
| --- |
| ==8795== 1 errors in context 5 of 8:  ==8795== Conditional jump or move depends on uninitialised value(s)  ==8795== at 0x4E881AF: vfprintf (vfprintf.c:1631)  ==8795== by 0x4E8F898: printf (printf.c:33)  ==8795== by 0x400548: main (valg.c:7) |

Theo lỗi, ở dòng 7 của hàm main của valg.c, cuộc gọi printf() truyền một biến chưa khởi tạo cho printf.

1. **Giải phóng bộ nhớ không hợp lệ**

|  |
| --- |
| ==8954== Invalid free() / delete / delete[] / realloc()  ==8954== at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)  ==8954== by 0x4005A8: main (valg.c:10)  ==8954== Address 0x5203040 is 0 bytes inside a block of size 240 free'd  ==8954== at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)  ==8954== by 0x40059C: main (valg.c:9)  ==8954== Block was alloc'd at  ==8954== at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)  ==8954== by 0x40058C: main (valg.c:7) |

Theo valgrind, mã giải phóng bộ nhớ một cách không hợp lệ (lần thứ hai) ở dòng 10 của valg.c, trong khi nó đã được giải phóng ở dòng 9 và khối bộ nhớ đã được cấp phát ở dòng 7.

## RUNNING VALGRIND

|  |
| --- |
| valgrind ./my-program arg1 arg2 < test-input |

Điều này sẽ chạy chương trình của bạn và tạo một báo cáo về bất kỳ phân bổ và giải phóng nào nó thực hiện. Nó cũng sẽ cảnh báo bạn về các lỗi thông thường như sử dụng bộ nhớ chưa khởi tạo, giải tham chiếu đến nơi lạ lẫm, viết ngoài phạm vi cuối khối được cấp phát bằng malloc, hoặc không giải phóng khối.

## ADDING FLAGS

Bạn cũng có thể bật thêm các kiểm tra, ví dụ:

|  |
| --- |
| valgrind -q --tool=memcheck --leak-check=yes ./my-program arg1 arg2 < test-input |

Xem ***valgrind --help*** để biết thêm thông tin về các tùy chọn (rất nhiều), hoặc xem tài liệu tại ***http://valgrind.org/*** để biết thông tin chi tiết về ý nghĩa của đầu ra.

# CÁC BIỂU THỨC VÀ THỰC TIỄN PHÁT TRIỂN PHỔ BIẾN TRONG LẬP TRÌNH C

## So sánh giá trị chữ và biến

Giả sử bạn đang so sánh một giá trị với một biến:

|  |
| --- |
| if (i == 2) // Cách không tốt  {  doSomething;  } |

Bây giờ giả sử bạn đã nhầm lẫn giữa == và =. Sau đó, bạn sẽ mất thời gian để nhận ra điều đó.

|  |
| --- |
| if (2 == i) // Cách tốt  {  doSomething;  } |

Khi đó, nếu dấu bằng được để thiếu, trình biên dịch sẽ cảnh báo về "cố gắng gán một giá trị chữ."

Điều này không bảo vệ bạn khi so sánh hai biến, nhưng mỗi điều nhỏ như vậy cũng giúp ích.

## Đừng để danh sách tham số của hàm trống – sử dụng “void”

Giả sử bạn đang tạo một hàm không yêu cầu đối số khi nó được gọi và bạn đang phải đối mặt với tình thế tiến thoái lưỡng nan về cách bạn nên xác định danh sách tham số trong nguyên mẫu hàm và định nghĩa hàm.

* Bạn có lựa chọn để giữ danh sách tham số trống cho cả nguyên mẫu và định nghĩa hàm. Do đó, chúng trông giống như câu lệnh gọi hàm mà bạn sẽ cần.
* Bạn đã đọc một nơi nào đó rằng một trong các ứng dụng của từ khóa "void" (chỉ có vài ứng dụng), là để định nghĩa danh sách tham số của các hàm không chấp nhận bất kỳ đối số nào trong lời gọi của chúng. Vì vậy, đây cũng là một lựa chọn.

Vậy, lựa chọn nào là đúng?

**Trả lời:** sử dụng từ khóa **"void"**

**Lời khuyên chung:** Nếu một ngôn ngữ cung cấp một số tính năng cụ thể để sử dụng cho một mục đích đặc biệt, bạn nên sử dụng nó trong mã của bạn. Ví dụ, sử dụng **enum** thay vì các macro #define (điều đó dành cho ví dụ khác).

Phần 6.7.6.3 "Khái báo hàm" của tiêu chuẩn C11, đoạn 10, nêu rõ:

Trường hợp đặc biệt của một tham số vô danh có kiểu void là mục duy nhất trong danh sách này cho biết rằng hàm không có tham số.

Đoạn 14 của cùng mục đó cũng chỉ ra sự khác biệt duy nhất:

... Một danh sách trống trong một khai báo hàm mà là một phần của định nghĩa của hàm đó chỉ ra rằng hàm không có tham số. Danh sách trống trong một khai báo hàm không phải là một phần của định nghĩa của hàm đó chỉ ra rằng không cung cấp thông tin về số hoặc kiểu tham số.

Một giải thích đơn giản được cung cấp bởi K&R (trang 72-73) cho các thông tin trên:

Hơn nữa, nếu một khai báo hàm không bao gồm các tham số, như double atof();, đó cũng được coi là không có gì được giả định về các đối số của atof; tất cả kiểm tra tham số bị tắt. Ý nghĩa đặc biệt này của danh sách tham số trống được thiết kế để cho phép các chương trình C cũ được biên dịch với các trình biên dịch mới. Nhưng việc sử dụng nó với các chương trình mới là một ý tưởng tồi. Nếu hàm có tham số, hãy khai báo chúng; nếu nó không có tham số, hãy sử dụng **void.**

Vì vậy, đây là cách mà nguyên mẫu hàm của bạn sẽ trông như thế này:

|  |
| --- |
| int foo(void); |

Và đây là cách định nghĩa hàm sẽ là:

|  |
| --- |
| int foo(void) {  ...  <các câu lệnh>  ...  return 1;  } |

Một lợi ích của việc sử dụng cách trên, so với cách định nghĩa hàm foo() (tức là không sử dụng từ khóa "void"), là trình biên dịch có thể phát hiện lỗi nếu bạn gọi hàm của bạn bằng câu lệnh lỗi như foo(42). Loại câu lệnh gọi hàm này sẽ không gây ra lỗi nào nếu bạn để danh sách tham số trống. Lỗi sẽ diễn ra một cách yên lặng, không phát hiện và mã vẫn sẽ thực thi.

Điều này cũng có nghĩa là bạn nên định nghĩa hàm main() như sau:

|  |
| --- |
| int main(int argc, char\* argv[]) {  ...  <các câu lệnh>  ...  return 0;  } |

Lưu ý rằng ngay cả khi hàm được định nghĩa với danh sách tham số trống, nó không cung cấp nguyên mẫu cho hàm, vì vậy trình biên dịch sẽ không cảnh báo nếu hàm sau đó được gọi với các đối số. Ví dụ:

|  |
| --- |
| #include <stdio.h>  static void parameterless(void)  {  printf("%s called\n", \_\_func\_\_);  }  int main(int argc, char\* argv[]) {  parameterless(3, "arguments", "provided");  return 0;  } |

Nếu mã đó được lưu trong tệp proto79.c, nó có thể được biên dịch trên Unix với GCC (phiên bản 7.1.0 trên macOS Sierra 10.12.5 được sử dụng cho mục đích thử nghiệm) như sau:

|  |
| --- |
| $ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -pedantic proto79.c -o proto79  $ |

Nếu bạn biên dịch với các tùy chọn nghiêm ngặt hơn, bạn sẽ nhận được lỗi:

|  |
| --- |
| $ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-styledefinition -pedantic proto79.c -o proto79  proto79.c:3:13: error: function declaration isn’t a prototype [-Werror=strict-prototypes]  static void parameterless()  ^~~~~~~~~~  proto79.c: In function ‘parameterless’:  proto79.c:3:13: error: old-style function definition [-Werror=old-style-definition]  cc1: all warnings being treated as errors  $ |

Nếu bạn đặt cho hàm nguyên mẫu một tham số prototype static void parameterless(void), thì quá trình biên dịch sẽ cho lỗi:

|  |
| --- |
| $ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-styledefinition -pedantic proto79.c -o proto79  proto79.c: In function ‘main’:  proto79.c:10:5: error: too many arguments to function ‘parameterless’  parameterless(3, "arguments", "provided");  ^~~~~~~~~  proto79.c:3:13: note: declared here  static void parameterless(void)  ^~~~~~~~~~~~~  $ |

Moral: Luôn luôn đảm bảo bạn có nguyên mẫu, và đảm bảo trình biên dịch của bạn cho bạn biết khi bạn không tuân thủ các quy tắc.

# NHỮNG LỖI PHỔ BIẾN

Phần này thảo luận về một số lỗi phổ biến mà một lập trình viên C nên biết và nên tránh mắc phải. Để biết thêm về một số sự cố không mong muốn và nguyên nhân của chúng, vui lòng xem **Undefined behavior**

## KẾT HỢP SINGED VÀ UNSIGNED INTEGERS TRONG CÁC PHÉP TOÁN SỐ HỌC

Thường thì không phải là một ý tưởng tốt khi kết hợp các số nguyên có dấu và không dấu trong các phép tính. Ví dụ, kết quả sẽ là gì trong ví dụ sau?

|  |
| --- |
| #include <stdio.h>  int main(void)  {  unsigned int a = 1000;  signed int b = -1;  if (a > b) puts("a is more than b");  else puts("a is less or equal than b");  return 0;  } |

Với giá trị 1000 lớn hơn -1, bạn sẽ hy vọng kết quả là a is more than b, tuy nhiên điều đó sẽ không đúng.

Các phép tính toán giữa các kiểu nguyên khác nhau được thực hiện trong một kiểu chung được xác định bởi những sự chuyển đổi thông thường của phép toán (xem cấu hình ngôn ngữ, 6.3.1.8).

Trong trường hợp này, "kiểu chung" là unsigned int, bởi vì, như được nêu trong ***Usual arithmetic conversions,***

714 Nếu phép toán có kiểu số nguyên không dấu có thứ hạng lớn hơn hoặc bằng thứ hạng của kiểu của phép toán khác, thì phép toán có kiểu số nguyên có dấu được chuyển đổi sang kiểu của phép toán có kiểu số nguyên không dấu.

Điều này có nghĩa rằng phần tử int b sẽ được chuyển đổi thành unsigned int trước khi thực hiện so sánh.

Khi -1 được chuyển đổi thành unsigned int, kết quả là giá trị unsigned int lớn nhất có thể, lớn hơn 1000, có nghĩa là a > b là sai.

## MACROS CHỈ LÀ SỰ THAY THẾ CHUỖI ĐƠN GIẢN

Macro là sự thay thế đơn giản cho chuỗi. (Nó hoạt động với các thông số tiền xử lý, không phải là chuỗi tùy ý.)

|  |
| --- |
| #include <stdio.h>  #define SQUARE(x) x\*x  int main(void) {  printf("%d\n", SQUARE(1+2));  return 0;  } |

Bạn có thể mong đợi đoạn mã này in ra 9 (3\*3), nhưng thực tế sẽ in ra 5 vì macro sẽ được mở rộng thành 1+2\*1+2.

Bạn nên bọc các đối số và toàn bộ biểu thức của macro trong dấu ngoặc đơn để tránh vấn đề này.

|  |
| --- |
| #include <stdio.h>  #define SQUARE(x) ((x)\*(x))  int main(void) {  printf("%d\n", SQUARE(1+2));  return 0;  } |

Một vấn đề khác là các đối số của macro không được đảm bảo được đánh giá một lần; chúng có thể không được đánh giá hoặc có thể được đánh giá nhiều lần.

|  |
| --- |
| #include <stdio.h>  #define MIN(x, y) ((x) <= (y) ? (x) : (y))  int main(void) {  int a = 0;  printf("%d\n", MIN(a++, 10));  printf("a = %d\n", a);  return 0;  } |

Trong đoạn mã này, macro sẽ được mở rộng thành ((a++) <= (10) ? (a++) : (10)). Vì a++ (0) nhỏ hơn 10, a++ sẽ được đánh giá hai lần và giá trị của a và giá trị được trả về từ MIN sẽ khác với bạn mong đợi.

Điều này có thể được tránh bằng cách sử dụng các hàm, nhưng lưu ý rằng các kiểu sẽ được cố định bởi định nghĩa hàm, trong khi macro có thể (quá) linh hoạt về kiểu.

|  |
| --- |
| #include <stdio.h>  int min(int x, int y) {  return x <= y ? x : y;  }  int main(void) {  int a = 0;  printf("%d\n", min(a++, 10));  printf("a = %d\n", a);  return 0;  } |

Bây giờ vấn đề của đánh giá hai lần đã được khắc phục, nhưng hàm min này không thể xử lý dữ liệu kiểu double mà không bị cắt xén, ví dụ.

Các chỉ thị macro có thể có hai loại:

|  |
| --- |
| #define OBJECT\_LIKE\_MACRO tiếp theo là một "danh sách thay thế" của mã thông báo tiền xử lý.  #define FUNCTION\_LIKE\_MACRO(with, arguments) tiếp theo là một danh sách thay thế |

Điều phân biệt hai loại macro này là ký tự theo sau sau bộ nhận dạng sau #define: nếu đó là dấu ngoặc tròn, đó là một macro dạng hàm; nếu không, đó là một macro dạng đối tượng. Nếu ý định là viết một macro dạng hàm, không được có khoảng trắng nào giữa cuối tên của macro và (. Kiểm tra điều này để có một giải thích chi tiết.

Phiên bản ≥ C99

Trong C99 hoặc mới hơn, bạn có thể sử dụng static inline int min(int x, int y) { … }.

Phiên bản ≥ C11

Trong C11, bạn có thể viết một biểu thức 'type-generic' cho min.

|  |
| --- |
| #include <stdio.h>  #define min(x, y) \_Generic((x), \  long double: min\_ld, \  unsigned long long: min\_ull, \  default: min\_i \  )(x, y)  #define gen\_min(suffix, type) \  static inline type min\_##suffix(type x, type y) { return (x < y) ? x : y; }  gen\_min(ld, long double)  gen\_min(ull, unsigned long long)  gen\_min(i, int)  int main(int argc, char\* argv[]) {  unsigned long long ull1 = 50ULL;  unsigned long long ull2 = 37ULL;  printf("min(%llu, %llu) = %llu\n", ull1, ull2, min(ull1, ull2));  long double ld1 = 3.141592653L;  long double ld2 = 3.141592652L;  printf("min(%.10Lf, %.10Lf) = %.10Lf\n", ld1, ld2, min(ld1, ld2));  int i1 = 3141653;  int i2 = 3141652;  printf("min(%d, %d) = %d\n", i1, i2, min(i1, i2));  return 0;  } |

The generic expression có thể được mở rộng với nhiều kiểu khác nhau như double, float, long long, unsigned long, long, unsigned - và các lời gọi macro gen\_min thích hợp được viết.

## QUÊN SAO CHÉP GIÁ TRỊ TRẢ VỀ CỦA REALLOC VÀO MỘT BIẾN TẠM

Khi realloc thất bại, nó trả về NULL. Nếu bạn gán giá trị của bộ đệm ban đầu cho giá trị trả về của realloc và nếu nó trả về NULL, thì bộ đệm ban đầu (con trỏ cũ) bị mất, dẫn đến **memory leak.**Giải pháp là sao chép vào một con trỏ tạm, và nếu con trỏ tạm không phải là NULL, sau đó sao chép vào bộ đệm thực sự.

|  |
| --- |
| char \*buf, \*tmp;  buf = malloc(...);  ...  /\* SAI \*/  if ((buf = realloc(buf, 16)) == NULL)  perror("realloc");  /\* ĐÚNG \*/  if ((tmp = realloc(buf, 16)) != NULL)  buf = tmp;  else  perror("realloc"); |

Trong trường hợp realloc thất bại, giá trị của biến tạm tmp vẫn giữ nguyên bộ nhớ cũ, giúp bạn không mất dữ liệu và tránh **memory leak.**

## QUÊN CẤP PHÁT THÊM MỘT BYTE CHO \0

Khi bạn sao chép một chuỗi vào một bộ đệm đã cấp phát bằng malloc, luôn nhớ thêm 1 vào strlen.

|  |
| --- |
| char \*dest = malloc(strlen(src)); /\* SAI \*/  char \*dest = malloc(strlen(src) + 1); /\* ĐÚNG \*/  strcpy(dest, src); |

Điều này là vì strlen không bao gồm ký tự \0 cuối cùng trong chiều dài. Nếu bạn tiếp cận SAI (như được hiển thị ở trên), khi gọi strcpy, chương trình của bạn sẽ gây ra hành vi không xác định.

Điều này cũng áp dụng trong tình huống bạn đang đọc một chuỗi có độ dài tối đa đã biết từ stdin hoặc một nguồn khác. Ví dụ:

|  |
| --- |
| #define MAX\_INPUT\_LEN 42  char buffer[MAX\_INPUT\_LEN]; /\* SAI \*/  char buffer[MAX\_INPUT\_LEN + 1]; /\* ĐÚNG \*/  scanf("%42s", buffer); /\* Đảm bảo rằng bộ đệm không bị tràn \*/ |

## HIỂU SAI VỀ SỰ ARRAY DECAY

Một vấn đề phổ biến trong mã sử dụng các mảng đa chiều, mảng các con trỏ, v.v. là sự thật rằng Type\*\* và Type[M][N] là các kiểu khác nhau:

|  |
| --- |
| #include <stdio.h>  void print\_strings(char \*\*strings, size\_t n)  {  size\_t i;  for (i = 0; i < n; i++)  puts(strings[i]);  }  int main(int argc, char\* argv[]) {  char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};  print\_strings(s, 4);  return 0;  } |

Kết quả mẫu từ trình biên dịch:

|  |
| --- |
| file1.c: In function 'main':  file1.c:13:23: error: passing argument 1 of 'print\_strings' from incompatible pointer type [- Wincompatible-pointer-types]  print\_strings(strings, 4);  ^  file1.c:3:10: note: expected 'char \*\*' but argument is of type 'char (\*)[20]'  void print\_strings(char \*\*strings, size\_t n) |

Lỗi cho biết rằng mảng s trong hàm main được chuyển đến hàm print\_strings, mà mong đợi một kiểu con trỏ khác với kiểu nó nhận được. Nó cũng bao gồm lưu ý về kiểu mà print\_strings mong đợi và kiểu đã truyền vào từ main.

Vấn đề xuất phát từ việc gọi là phân rã mảng. Khi s với kiểu char[4][20] (mảng của 4 mảng có 20 ký tự) được truyền vào hàm, nó sẽ biến thành một con trỏ đến phần tử đầu tiên của nó như thể bạn đã viết &s[0], kiểu char (\*)[20] (con trỏ đến 1 mảng có 20 ký tự). Điều này xảy ra cho bất kỳ mảng nào, bao gồm cả mảng con trỏ, mảng các mảng của mảng (mảng 3 chiều) và mảng các con trỏ đến mảng. Dưới đây là một bảng minh họa về việc gì xảy ra khi một mảng phân rã. Nhấn mạnh các thay đổi trong mô tả kiểu để minh họa việc gì xảy ra:

|  |  |  |  |
| --- | --- | --- | --- |
| trước khi phân rã |  | sau khi phân rã |  |
| char [20] | mảng (20 ký tự) | char \* | con trỏ tới (1 ký tự) |
| char [4][20] | mảng của (4 mảng 20 ký tự) | char (\*)[20] | con trỏ tới (1 mảng 20 ký tự) |
| char \*[4] | mảng (4 con trỏ tới 1 ký tự) | char \*\* | con trỏ tới (1 con trỏ tới 1 ký tự) |
| char [3][4][20] | mảng của (3 mảng 4 mảng 20 ký tự) | char (\*)[4][20] | con trỏ tới (1 mảng 4 mảng 20 ký tự) |
| char (\*[4])[20] | mảng (4 con trỏ tới 1 mảng 20 ký tự) | char (\*\*)[20] | con trỏ tới (1 con trỏ tới 1 mảng 20 ký tự) |

Nếu một mảng có thể phân rã thành con trỏ, thì có thể nói rằng con trỏ có thể được coi là một mảng có ít nhất 1 phần tử. Một ngoại lệ cho điều này là con trỏ null, trỏ đến không gì và do đó không phải là một mảng.

Sự phân rã mảng chỉ xảy ra một lần. Nếu một mảng đã phân rã thành con trỏ, thì nó đã trở thành một con trỏ, không phải là mảng. Ngay cả khi bạn có một con trỏ đến một mảng, hãy nhớ rằng con trỏ có thể được xem xét là một mảng có ít nhất một phần tử, vì vậy sự phân rã mảng đã xảy ra.

Nói cách khác, một con trỏ đến một mảng (char (\*)[20]) sẽ không bao giờ trở thành một con trỏ đến một con trỏ (char \*\*). Để sửa hàm print\_strings, chỉ cần làm cho nó nhận kiểu đúng:

|  |
| --- |
| void print\_strings(char (\*strings)[20], size\_t n)  /\* HOẶC \*/  void print\_strings(char strings[][20], size\_t n) |

Một vấn đề nảy sinh khi bạn muốn hàm print\_strings trở nên tổng quát cho bất kỳ mảng ký tự nào: nếu có 30 ký tự thay vì 20? Hoặc 50? Câu trả lời là thêm một tham số khác trước tham số mảng:

|  |
| --- |
| #include <stdio.h>  /\*  \* Lưu ý về việc sắp xếp lại các tham số và thay đổi tên tham số  \* từ các định nghĩa trước đó:  \* n (số lượng chuỗi)  \* => scount (số lượng chuỗi)  \*  \* Tất nhiên, bạn cũng có thể sử dụng một trong các dạng được khuyến nghị dưới đây  \* cho tham số `strings` thay thế:  \*  \* char strings[scount][ccount]  \* char strings[][ccount]  \*/  void in\_cac\_chuoi(size\_t so\_luong\_chuoi, size\_t do\_dai\_chuoi, char (\*chuoi)[do\_dai\_chuoi])  {  size\_t i;  for (i = 0; i < so\_luong\_chuoi; i++)  puts(chuoi[i]);  }  int main(int argc, char\* argv[]) {  char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};  in\_cac\_chuoi(4, 20, s);  return 0;  } |

Biên dịch nó sẽ không gây ra lỗi và cho ra kết quả như mong đợi:

|  |
| --- |
| Example 1  Example 2  Example 3  Example 4 |

## QUÊN GIẢI PHÓNG BỘ NHỚ (MEMORY LEAKS)

Một thực hành tốt trong lập trình là giải phóng bất kỳ bộ nhớ nào đã được cấp phát trực tiếp bởi mã của bạn, hoặc ngầm định thông qua việc gọi các hàm nội bộ hoặc ngoại vi, như một API thư viện như strdup(). Quên giải phóng bộ nhớ có thể dẫn đến việc xảy ra rò rỉ bộ nhớ, dẫn đến việc tích luỹ một lượng lớn bộ nhớ bị lãng phí mà không sử dụng, có thể gây ra sự cố hoặc hành vi không xác định. Các vấn đề có khả năng xảy ra nếu việc rò rỉ được xảy ra lặp đi lặp lại trong một vòng lặp hoặc hàm đệ quy. Nguy cơ của việc gặp sự cố trong chương trình tăng lên khi chương trình rò rỉ chạy càng lâu. Đôi khi các vấn đề có thể xuất hiện ngay lập tức; lúc khác, vấn đề có thể không thấy trong vài giờ hoặc ngay cả sau nhiều năm hoạt động liên tục. Sự cố do cạn kiệt bộ nhớ có thể gây ra hậu quả nghiêm trọng, tùy thuộc vào tình huống.

Vòng lặp vô hạn dưới đây là một ví dụ về việc rò rỉ bộ nhớ\*\* có thể dần tiêu thụ bộ nhớ có sẵn bằng cách gọi hàm getline(), một hàm tạo bộ nhớ mới ngầm định mà không giải phóng bộ nhớ đó.

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  int main(int argc, char\* argv[]) {  char \*line = NULL;  size\_t size = 0;    /\* Vòng lặp dưới đây rò rỉ bộ nhớ càng nhanh càng tốt \*/  for (;;)  {  getline(&line, &size, stdin); /\* Bộ nhớ mới được cấp phát ngầm định \*/  /\* <làm bất kỳ điều gì> \*/  line = NULL;  }  return 0;  } |

Ngược lại, đoạn mã dưới đây cũng sử dụng hàm getline(), nhưng lần này, bộ nhớ đã được cấp phát được giải phóng đúng cách, tránh rò rỉ.

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  int main(int argc, char\* argv[]) {  char \*line = NULL;  size\_t size = 0;  for (;;)  {  if (getline(&line, &size, stdin) < 0)  {  free(line);  line = NULL;  /\* Xử lý lỗi như đặt cờ, thoát khỏi vòng lặp và/hoặc kết thúc chương trình \*/  }  /\* <làm bất kỳ điều gì> \*/  free(line);  line = NULL;  }  return 0;  } |

Rò rỉ bộ nhớ không luôn luôn có hậu quả rõ ràng và không nhất thiết là một vấn đề chức năng. Mặc dù "thực hành tốt" đề ra là giải phóng bộ nhớ một cách nghiêm ngặt tại các điểm và điều kiện chiến lược, nhằm giảm bớt dấu chân bộ nhớ và giảm nguy cơ cạn kiệt bộ nhớ, nhưng cũng có những ngoại lệ. Ví dụ, nếu chương trình có thời gian hoạt động và phạm vi giới hạn, nguy cơ gặp sự cố cấp phát có thể được coi là quá nhỏ để lo lắng. Trong trường hợp đó, việc bỏ qua việc giải phóng rõ ràng có thể được coi là chấp nhận được. Ví dụ, hầu hết các hệ điều hành hiện đại tự động giải phóng toàn bộ bộ nhớ mà chương trình tiêu thụ khi chương trình kết thúc, cho dù đó là do chương trình gặp lỗi, cuộc gọi hệ thống đến hàm exit(), chấm dứt tiến trình hoặc đạt đến cuối main(). Việc giải phóng bộ nhớ một cách rõ ràng tại thời điểm kết thúc chương trình có thể gây ra tình trạng lặp lại hoặc giới hạn hiệu suất.

Việc cấp phát có thể thất bại nếu bộ nhớ không đủ, và việc xử lý sự cố nên được xem xét ở mức độ phù hợp trên ngăn xếp cuộc gọi. Hàm getline(), được hiển thị ở trên, là một tình huống sử dụng thú vị vì nó là một hàm thư viện không chỉ cấp phát bộ nhớ mới mà còn để cho người gọi giải phóng, nhưng có thể thất bại vì nhiều lý do khác nhau, tất cả những lý do đó đều phải được xem xét. Do đó, khi sử dụng một API C, đọc tài liệu (trang man) và chú ý đặc biệt đến các điều kiện lỗi và việc sử dụng bộ nhớ, và biết rõ lớp phần mềm nào chịu trách nhiệm giải phóng bộ nhớ được trả về.

Một thực hành xử lý bộ nhớ khác thường là đặt các con trỏ bộ nhớ thành NULL ngay sau khi bộ nhớ mà con trỏ đó tham chiếu đã được giải phóng, để các con trỏ này có thể được kiểm tra tính hợp lệ bất kỳ lúc nào (ví dụ: kiểm tra NULL / không NULL), bởi vì truy cập vào bộ nhớ đã giải phóng có thể dẫn đến các vấn đề nghiêm trọng như lấy dữ liệu vô nghĩa (phép đọc), hoặc hỏng dữ liệu (phép ghi) và/hoặc làm cho chương trình bị sập. Trong hầu hết các hệ điều hành hiện đại, việc giải phóng vị trí bộ nhớ 0 (NULL) không có tác dụng (ví dụ: nó là vô hại), như được yêu cầu bởi tiêu chuẩn C — vì vậy bằng cách thiết lập một con trỏ thành NULL, không có rủi ro của việc giải phóng bộ nhớ hai lần nếu con trỏ được chuyển cho hàm free(). Hãy nhớ rằng việc giải phóng bộ nhớ hai lần có thể dẫn đến những vấn đề mất thời gian, khó hiểu và khó chẩn đoán.

## SAO CHÉP QUÁ NHIỀU

|  |
| --- |
| char buf[8]; /\* tiny buffer, easy to overflow \*/  printf("What is your name?\n");  scanf("%s", buf); /\* WRONG \*/  scanf("%7s", buf); /\* RIGHT \*/ |

Nếu người dùng nhập vào một chuỗi dài hơn 7 ký tự (-1 đối với ký tự kết thúc null), bộ nhớ phía sau bộ đệm buf sẽ bị ghi đè. Điều này dẫn đến hành vi không xác định. Tin tặc độc hại thường khai thác điều này để ghi đè địa chỉ trả lại và thay đổi nó thành địa chỉ chứa mã độc của tin tặc.

## NHẦM LẪN VIẾT = THAY VÌ == KHI SO SÁNH

Toán tử = được sử dụng để gán giá trị.

Toán tử == được sử dụng để so sánh.

Mọi người cần cẩn trọng để không nhầm lẫn hai toán tử này. Đôi khi người ta viết sai như sau:

|  |
| --- |
| /\* gán y cho x \*/  if (x = y) {  /\* logic \*/  } |

trong khi thực ra điều mong muốn là:

|  |
| --- |
| /\* so sánh x có bằng y không \*/  if (x == y) {  /\* logic \*/  } |

Phần đầu gán giá trị của y cho x và kiểm tra xem giá trị đó có khác 0 hay không, thay vì thực hiện so sánh. Điều này tương đương với:

|  |
| --- |
| if ((x = y) != 0) {  /\* logic \*/  } |

Có những trường hợp khi kiểm tra kết quả của việc gán giá trị được dự định và thường được sử dụng, vì nó tránh việc phải nhân đôi mã và phải xử lý lần đầu một cách đặc biệt. So sánh:

|  |
| --- |
| while ((c = getopt\_long(argc, argv, short\_options, long\_options, &option\_index)) != -1) {  switch (c) {  ...  }  } |

Với

|  |
| --- |
| while ((c = getopt\_long(argc, argv, short\_options, long\_options, &option\_index)) != -1) {  switch (c) {  ...  }  } |

Các trình biên dịch hiện đại sẽ nhận ra mẫu này và không cảnh báo khi việc gán được đặt trong dấu ngoặc như trên, nhưng có thể cảnh báo cho các sử dụng khác. Ví dụ:

|  |
| --- |
| if (x = y) /\* cảnh báo \*/  if ((x = y)) /\* không cảnh báo \*/  if ((x = y) != 0) /\* không cảnh báo; tường minh \*/ |

Một số lập trình viên sử dụng chiến lược đặt hằng số ở bên trái của toán tử (thường được gọi là điều kiện Yoda). Bởi vì hằng số là lvalue, kiểu điều kiện này sẽ khiến trình biên dịch báo lỗi nếu sử dụng sai toán tử.

|  |
| --- |
| if (5 = y) /\* Lỗi \*/  if (5 == y) /\* Không lỗi \*/ |

Tuy nhiên, cách này làm giảm tính đọc của mã và không được coi là cần thiết nếu lập trình viên tuân theo các nguyên tắc lập trình C tốt, và không giúp khi so sánh hai biến nên nó không phải là một giải pháp phổ quát. Hơn nữa, nhiều trình biên dịch hiện đại có thể cảnh báo khi mã được viết với các điều kiện Yoda.

## KÍ TỰ XUỐNG DÒNG KHÔNG ĐƯỢC SƯ DỤNG TRONG LỜI GỌI SCANF() THÔNG THƯỜNG

Khi chương trình sau đây được thực thi với đầu vào như sau:

|  |
| --- |
| #include <stdio.h>  #include <string.h>  int main(int argc, char\* argv[]) {  int num = 0;  char str[128], \*lf;  scanf("%d", &num);  fgets(str, sizeof(str), stdin);    if ((lf = strchr(str, '\n')) != NULL) \*lf = '\0';  printf("%d \"%s\"\n", num, str);  return 0;  } |

và đầu vào là:

|  |
| --- |
| 42  life |

đầu ra sẽ là 42 "" thay vì kết quả mong đợi là 42 "life".

Lý do là do ký tự xuống dòng sau số 42 không được sử dụng trong lời gọi scanf() và nó lại được sử dụng bởi fgets() trước khi đọc chuỗi "life". Sau đó, fgets() ngừng đọc trước khi đọc "life".

Để tránh vấn đề này, một cách hữu ích khi độ dài tối đa của dòng là đã biết - ví dụ khi giải quyết các vấn đề trong hệ thống trọng tài trực tuyến - là tránh sử dụng scanf() trực tiếp và đọc tất cả các dòng qua fgets(). Bạn có thể sử dụng sscanf() để phân tích các dòng đã đọc.

|  |
| --- |
| #include <stdio.h>  #include <string.h>  int main(int argc, char\* argv[]) {  int num = 0;  char line\_buffer[128] = "", str[128], \*lf;  fgets(line\_buffer, sizeof(line\_buffer), stdin);  sscanf(line\_buffer, "%d", &num);  fgets(str, sizeof(str), stdin);    if ((lf = strchr(str, '\n')) != NULL) \*lf = '\0';  printf("%d \"%s\"\n", num, str);  return 0;  } |

Một cách khác là đọc cho đến khi bạn gặp một ký tự xuống dòng sau khi sử dụng scanf() và trước khi sử dụng fgets().

|  |
| --- |
| #include <stdio.h>  #include <string.h>  int main(int argc, char\* argv[]) {  int num = 0;  char str[128], \*lf;  int c;  scanf("%d", &num);  while ((c = getchar()) != '\n' && c != EOF);  fgets(str, sizeof(str), stdin);    if ((lf = strchr(str, '\n')) != NULL) \*lf = '\0';  printf("%d \"%s\"\n", num, str);  return 0;  } |

## THÊM DẤU CHẤM PHẨY VÀO #DEFINE

Dễ dàng bị nhầm lẫn trong trình xử lý tiền xử lý C và xem nó như một phần của C chính, nhưng đó là sai lầm vì trình xử lý tiền xử lý chỉ là cơ chế thay thế văn bản. Ví dụ, nếu bạn viết

|  |
| --- |
| #define MAX 100;  int arr[MAX]; |

mã sẽ mở rộng thành

|  |
| --- |
| int arr[100;]; |

điều này là một lỗi cú pháp. Giải pháp là loại bỏ dấu chấm phẩy khỏi dòng #define. Gần như luôn luôn là một lỗi khi kết thúc #define bằng dấu chấm phẩy.

## SỬ DỤNG DẤU CHẤM PHẨY KHÔNG CẨN THẬN

Hãy cẩn thận khi sử dụng dấu chấm phẩy. Ví dụ sau đây

|  |
| --- |
| if (x > a);  a = x; |

thực chất có nghĩa là:

|  |
| --- |
| if (x > a) {}  a = x; |

điều này có nghĩa là x sẽ được gán cho a trong mọi trường hợp, điều đó có thể không phải là điều bạn mong muốn ban đầu.

Đôi khi, thiếu dấu chấm phẩy cũng có thể gây ra vấn đề khó phát hiện:

|  |
| --- |
| if (i < 0)  return  day = date[0];  hour = date[1];  minute = date[2]; |

Dấu chấm phẩy sau return đã bị bỏ sót, vì vậy day=date[0] sẽ được trả về.

Một kỹ thuật để tránh vấn đề này và các vấn đề tương tự là luôn sử dụng dấu ngoặc nhọn cho các điều kiện và vòng lặp có nhiều dòng. Ví dụ:

|  |
| --- |
| if (x > a) {  a = x;  } |

## LỖI UNDEFINED REFERENCE KHI LINKING

Một trong những lỗi phổ biến nhất trong quá trình biên dịch xảy ra trong giai đoạn linking. Lỗi có dạng tương tự như sau:

|  |
| --- |
| $ gcc undefined\_reference.c  /tmp/ccoXhwF0.o: In function `main':  undefined\_reference.c:(.text+0x15): undefined reference to `foo'  collect2: error: ld returned 1 exit status  $ |

Vậy, hãy xem xét mã đã tạo ra lỗi này:

|  |
| --- |
| int foo(void);  int main(int argc, char \*\*argv) {  int foo\_val;  foo\_val = foo();  return foo\_val;  } |

Ở đây, chúng ta thấy một sự khai báo của foo (int foo();), nhưng không có định nghĩa thực tế cho nó (hàm thực sự). Vì vậy, chúng ta cung cấp trình biên dịch với phần khai báo hàm, nhưng không có hàm như vậy được định nghĩa ở bất kỳ đâu. Do đó, giai đoạn biên dịch diễn ra mà không có vấn đề gì, nhưng trình liên kết dừng với lỗi "Undefined reference".

Để sửa lỗi này trong chương trình nhỏ của chúng ta, chúng ta chỉ cần thêm một định nghĩa cho foo:

|  |
| --- |
| /\* Khai báo foo \*/  int foo(void);  /\* Định nghĩa foo \*/  int foo(void)  {  return 5;  }  int main(int argc, char \*\*argv) {  int foo\_val;  foo\_val = foo();  return foo\_val;  } |

Bây giờ mã này sẽ được biên dịch thành công. Trường hợp thay thế khác xảy ra khi mã nguồn cho foo() nằm trong một tệp nguồn riêng foo.c (và có một tiêu đề foo.h để khai báo foo() và được bao gồm cả trong foo.c và undefined\_reference.c). Lúc này, để sửa lỗi, chúng ta cần liên kết cả tệp đối tượng từ foo.c và undefined\_reference.c, hoặc biên dịch cả hai tệp nguồn:

|  |
| --- |
| $ gcc -c undefined\_reference.c  $ gcc -c foo.c  $ gcc -o working\_program undefined\_reference.o foo.o |

Hoặc

|  |
| --- |
| $ gcc -o working\_program undefined\_reference.c foo.c |

Một tình huống phức tạp hơn là khi liên quan đến thư viện, như trong mã sau:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <math.h>  int main(int argc, char \*\*argv) {  double first;  double second;  double power;  if (argc != 3)  {  fprintf(stderr, "Usage: %s <denom> <nom>\n", argv[0]);  return EXIT\_FAILURE;  }  /\* Chuyển đổi đầu vào người dùng thành số, kiểm tra lỗi bổ sung  \* nên được thực hiện ở đây. \*/  first = strtod(argv[1], NULL);  second = strtod(argv[2], NULL);  /\* Sử dụng hàm pow() từ libm - điều này sẽ gây ra lỗi liên kết  \* trừ khi mã này được biên dịch với libm! \*/  power = pow(first, second);    printf("%f mũ %f = %f\n", first, second, power);  return EXIT\_SUCCESS;  } |

Mã này cú pháp đúng, khai báo cho pow() tồn tại từ #include <math.h>, vì vậy chúng ta cố gắng biên dịch và liên kết (link) nhưng nhận được lỗi như sau:

|  |
| --- |
| $ gcc no\_library\_in\_link.c -o no\_library\_in\_link  /tmp/ccduQQqA.o: In **function** `main':  **no\_library\_in\_link.c:(.text+0x8b): undefined reference to `pow'**  collect2: error: ld returned 1 exit status  $ |

Điều này xảy ra vì trình liên kết không tìm thấy định nghĩa cho pow() trong giai đoạn liên kết. Để sửa lỗi này, chúng ta cần chỉ định rằng chúng ta muốn liên kết với thư viện toán học libm bằng cách chỉ định cờ -lm. (Lưu ý rằng có các nền tảng như macOS mà không cần -lm, nhưng khi bạn gặp lỗi tham chiếu chưa xác định, thì thư viện này cần thiết.)

Vì vậy, chúng ta thực hiện giai đoạn biên dịch lại, lần này chỉ định thư viện (sau các tệp nguồn hoặc đối tượng):

|  |
| --- |
| $ gcc no\_library\_in\_link.c -lm -o library\_in\_link\_cmd  $ ./library\_in\_link\_cmd 2 4  2.000000 mũ 4.000000 = 16.000000 |

Và nó hoạt động

## KIỂM TRA BIỂU THỨC LOGIC VỚI TRUE

Tiêu chuẩn C ban đầu không có kiểu dữ liệu Boolean nội tại, vì vậy bool, **true** và **false** không có ý nghĩa cố định và thường được định nghĩa bởi các lập trình viên. Thông thường, **true** sẽ được định nghĩa là 1 và **false** sẽ được định nghĩa là 0.

Phiên bản ≥ C99

C99 thêm kiểu dữ liệu tích hợp \_Bool và tiêu đề **<stdbool.h>** định nghĩa bool (mở rộng thành \_Bool), **false** và **true**. Nó cũng cho phép bạn định nghĩa lại bool, **true** và **false**, nhưng ghi chú rằng đây là tính năng sắp cải tổ.

Quan trọng hơn, các biểu thức logic xử lý bất cứ thứ gì có giá trị 0 như là false và bất kỳ giá trị khác không bằng 0 như là true. Ví dụ:

|  |
| --- |
| /\* Trả về 'true' nếu bit quan trọng nhất được đặt \*/  bool isUpperBitSet(uint8\_t bitField)  {  if ((bitField & 0x80) == true) /\* Phép so sánh chỉ thành công nếu true là 0x80 và bitField có  bit đó được đặt \*/  {  return true;  }  else  {  return false;  }  } |

Trong ví dụ trên, hàm đang kiểm tra xem bit quan trọng nhất có được đặt và trả về true nếu có. Tuy nhiên, bằng cách kiểm tra một cách rõ ràng với true, câu lệnh if chỉ sẽ thành công nếu (bitfield & 0x80) được đánh giá thành true, có nghĩa là true được định nghĩa như là 1 và rất ít khi là 0x80. Hãy kiểm tra một cách rõ ràng cho trường hợp bạn mong đợi:

|  |
| --- |
| /\* Trả về 'true' nếu bit quan trọng nhất được đặt \*/  bool isUpperBitSet(uint8\_t bitField)  {  if ((bitField & 0x80) == 0x80) /\* Kiểm tra rõ ràng cho trường hợp chúng ta mong đợi \*/  {  return true;  }  else  {  return false;  }  } |

Hoặc đánh giá bất kỳ giá trị không bằng 0 nào như là true.

|  |
| --- |
| /\* Trả về 'true' nếu bit quan trọng nhất được đặt \*/  bool isUpperBitSet(uint8\_t bitField)  {  /\* Nếu bit quan trọng nhất được đặt, kết quả là 0x80 mà if sẽ đánh giá là true \*/  if (bitField & 0x80)  {  return true;  }  else  {  return false;  }  } |

## THỰC HIỆN MỞ RỘNG THÊM TRONG CON TRỎ SỐ HỌC

Trong phép tính con trỏ, số nguyên được thêm hoặc trừ vào con trỏ được hiểu không phải là sự thay đổi địa chỉ mà là số phần tử cần di chuyển.

|  |
| --- |
| #include <stdio.h>  int main(int argc, char\* argv[]) {  int array[] = {1, 2, 3, 4, 5};  int \*ptr = &array[0];  int \*ptr2 = ptr + sizeof(int) \* 2; /\* sai \*/  printf("%d %d\n", \*ptr, \*ptr2);  return 0;  } |

Mã này thực hiện mở rộng thêm trong việc tính toán con trỏ được gán cho ptr2. Nếu sizeof(int)là 4, điển hình trong môi trường 32 bit hiện đại, biểu thức này là viết tắt của "8 phần tử sau array[0]", nằm ngoài phạm vi và nó gọi hành vi không xác định .

Để ptr2 trỏ tới vị trí 2 phần tử sau array[0], bạn chỉ cần thêm 2.

|  |
| --- |
| #include <stdio.h>  int main(int argc, char\* argv[]) {  int array[] = {1, 2, 3, 4, 5};  int \*ptr = &array[0];  int \*ptr2 = ptr + 2;  printf("%d %d\n", \*ptr, \*ptr2); /\* "1 3" sẽ được in ra \*/  return 0;  } |

Phép tính con trỏ rõ ràng bằng các toán tử cộng có thể gây nhầm lẫn, vì vậy sử dụng chỉ số mảng có thể tốt hơn.

|  |
| --- |
| #include <stdio.h>  int main(int argc, char\* argv[]) {  int array[] = {1, 2, 3, 4, 5};  int \*ptr = &array[0];  int \*ptr2 = &ptr[2];  printf("%d %d\n", \*ptr, \*ptr2); /\* "1 3" sẽ được in ra \*/  return 0;  } |

E1[E2] tương đồng với (\*((E1)+(E2))) (N1570 6.5.2.1, đoạn 2), và &(E1[E2]) tương đương với ((E1)+(E2)) (N1570 6.5.3.2, ghi chú 102).

Hoặc nếu thích sử dụng toán tử con trỏ, việc chuyển đổi con trỏ để trỏ tới một kiểu dữ liệu khác có thể cho phép địa chỉ byte. Tuy nhiên, cần thận trọng: : endianness có thể trở thành vấn đề, và chuyển đổi sang các kiểu dữ liệu khác ngoài 'con trỏ tới ký tự' có thể dẫn đến strict aliasing problems

|  |
| --- |
| #include <stdio.h>  int main(int argc, char\* argv[]) {  int array[3] = {1,2,3}; // 4 bytes \* 3 allocated  unsigned char \*ptr = (unsigned char \*) array; // unsigned chars only take 1 byte  /\*  \* Now any pointer arithmetic on ptr will match  \* bytes in memory. ptr can be treated like it  \* was declared as: unsigned char ptr[12];  \*/  return 0;  } |

## CHÚ THÍCH NHIỀU DÒNG KHÔNG THỂ LỒNG NHAU

Trong ngôn ngữ C, bình luận nhiều dòng bằng /\* và \*/ không thể lồng nhau.

Nếu bạn chú thích một khối mã hoặc hàm bằng kiểu bình luận này:

|  |
| --- |
| /\*  \* max(): Tìm số nguyên lớn nhất trong một mảng và trả về nó.  \* Nếu độ dài mảng nhỏ hơn 1, kết quả là không xác định.  \* arr: Mảng các số nguyên cần tìm.  \* num: Số lượng số nguyên trong arr.  \*/  int max(int arr[], int num)  {  int max = arr[0];  for (int i = 0; i < num; i++)  if (arr[i] > max)  max = arr[i];  return max;  } |

Bạn sẽ không thể dễ dàng chú thích nó ra:

|  |
| --- |
| // Cố gắng chú thích khối...  /\*  /\*  \* max(): Tìm số nguyên lớn nhất trong một mảng và trả về nó.  \* Nếu độ dài mảng nhỏ hơn 1, kết quả là không xác định.  \* arr: Mảng các số nguyên cần tìm.  \* num: Số lượng số nguyên trong arr.  \*/  int max(int arr[], int num)  {  int max = arr[0];  for (int i = 0; i < num; i++)  if (arr[i] > max)  max = arr[i];  return max;  }  // Gây lỗi ở dòng dưới...  \*/ |

Một giải pháp là sử dụng kiểu chú thích của C99:

|  |
| --- |
| // max(): Tìm số nguyên lớn nhất trong một mảng và trả về nó.  // Nếu độ dài mảng nhỏ hơn 1, kết quả là không xác định.  // arr: Mảng các số nguyên cần tìm.  // num: Số lượng số nguyên trong arr.  int max(int arr[], int num)  {  int max = arr[0];  for (int i = 0; i < num; i++)  if (arr[i] > max)  max = arr[i];  return max;  } |

Bây giờ, toàn bộ khối mã có thể được chú thích ra dễ dàng:

|  |
| --- |
| /\*  // max(): Tìm số nguyên lớn nhất trong một mảng và trả về nó.  // Nếu độ dài mảng nhỏ hơn 1, kết quả là không xác định.  // arr: Mảng các số nguyên cần tìm.  // num: Số lượng số nguyên trong arr.  int max(int arr[], int num)  {  int max = arr[0];  for (int i = 0; i < num; i++)  if (arr[i] > max)  max = arr[i];  return max;  }  \*/ |

Một giải pháp khác là tránh vô hiệu hóa mã bằng cách sử dụng các chỉ thị tiền xử lý #ifdef hoặc #ifndef thay vì cú pháp bình luận. Các chỉ thị này có khả năng lồng nhau, để bạn tự do chú thích mã của bạn theo kiểu bạn ưa thích.

|  |
| --- |
| #define DISABLE\_MAX /\* Xóa hoặc chú thích dòng này để bật khối mã max() \*/  #ifdef DISABLE\_MAX  /\*  \* max(): Tìm số nguyên lớn nhất trong một mảng và trả về nó.  \* Nếu độ dài mảng nhỏ hơn 1, kết quả là không xác định.  \* arr: Mảng các số nguyên cần tìm.  \* num: Số lượng số nguyên trong arr.  \*/  int max(int arr[], int num)  {  int max = arr[0];  for (int i = 0; i < num; i++)  if (arr[i] > max)  max = arr[i];  return max;  }  #endif |

Một số hướng dẫn thậm chí khuyến nghị rằng các phần mã không bao giờ được chú thích và nếu mã phải tạm thời bị vô hiệu hóa, bạn có thể sử dụng chỉ thị tiền xử lý #if 0.

Xem #if 0 để tắt các phần code

## BỎ QUA GIÁ TRỊ TRẢ VỀ CỦA CÁC HÀM THƯ VIỆN

Hầu hết mọi hàm trong thư viện chuẩn C đều trả về một giá trị thành công và một giá trị lỗi. Ví dụ, hàm malloc sẽ trả về con trỏ đến khối bộ nhớ được cấp phát bởi hàm khi thành công và, nếu hàm không thể cấp phát khối bộ nhớ yêu cầu, trả về con trỏ null. Do đó, bạn luôn nên kiểm tra giá trị trả về để dễ dàng gỡ lỗi hơn.

Đây là một ví dụ không tốt:

|  |
| --- |
| char\* x = malloc(100000000000UL \* sizeof \*x);  /\* thêm code \*/  scanf("%s", x); /\* Điều này có thể gây ra hành vi không xác định và nếu may mắn gây lỗi  vi phạm phân đoạn, trừ khi hệ thống của bạn có nhiều bộ nhớ \*/ |

Đây là một ví dụ tốt:

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  int main(void)  {  char\* x = malloc(100000000000UL \* sizeof \*x);  if (x == NULL) {  perror("malloc() failed");  exit(EXIT\_FAILURE);  }  if (scanf("%s", x) != 1) {  fprintf(stderr, "could not read string\n");  free(x);  exit(EXIT\_FAILURE);  }  /\* Thực hiện công việc với x. \*/  /\* Dọn dẹp. \*/  free(x);  return EXIT\_SUCCESS;  } |

Bằng cách này, bạn biết ngay lập tức nguyên nhân gây ra lỗi, nếu không bạn có thể dành hàng giờ để tìm kiếm lỗi ở một nơi hoàn toàn sai.

## SO SÁNH SỐ DẤU CHẤM ĐỘNG

Các loại số dấu chấm động (float, double và long double) không thể biểu diễn chính xác một số số do chúng có độ chính xác hữu hạn và biểu diễn các giá trị trong một định dạng nhị phân. Tương tự như chúng ta có các số thập phân lặp lại trong cơ số 10 cho các phân số như 1/3, cũng có các phân số không thể biểu diễn được theo cách hữu hạn trong cơ số nhị phân (như 1/3, nhưng quan trọng hơn là 1/10). Không nên so sánh trực tiếp các giá trị số dấu chấm động; hãy sử dụng một giá trị delta thay thế.

|  |
| --- |
| #include <float.h> // cho DBL\_EPSILON và FLT\_EPSILON  #include <math.h> // cho fabs()  int main(void)  {  double a = 0.1; // không chính xác: (nhị phân) 0.000110...  // có thể sai hoặc đúng  if (a + a + a + a + a + a + a + a + a + a == 1.0) {  printf("10 \* 0.1 thực sự là 1.0. Điều này không được đảm bảo trong trường hợp tổng quát.\n");  }  // Sử dụng một giá trị delta nhỏ.  if (fabs(a + a + a + a + a + a + a + a + a + a - 1.0) < 0.000001) {  // C99 5.2.4.2.2p8 đảm bảo ít nhất 10 chữ số thập phân  // của độ chính xác cho kiểu double.  printf("10 \* 0.1 gần như là 1.0.\n");  }  return 0;  } |

Ví dụ khác

|  |
| --- |
| #include <stdio.h>  #include <math.h>  static inline double rel\_diff(double a, double b)  {  return fabs(a - b) / fmax(fabs(a), fabs(b));  }  int main(void)  {  double d1 = 3.14159265358979;  double d2 = 355.0 / 113.0;    double epsilon = 1.0;  for (int i = 0; i < 10; i++)  {  if (rel\_diff(d1, d2) < epsilon)  printf("%d:%.10f <=> %.10f trong ngưỡng %.10f (sự chênh lệch tương đối %.4E)\n",  i, d1, d2, epsilon, rel\_diff(d1, d2));  else  printf("%d:%.10f <=> %.10f vượt quá ngưỡng %.10f (sự chênh lệch tương đối %.4E)\n",  i, d1, d2, epsilon, rel\_diff(d1, d2));  epsilon /= 10.0;  }  return 0;  } |

Kết quả

|  |
| --- |
| 0:3.1415926536 <=> 3.1415929204 trong ngưỡng 1.0000000000 (sự chênh lệch tương đối 8.4914E-08)  1:3.1415926536 <=> 3.1415929204 trong ngưỡng 0.1000000000 (sự chênh lệch tương đối 8.4914E-08)  2:3.1415926536 <=> 3.1415929204 trong ngưỡng 0.0100000000 (sự chênh lệch tương đối 8.4914E-08)  3:3.1415926536 <=> 3.1415929204 trong ngưỡng 0.0010000000 (sự chênh lệch tương đối 8.4914E-08)  4:3.1415926536 <=> 3.1415929204 trong ngưỡng 0.0001000000 (sự chênh lệch tương đối 8.4914E-08)  5:3.1415926536 <=> 3.1415929204 trong ngưỡng 0.0000100000 (sự chênh lệch tương đối 8.4914E-08)  6:3.1415926536 <=> 3.1415929204 trong ngưỡng 0.0000010000 (sự chênh lệch tương đối 8.4914E-08)  7:3.1415926536 <=> 3.1415929204 trong ngưỡng 0.0000001000 (sự chênh lệch tương đối 8.4914E-08)  8:3.1415926536 <=> 3.1415929204 vượt quá ngưỡng 0.0000000100 (sự chênh lệch tương đối 8.4914E-08)  9:3.1415926536 <=> 3.1415929204 vượt quá ngưỡng 0.0000000010 (sự chênh lệch tương đối 8.4914E-08) |

## CHỮ SỐ THẬP PHÂN DẤU CHẤM ĐỘNG MẶC ĐỊNH LÀ KIỂU DOUBLE

Khi làm việc với các chữ số thập phân dấu chấm động, cần lưu ý rằng mặc định chúng là kiểu double. Điều này có thể dẫn đến hành vi không mong muốn khi khởi tạo biến kiểu float bằng các giá trị chữ số thập phân dấu chấm động hoặc so sánh chúng với các giá trị chữ số thập phân dấu chấm động.

Ví dụ:

|  |
| --- |
| #include <stdio.h>  int main() {  float n;  n = 0.1;  if (n > 0.1) printf("Wierd\n");  return 0;  }  // Prints "Wierd" when n is float |

Ở đây, nđược khởi tạo và làm tròn thành độ chính xác duy nhất, dẫn đến giá trị 0,10000000149011612. Sau đó, nđược chuyển đổi trở lại độ chính xác kép để so sánh với 0.1chữ (tương đương với 0,10000000000000001), dẫn đến sự không khớp.

Bên cạnh lỗi làm tròn, việc trộn floatcác biến với doublechữ sẽ dẫn đến hiệu suất kém trên các nền tảng không có hỗ trợ phần cứng cho độ chính xác kép.

## SỬ DỤNG HẰNG KÍ TỰ THAY VÌ CHUỖI KÍ TỰ VÀ NGƯỢC LẠI

Trong ngôn ngữ C, hằng ký tự và chuỗi ký tự là hai khái niệm khác nhau.

Hằng ký tự, được bao quanh bởi dấu nháy đơn như 'a', là một hằng số ký tự. Một hằng ký tự là một số nguyên có giá trị là mã ký tự đại diện cho ký tự tương ứng. Cách giải thích hằng ký tự chứa nhiều ký tự như 'abc' là được xác định bởi triển khai của ngôn ngữ.

Một chuỗi ký tự là một dãy ký tự được bao quanh bởi dấu nháy kép như "abc". Chuỗi ký tự là một mảng không thể thay đổi với các phần tử kiểu char. Chuỗi trong dấu nháy kép kèm theo ký tự null kết thúc là nội dung của chuỗi, do đó "abc" có 4 phần tử ({'a', 'b', 'c', '\0'}).

Trong ví dụ này, hằng ký tự được sử dụng trong trường hợp nên sử dụng chuỗi ký tự. Hằng ký tự này sẽ được chuyển đổi thành một con trỏ theo cách được xác định bởi triển khai của ngôn ngữ và khó có cơ hội để con trỏ đã chuyển đổi này hợp lệ, do đó ví dụ này sẽ gây ra lỗi hành vi không xác định.

|  |
| --- |
| #include <stdio.h>  int main(void) {  const char \*hello = 'hello, world'; /\* sai \*/  puts(hello);  return 0;  } |

Trong ví dụ này, một chuỗi ký tự được sử dụng trong trường hợp nên sử dụng hằng ký tự. Con trỏ được chuyển đổi từ chuỗi ký tự này sẽ được chuyển đổi thành một số nguyên theo cách được xác định bởi triển khai của ngôn ngữ, và sau đó sẽ được chuyển đổi thành char theo cách được xác định bởi triển khai của ngôn ngữ. (Cách chuyển đổi một số nguyên thành kiểu có dấu không thể biểu diễn giá trị cần chuyển đổi là do triển khai của ngôn ngữ quy định, và liệu kiểu char có phải là kiểu có dấu không cũng là do triển khai quy định.) Kết quả xuất ra sẽ là một cái gì đó vô nghĩa.

|  |
| --- |
| #include <stdio.h>  int main(void) {  char c = "a"; /\* sai \*/  printf("%c\n", c);  return 0;  } |

Hầu hết các trình biên dịch đều sẽ phát ra cảnh báo về các lỗi này. Nếu trình biên dịch không cảnh báo, bạn cần sử dụng nhiều tùy chọn cảnh báo hơn, hoặc đề xuất sử dụng trình biên dịch tốt hơn.

## HÀM ĐỆ QUY – THIẾU ĐIỀU KIỆN CƠ SỞ

Việc tính giai thừa của một số là một ví dụ cổ điển về hàm đệ quy.

**Thiếu Điều Kiện Cơ Sở:**

|  |
| --- |
| #include <stdio.h>  int factorial(int n)  {  return n \* factorial(n - 1);  }  int main()  {  printf("Giai thừa của %d = %d\n", 3, factorial(3));  return 0;  } |

Đầu ra điển hình:**Segmentation fault: 11**

Vấn đề với chức năng này là nó sẽ lặp vô hạn, gây ra lỗi phân đoạn — nó cần một điều kiện cơ sở để dừng đệ quy.

**Điều kiện cơ sở được khai báo:**

|  |
| --- |
| #include <stdio.h>  int factorial(int n)  {  if (n == 1) // Điều kiện cơ sở, rất quan trọng trong thiết kế hàm đệ quy.  {  return 1;  }  else  {  return n \* factorial(n - 1);  }  }  int main()  {  printf("Giai thừa của %d = %d\n", 3, factorial(3));  return 0;  } |

Kết quả mẫu:

|  |
| --- |
| Giai thừa của 3 = 6 |

Hàm này sẽ kết thúc ngay khi đạt đến điều kiện n bằng 1 (miễn là giá trị ban đầu của n đủ nhỏ - giới hạn trên là 12 khi int có kích thước 32 bit).

**Các quy tắc cần tuân theo:**

1. Khởi tạo thuật toán. Chương trình đệ quy thường cần một giá trị khởi đầu để bắt đầu. Điều này có thể được thực hiện thông qua việc sử dụng tham số được truyền vào hàm hoặc thông qua việc cung cấp một hàm cổng không đệ quy nhưng thiết lập giá trị khởi đầu cho tính toán đệ quy.
2. Kiểm tra xem giá trị hiện tại đang xử lý có khớp với trường hợp cơ sở hay không. Nếu có, xử lý và trả về giá trị.
3. Định lại câu trả lời dưới dạng một vấn đề con nhỏ hơn hoặc đơn giản hơn.
4. Chạy thuật toán trên vấn đề con.
5. Kết hợp các kết quả để tạo thành câu trả lời.
6. Trả về kết quả.

Nguồn: Hàm Đệ Quy

## VƯỢT QUA GIỚI HẠN MẢNG

Mảng trong C được đánh chỉ mục bằng số 0, có nghĩa là chỉ mục luôn bắt đầu từ 0 và kết thúc với chỉ mục bằng độ dài của mảng trừ 1. Do đó, mã sau sẽ không xuất ra phần tử đầu tiên của mảng và sẽ xuất ra giá trị ngẫu nhiên cho giá trị cuối cùng mà nó in ra.

|  |
| --- |
| #include <stdio.h>  int main(void)  {  int x = 0;  int myArray[5] = {1, 2, 3, 4, 5}; // Khai báo 5 phần tử  for(x = 1; x <= 5; x++) // Lặp từ 1 đến 5.  printf("%d\t", myArray[x]);  printf("\n");  return 0;  } |

Kết quả: 2 3 4 5 Giá\_trị\_ngẫu\_nhiên

Mã dưới đây mô tả cách đạt được kết quả mong muốn một cách chính xác:

|  |
| --- |
| #include <stdio.h>  int main(void)  {  int x = 0;  int myArray[5] = {1, 2, 3, 4, 5}; // Khai báo 5 phần tử  for(x = 0; x < 5; x++) // Lặp từ 0 đến 4.  printf("%d\t", myArray[x]);    printf("\n");  return 0;  } |

Kết quả: 1 2 3 4 5

Quan trọng là biết độ dài của mảng trước khi làm việc với nó, nếu không bạn có thể gây hỏng buffer hoặc gây ra lỗi chia cho không bằng cách truy cập vào các vị trí bộ nhớ nằm ngoài phạm vi.

## TRUYỀN MẢNG KHÔNG KỀ NHAU VÀO HÀM MONG ĐỢI MẢNG ĐA CHIỀU “THỰC SỰ”

Khi cấp phát các mảng đa chiều bằng malloc, calloc và realloc, một mẫu phổ biến là cấp phát các mảng con bằng nhiều lệnh gọi (ngay cả khi lệnh gọi chỉ xuất hiện một lần, nó có thể nằm trong vòng lặp):

|  |
| --- |
| /\* Cũng có thể là `int \*\*` với malloc được sử dụng để cấp phát mảng ngoại. \*/  int \*array[4];  int i;  /\* Cấp phát 4 mảng có 16 phần tử int. \*/  for (i = 0; i < 4; i++)  array[i] = malloc(16 \* sizeof(\*array[i])); |

Sự khác biệt về byte giữa phần tử cuối cùng của một trong các mảng con và phần tử đầu tiên của mảng con tiếp theo có thể không phải là 0 như chúng sẽ là với một mảng đa chiều "thực sự" (ví dụ: int array[4][16];):

|  |
| --- |
| /\* 0x40003c, 0x402000 \*/  printf("%p, %p\n", (void \*)(array[0] + 15), (void \*)array[1]); |

Tính cả kích thước của int, bạn có được sự khác biệt là 8128 byte (8132-4), tương đương với 2032 phần tử mảng có kích thước int, và đó là vấn đề: một mảng đa chiều "thực sự" không có khoảng trống giữa các phần tử.

Nếu bạn cần sử dụng một mảng được cấp phát động với một hàm mong đợi một mảng đa chiều "thực sự", bạn nên cấp phát một đối tượng có kiểu int \* và sử dụng toán học để thực hiện tính toán:

|  |
| --- |
| void func(int M, int N, int \*array);  ...  int \*array;  int M = 4, N = 16;  array = calloc(M, N \* sizeof(\*array));  array[i \* N + j] = 1;  func(M, N, array); |

Nếu N là một macro hoặc một số nguyên nguyên thay vì một biến, mã có thể đơn giản sử dụng cú pháp mảng 2-D sau khi cấp phát con trỏ cho một mảng:

|  |
| --- |
| void func(int M, int N, int \*array);  #define N 16  void func\_N(int M, int (\*array)[N]);  ...  int M = 4;  int (\*array)[N];  array = calloc(M, sizeof(\*array));  array[i][j] = 1;  func(M, N, (int \*)array);  func\_N(M, array); |

Phiên bản ≥ C99

Nếu N không phải là một macro hoặc một số nguyên nguyên, thì array sẽ trỏ đến một mảng có độ dài biến (VLA). Điều này vẫn có thể sử dụng với func bằng cách ép kiểu thành int \* và một hàm mới func\_vla sẽ thay thế func\_N:

|  |
| --- |
| void func(int M, int N, int \*array);  void func\_vla(int M, int N, int array[M][N]);  ...  int M = 4, N = 16;  int (\*array)[N];  array = calloc(M, sizeof(\*array));  array[i][j] = 1;  func(M, N, (int \*)array);  func\_vla(M, N, array); |

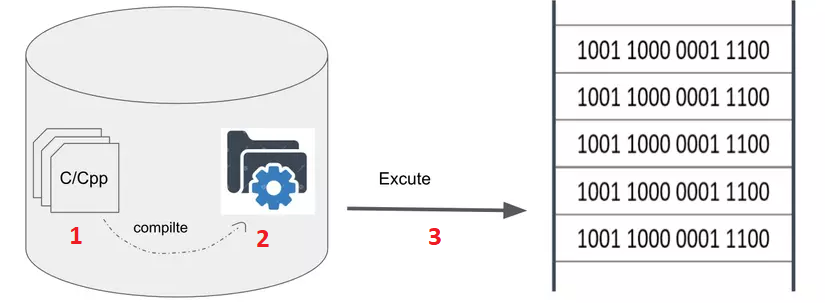
Phiên bản ≥ C11

Lưu ý: VLA là tùy chọn từ phiên bản C11. Nếu triển khai của bạn hỗ trợ C11 và x

# MEMORY LAYOUT

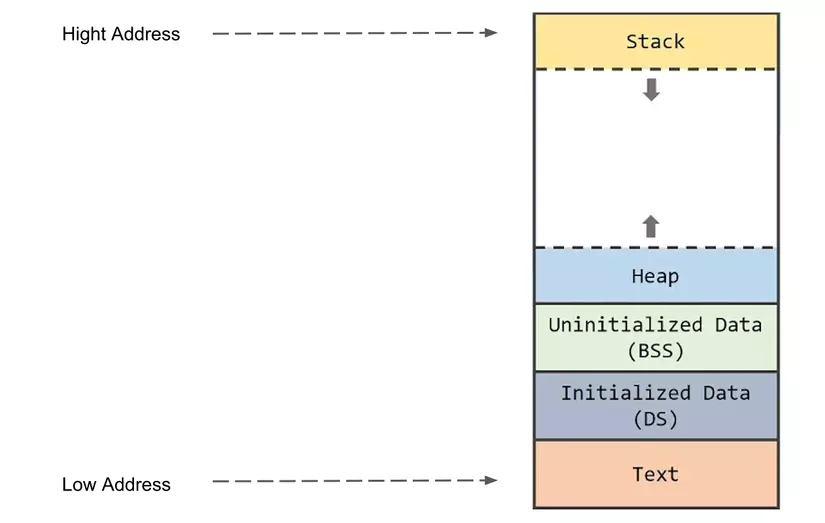
## Process In Program

1. Khi chúng ta viết 1 chương trình C/C++ , source code của chúng ta sẽ được lưu trong ổ cứng.
2. Sau đó khi chúng ta Compile source code (build), hệ thống sẽ tạo ra file cài đặt và file này cũng được lưu trong ổ cứng.
3. Cuối cùng là quá trình chúng ta run file program, thực chất đây là quá trình chúng ta load chương trình vào Memory và Excute.



**Hình 1**: Process Program

## Memory Layout In Program



**Hình 2**: Memory Layout In C

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

### Text Segment

Text Segment ở vùng nhớ của địa chỉ thấp nhất, mã máy trong trương trình sẽ được lưu trữ tại đây.

Vùng lưu trữ này chỉ có thể đọc.

### Initialized Data Segment (DS)

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

**VD:**

|  |
| --- |
| int global = 100;  int foo() {  static int number = 10;  return 0;  } |
| Trong đoạn chương chính trên, biến global được khởi tạo với giá trị 100, và biến **static** number được khởi với giá trị 10 bởi programmer nên được lưu trữ vào Initialized Data Segment. |

### Uninitialized Data Segment (BSS)

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

**VD:**

|  |
| --- |
| int global;  int foo() {  static int number = 0;  return 0;  } |
| Trong đoạn chương trình trên, biến global **không được khởi tạo giá trị** mặc định và biến **static** number **được khởi tạo với giá trị bằng 0** sẽ được lưu trữ vào Uninitialized Data Segment. |

### Heap (Dynamic Memory Allocation)

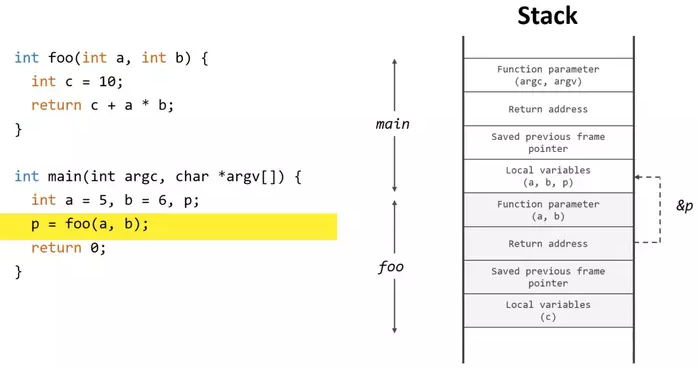
Trong C/C++ chúng ta có thể hoàn toàn control được quá trình cấp phát hoặc giải phóng bộ nhớ bằng các lệnh như malloc, calloc, relloc, free, new, delete, ... Vùng nhớ được cấp phát chính là HEAP, vùng nhớ này sẽ phình lên ( grows upward ) mỗi khi bạn cấp phát. Khi sử dụng xong các bạn phải delete vùng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

### Stack (Automatic Variable Storage)

Stack là một vùng nhớ được cấp phát tự động và có cấu trúc LIFO (Last In First Out) để chứa các Function Frame. Mỗi khi chương trình được gọi, thì các Function Frame sẽ được gọi và push vào trong stack.

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

**VD:** Function Frame có cấu trúc như ví dụ dưới đây.



Khi main() được gọi, function frame của main() sẽ được push vào stack, và khi function foo() được gọi thì function frame của foo() sẽ được push vào.

Cấu trúc của một function frame gồm bốn phần chính: Function Parameter, Return Address, Saved Previus Frame Pointer, Local Variable.

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